A Feature Composition Problem and a Solution Based on C++ Template Metaprogramming

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 usually a suitable concept for library implementation: it results in easily scalable and maintainable code. After specifying and implementing orthogonal features, we aim to easily assemble library components. In real life, components can be used only after appropriate refinement steps, progressively adding features in each step. Therefore the specific solution for a particular task can be produced by composing a set of refined components. Unfortunately, a subtype anomaly occurs in object-oriented languages between such composite components that have different numbers of features from different refinement stages. In this article we analyse this anomaly that we named chevron-shape inheritance and present a framework based on standard C++ template metaprogramming.

1 Introduction

The creation of large scale software systems is still a critical challenge 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 already exist as some foundation library module, otherwise they can be produced by reasonable efforts. This *separation of concerns* is widely discussed in [25] and [12]. In object-oriented libraries these concerns are mostly implemented as separate classes.

Possessing our premanufactured components we have several methodologies to assemble a full system from the required components. This so-called *collaboration based design* is supported by aspect oriented programming [21], subject oriented programming [27] [28], feature oriented programming [10] and composition filters [19]. Besides, the assembly can be naturally expressed by deriving from all required components using multiple inheritance in languages that support this feature, such as C++. This *mixin-based*[1] technique is highly attractive for implementing collaboration-based design [13]. Whichever approach we

[1] There is a number of different meanings of "mixins". We use the term *mixin* according to Batory and Smaragdakis [13].

choose, the basic idea is to easily create the solution as a union of components implemented in separate modules.

However, in real life it is hard to find a component that implements the required feature *exactly*. In most cases we have to customize the components to meet the requirements of the current task by adding *features* [12]. Specializations for every separate component are made orthogonally which leads to separated refinement chains, each representing refinement steps of individual features. The specific solution for a particular task can be finally produced by assembling specialized components from appropriate layers of different chains.

In object-oriented languages we represent our components as classes. Refinement is regularly expressed by inheritance, hence we gain a subtype relationship between the refined and the original component. In [11], Batory et al. claim that "the only classes that are instantiated in a synthesized application are the terminal classes of the refinement chains [...] Non-terminal classes [...] are *never* instantiated." Nevertheless, it is a frequent scenario that some client code handles the objects of terminal classes through an interface of an earlier refinement layer. For example, the client code has been implemented prior to the final refinement steps or the refinements serving implementational purposes only should be hidden. In such cases, subtyping should be provided between refinement layers.

In this article we present examples where conventional subtyping yields undesired effects, give a detailed discussion of an inheritance anomaly, dispute existing or proposed alternative solutions and introduce a solution based on code transformation using standard C++ template metaprogramming.

2 Problem description

To define the problem, we rely on the notation and formalism of AHEAD [11]. In this model, we consider features as refinement transformations. Every feature f_i is a function that transforms a (possibly empty) component. As a result, the represented feature will be added to the component. Assuming an existing component c_i, we mark such extensions by $f_i \bullet c_i$.

For an easier creation of programs, we allow the composition[2] of features. A composition is represented by a set, e.g. a union of features is $\{f_i, f_j\}$. An extension of a compound component by a set of features is formalized as

$$f \bullet c = \{f_1, ..., f_n\} \bullet \{c_1, ..., c_n\} = \{f_1 \bullet c_1, ..., f_n \bullet c_n\} \qquad (1)$$

Note that feature addition is distributive over the composition above [11]. In widely used object-oriented languages, such as Java, C++, C# or Eiffel, feature addition (component refinement) is implemented by inheritance. Additionally, composition may also be implemented by multiple inheritance[3] or aggregation.

[2] Such compositions are referred as *collectives* in [11].

[3] 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.

What is the problem with inheritance? In object-oriented languages, reusing code and defining subtype relations is not clearly separated, they are both expressed through inheritance. Thus, composition implemented by multiple inheritance or aggregation fails to fulfil the distributive property. Hence the forementioned languages does not conform to equation (1).

The service for a specific user requirement can be constructed as a composition of refined features. In the same time, we should be able to use any subset of features from different refinement stages as an interface to this service. Therefore a subtype relation should be provided between any of these collectives irrespectively of the number and refinement level of participant concern classes. Thus we have to decide: if we derive the refined collaboration from the original collaboration class, we lose the subtype relationship with classes implementing the refined features; otherwise (deriving from the refined features) we lose the subtype relationship with the original collective. In figure 1 the general structure of the anomaly can be seen, according to the two mentioned cases respectively. We have named this anomaly *chevron-shape inheritance.*

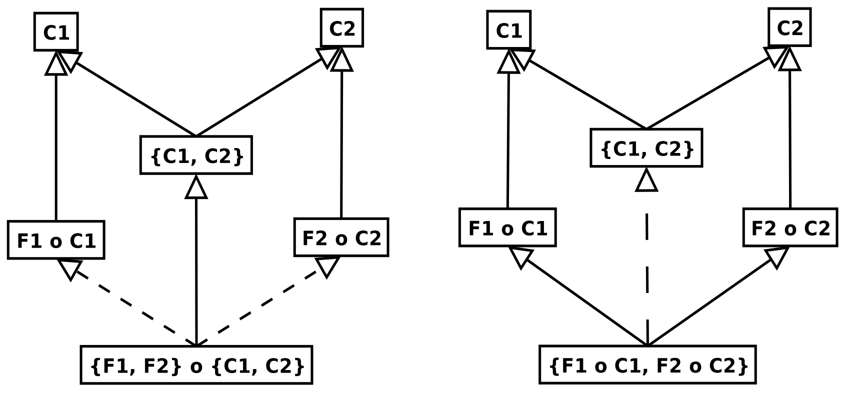


Fig. 1. Chevron-shape inheritance. (Missing subtypes are marked by dashed lines)

3 Examples

To clarify the formalism above, let us introduce an example from the C++ Standard Library. In Fig. 2 you can see the class hierarchy of the stream implementation in GNU C++ as specified by the C++ standard. (We omit the fact that all the following classes are *templates* by the standard, because this does not affect the problem).

Classes `istream` and `ostream` are representing input and output streams as orthogonal concerns. (There is a common base class `ios` for both classes that holds some general stream functionality.) Class `iostream` unifies input and output functionalities representing streams that can be both read and written. Using

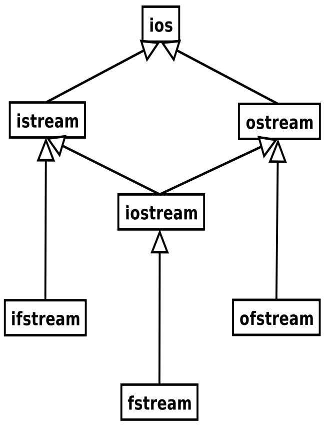


Fig. 2. I/O library according to the C++ standard

the formalism introduced above, $iostream = \{istream, ostream\}$. `iostream` is implemented by multiple inheritance from classes `istream` and `ostream`. The result is a well known anomaly called *diamond-shape inheritance*. It is usually resolved using *virtual inheritance* in classes `istream` and `ostream`.

The library contains refinements for both input and output streams. Streams opened over certain physical devices belong to classes `ifstream` or `ofstream` as refinements of `istream` and `ostream` respectively. Formalising this, we gain $ifstream = fileio \bullet istream$, class `ofstream` can be defined analogously. Similar refinements exist for streams stored in a memory buffer (e.g. `istringstream` and `ostringstream`). These refinements are implemented using inheritance. Class `fstream` (and `stringstream` also) inherits from `iostream` and represents file streams for both input and output operations. For `fstream`, we gain $fstream = fileio \bullet iostream$. Until this point, the class hierarchy is specified by the C++ standard.

Surprisingly, this construction causes some unexpected results. Intuitively, `fstream` is clearly a subtype of both `ifstream` and `ofstream`, so $fstream = \{ifstream, ofstream\}$. The inheritance hieararchy described above does not express this, hence there is no conversion from `fstream` to either `istream` or `ostream`. Thus,

$$\begin{aligned} fstream &= fileio \bullet iostream = fileio \bullet \{istream, ostream\} \\ &\neq \{fileio \bullet istream, fileio \bullet ostream\} = \{ifstream, ofstream\} \end{aligned} \tag{2}$$

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 examining classes `iostream` and `istream` this may be an astonishing fact.

There is another possible construction scheme for the I/O library that is described in [29] and also referred in Stroustrup's fundamental book *The Design* *and Evolution of C++* [17]. Classes ifstream and ofstream are derived from ifstream and ostream respectively, and also from class fstreambase, which represents an orthogonal, third concern (file operations). Here, fstream is derived from ifstream and ofstream, therefore our previous problem is substituted with another one: iostream is not an ancestor of fstream anymore, therefore cannot be used as an interface for input and output file operations.

The implementation technique of file streams is not covered by the standard. Examining certain implementations like the one in the old GNU C++ version 2.95 we get an even more confusing picture (see figure 3). The problem arises at the implementation of file streams. Since all file streams handle files, it is highly attractive to detach file-specific functionality into fstreambase. The consequence of this structure is a kind of mixture of the two previous approaches: ifstream and ofstream are descendents of istream and ostream respectively, and fstreambase like in [29]. However, fstream is inherited from iostream and fstreambase as in the current standard.

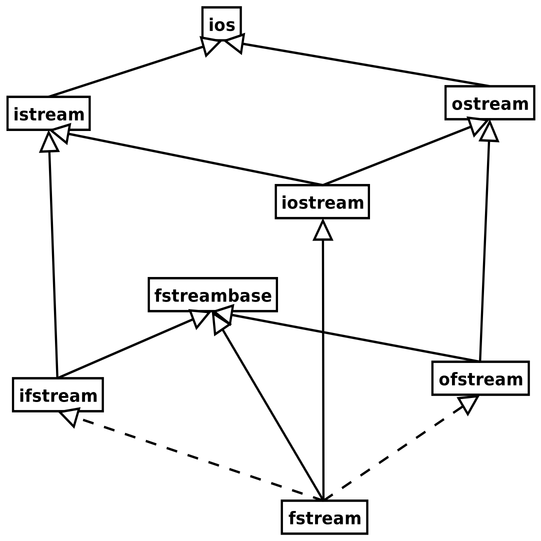


Fig. 3. The GNU implementation of the I/O library

The current C++ standard votes for the first solution. No matter, which one we choose, disturbing gaps remain in subtype relations between refinement stages. It seems we can not express the whole subtype graph that the user would find natural.

Another example is from the Eiffel programming language [18]. 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 provided by the compiler, we have to resolve it by hand creating conversion functions.

4 Classical Approaches

In this section we discuss several widely used methods that may promise a possible solution for the anomaly and analyze the results.

Virtual Inheritance (opposed to conventional inheritance) guarantees that when a class occurs as a superclass several times, its members will be not duplicated in the descendants. Virtual inheritance can usually solve issues related to multiple inheritance and with a combination of abstract classes it supports a programming style where the abstract bases define interfaces and several derived classes contribute to the implementation [17] [16].

Virtual inheritance has several drawbacks in our case. Besides having memory and runtime penalties, we must explicitly mark our intention to use a class as a *virtual base*, hence it is intrusive and can not be a solution using precompiled libraries. Additionally, in the case of several feature refinement chains, the number of possible collectives grows exponentially, only an automatic mechanism provides an acceptable solution, thus it provides a suitable solution only for a small number of features.

Signatures play an important role in certain functional languages, like Standard ML. A signature prescribes the typenames, values and nested structures that must appear in a structure. That way signatures constrain the contents of structures [7].

Signatures for C++ were proposed by Gerald Baumgartner [4]. They provide a facility similar to interfaces, but in a non-intrusive way. Signatures have features similar to classes, e.g. they can inherit from other signatures, and a compiler can check whether a class has all members to meet the requirements of a signature. However, using signatures, unintentional conversions may occur: though it is conceptionally wrong, a `Gun` can be cast to a `Camera` because they both have function `shoot()` and signatures ignore any semantic information. Additionally, signatures are non-standard language extensions for C++.

Structural subtyping binds the subtype relation to data structures instead of inheritance. Languages like C++, Java or Eiffel declare subtyping at the point of class definition. Contrary, subtype relations are based on structural subtyping in many functional languages where existence of subtype relations can be decided based on structural conformance. The reader can find a well known implementation in the Ocaml language [8]. We suggest reading [20] on the theoretical background of structural subtyping.

Structural subtyping provides an excellent solution to our anomaly: in languages supporting this feature our anomaly does not exist. Unfortunately, structural subtyping suffers from the problem of accidental conformance the same way as signatures do. Furthermore, no widely used object-oriented language provides structural subtyping. Recently, several attempts were made to unify the object oriented and structural approaches, see [9].

Aspects address the subtyping problem a different way than structural subtyping. Instead of providing an algorithmic model to implicitly deduce subtype relations, another approach is to provide a language mechanishm external to the class definition that establishes a subtyping relation [23]. Aspect-oriented programming systems, such as AspectJ [22] allow modification of types independently of their original definitions. For example, an existing class can be modified to implement a newly created interface using static cross-cutting [21]. Though aspects are usable to weave a single feature into a hierarchy, we find the same subtyping anomaly when aspects have their own refinement chains.

5 Implementation

Beside object-orientation, the C++ language also has a rich feature set for supporting generative programming. C++ templates provide parametric polymorphism as an extension to inclusion polymorphism provided by inheritance. We can create template specializations to have a completely different implementation from the general one for some special template arguments. Thus we can create a matrix class that stores elements in a plain array except for booleans, where it stores an array of chars each representing (mostly) eight booleans. Because booleans can be passed as template arguments, we can specialize upon compile time conditions. Specializations also allow us to write algorithms running in compilation time inside the compiler instead of runtime in the program. This approach is called template metaprogramming [5].

Theoretically, template metaprogramming in C++ is a Turing complete language in itself, therefore any algorithm can be expressed as a metaprogram (see [15]). Practically, compilers have limitations in resources (e.g. a maximal depth of recursion during template instantiation) so this possibility must be used with care. Additionally, programming compile time algorithms is still an uncomfortable effort lacking standard libraries and debug tools. However, it is very useful for simple cases, especially to give a performance boost. (See expression templates [6]).

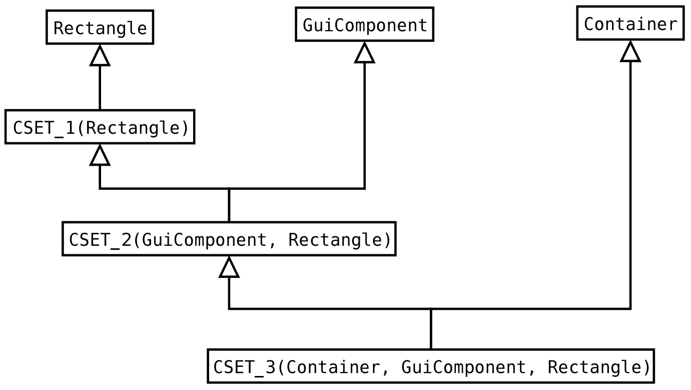
5.1 CSet

The template metaprogramming features discussed above enable us to solve the chevron-shape anomaly described in section 2. To achieve this goal we perform an automatic transformation to simulate a subtype relationship between collectives: based on the possibilities of template metaprogramming we implement automatic conversions between them. In the remaining part of the article we call these sets `CSets` (where `C` can be pronounced as any of class, concern, collaboration, collective, chevron, etc, as conceptually needed). Presenting the technical implementation details is out of the scope of this paper, it can be found in [1]. In this article we concentrate on usage and applicability of the `CSet` framework in feature-oriented programming.

Creating `CSet`, our first task is to assemble the collaborating classes into a single entity which we implement as a C++ class using multiple inheritance. Class `CSet` is written to directly inherit from all classes in a recursive way, according to the recursive structure of typelists.

```
// --- Inherit from all types in list
typedef TYPELIST_3(Container, Rectangle, GuiComponent) WindowList;
typedef CSet<WindowList> Window;
Window win;
win.add( Button("OK") ); // --- Container method
win.move(13, 42);        // --- Rectangle method
win.draw();              // --- GuiComponent method
```

Above we assemble three features into a `CSet` called `Window`: `Container`, `Rectangle` and `GuiComponent`. The structure of the created `CSet` can be seen in Fig. 4. After having a window object we can call the methods of all three classes.



Rectangle
GuiComponent
Container
CSET_1(Rectangle)
CSET_2(GuiComponent, Rectangle)
CSET_3(Container, GuiComponent, Rectangle)

Fig. 4. Example for a CSet hierarchy

The main issue in `CSet` is the support of conversions between all appropriate collectives. The implementation is based on the fact that constructors are ordinary functions in C++, therefore they can be defined to be templates as well. This way we can make the conversion in elementary recursive steps using built-in conversions provided by the compiler. Because the conversion is built into constructors, it can be used in a completely transparent form without any function calls for conversion:

```
// --- Conversion using the constructor
CSET_2(GuiComponent, Shape) widget(win);
```

```
widget = win; // --- or the assignment operator
```

Note that `CSet` has `CSET_N` macros similar to `TYPELIST_N` providing an easier form of definition. Our object `win` can be converted to the collective of features `GuiComponent` and `Shape` because `win` itself is an instance of `GuiComponent`, furthermore it also can be converted to a shape since `win` has feature `Rectangle` which is refined from `Shape`. Thus the conversion is legal and the object `widget` can be initialized using `win`.

Similarly to the constructor, the assignment operator can be defined as a template, too. Template functions provide another advantage: not only `CSets`, but other user objects can be converted with these functions.

```
// --- Create a user class and an instance
struct MyWindow : public GuiComponent,
    public Circle, public Vector { ... };
MyWindow myObj;

// --- Conversion from user object
CSET_2(Shape, GuiComponent) widget(myObj);
```

5.2 Dynamic binding

So far we are able to perform appropriate conversions between matching `CSets`. Unfortunately these conversions are done by value. This may imply the loss of dynamic data of the converted object which is often called *slicing* in C++. To avoid slicing, we have to convert our objects by pointer or reference. We follow the conventional way in our implementation and create our own smart pointers and references. Because the implementation and usage of these classes are very similar, we introduce only our smart pointer class called `CSetPtr`.

Conversions using class `CSetPtr` can be written the same way as with `CSet`, but using pointers we bind dynamically.

```
// --- Conversion using the constructor
CSETPTR_2(GuiComponent, Shape) widgetPtr(win);

widgetPtr = win; // --- or assignment operator
```

In `CSetPtr` we aggregate pointer data members instead of inheriting from ancestors to implement elementary conversion steps. As a result, the structure of `CSetPtr` created by the previous definition is completely different from the structure of an appropriate `CSet`. The created hierarchy and the essence of dynamic binding can be seen in figure 5.

`CSetPtr` holds a pointer member for each type in the set, so every pointer can be set to the appropriate part of an adequate `CSet` object. This way we can utilize dynamic binding provided by conventional pointers in C++. `CSetPtr` can be transparently converted to any of its pointer members, so virtual functions can be called easily.

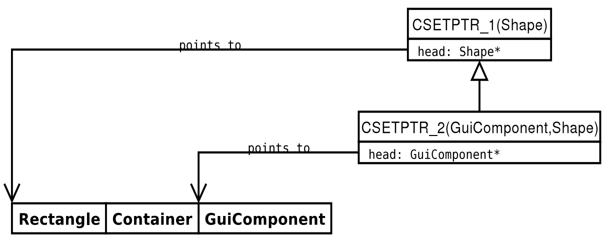


Fig. 5. Example for a CSetPtr hierarchy

```
Shape *shape = widgetPtr;
shape->draw(); // --- Use dynamic binding
```

The difference between `CSetPtr` and `CSetRef` comes from the type of their data members: `CSetRef` holds references instead of pointers. Thus it must be initialized and conversions are done by reference or value after initialization.

5.3 Limitations and Further Work

Though our solution relies on standard C++ features only, we encountered problems with some compilers regarding conformance to the standard. Aged compilers tend to fail providing language requirements like partial template specialization or has unacceptable compile time, exponentially growing by the number of composed features. Hopefully these problems will disappear in new compiler versions.

Another kind of problems is related to our implementation of feature composition. While we exploit the advantages of multiple inheritance, we also suffer from its usual drawbacks like possible name resolution disambiguities.

Our current version does not provide `const` correctness which is an essential language feature to improve semantical correctness of complex C++ programs, e.g. `const` member functions, pointers to `const` data or `const_iterator`.

In our future work we plan to further improve our solution by eliminating the problems enlisted above.

6 Summary

The subtyping mechanism of current object oriented languages is not flexible enough to express required subtype relationships that arise at the implementation of collaboration based designs. We described an anomaly called chevron-shape inheritance which arises assembling collectives created during the stepwise refinement of features. We have introduced a framework called `CSet` based on C++ template metaprogramming to transform the subtyping mechanism of the C++ language. `CSets` make subtype relationships created during refinement distributive with feature composition. It supports coercion polymorphism between appropriate collectives and 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.

References

1. István Zólyomi, Zoltán Porkoláb, Tamás Kozsik: An extension to the subtype relationship in C++. GPCE 2003, LNCS 2830 (2003), pp. 209 - 227.
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)
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, 1995, pp. 36-43.
6. Todd Veldhuizen: Expression Templates. C++ Report vol. 7, no. 5, 1995, pp. 26-31.
7. Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock: A Comparative Study of Language Support for Generic Programming. Proceedings of the 18th ACM SIGPLAN OOPSLA 2003, pp. 115-134.
8. Leroy, Xavier et al.: The Objective Caml system, release 3.0.8 (July 2004), documentation and user's manual. http://caml.inria.fr/ocaml/htmlman/index.html
9. Jeremy Siek: A Language for Generic Programming. PhD thesis, Indiana University, August 2005.
10. Don Batory: A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. Technical Report, TR-CCTC/DI-35, GTTSE 2005, pp. 153-186.
11. Don Batory, Jacob Neal Sarvela, Axel Rauschmayer: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering, vol. 30, no. 6, pp. 355-371.
12. Don Batory, Jia Liu, Jacob Neal Sarvela: Refinements and multi-dimensional separation of concerns. Proceedings of the 9th European Software Engineering Conference, 2003.
13. Yannis Smaragdakis, Don Batory: Mixin-Based Programming in C++. In proceedings of Net.Object Days 2000 pp. 464-478.
14. Yannis Smaragdakis, Don Batory: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Transactions of Software Engineering and Methodology Vol. 11, No. 2, April 2002, pp. 215-255.
15. Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)
16. Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)
17. Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)
18. Bertrand Meyer: Eiffel: The Language. Prentice Hall (1991)
19. Lodewijk Bergmans, Mehmet Aksit: Composing Crosscutting Concerns Using Composition Filters. Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.
20. 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.

21. 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.
22. Gregor Kiczales et al.: An overview of AspectJ. LNCS 2072 (2001), pp. 327-355.
23. G. Baumgartner, M. Jansche, K. Läufer: Half & Half: Multiply Dispatch and Retroactive Abstraction for Java. Technical Report OSU-CISRC-5/01-TR08. Ohio State University, 2002.
24. 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.
25. 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
26. Harold Ossher, Peri Tarr: Hiper/J. Multidemensional Separation of Concerns for Java. International Conference on Software Engineering 2001. ACM pp. 734-737.
27. William Harrison, Harold Ossher: Subject-oriented programming: a critique of pure objects. Proceedings of 8th OOPSLA 1993, Washington D.C., USA. pp. 411-428.
28. Subject Oriented Programming. http://www.research.ibm.com/sop
29. Jonathan E. Shopiro: An Example of Multiple Inheritance in C++: a Model of the Iostream Library. ACM SIGPLAN Notices, December, 1989.