Constructors for Multiple Inheritance

Marko Martin
Darmstadt University of Technology
Karolinenplatz 5
64289 Darmstadt

m_martin@rbg.informatik.tu-darmstadt.de

ABSTRACT

Multiple inheritance is a powerful mechanism of many object-oriented programming languages. However, construction of instances for classes which are subject to multiple inheritance raises many problems. These include the order in which constructors should be invoked, the choice of initial values for object fields, and handling of the diamond problem.

In this paper, we focus on eight goals which are desirable for implementations of multiple inheritance in order to cope with those problems and compare if and how they are achieved by different solutions. Thereby, we analyze why conventional C++ does not achieve all goals and what proposals for advanced solutions exist. We give examples for the solutions and point out their strengths and weaknesses.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, inheritance, polymorphism.*

General Terms

Languages

Keywords

Multiple inheritance, constructors, diamond problem

1. Introduction

Multiple inheritance is a powerful tool available in many state-of-the-art object-oriented programming languages. It allows for combining functionality of different classes in one class in a modularized manner. However, it also introduces many semantic problems related to the instantiation of class instances which are subject to multiple inheritance.

The main purpose of object instantiation is to set the object to a valid initial state. The state of an object is usually constituted by its fields; therefore, construction of an object includes setting its fields to initial values. Sometimes, class constructors also manipulate global state such as an instance counter. As instances of a class can often be constructed in different ways, multiple constructors are necessary to express these differences.

In a single inheritance hierarchy, object initialization is easy and well understood: Each class of the hierarchy calls a certain constructor of its unambiguous superclass. This may also occur implicitly if a superclass provides a default constructor without arguments.

In contrast, a multiple inheritance hierarchy raises different questions on the constructor semantics, for example: If a class has multiple superclasses, in which order should the super-constructors be invoked? For another question, consider the diamond problem [2] which arises when a superclass Y of a class X is reachable via multiple paths in the inheritance hierarchy: Which constructor of Y should be invoked with which arguments? Remember that the different direct subclasses of Y in this hierarchy might use different constructors of Y or the same constructor, but different arguments.

We will see how these questions are answered in different solutions. We will discuss some C++-based solutions as well as the mixin solution implemented in Scala. Finally, we will have a look at a new proposed Java extension, namely CZ, and we will shortly discuss why most problems of multiple inheritance are circumvented by traits which, however, limit expressiveness.

The rest of this paper is structured as follows: Section 2 introduces the desired goals of multiple inheritance by means of two examples. Section 3 outlines some solutions and discusses them with respect to the previously introduced goals. Examples show the usefulness or the weakness of the solutions. Section 4 compares the presented solutions with respect to the goals they achieve or not and concludes on the insights we have gained in this paper. Section 5 sums up.

2. Goals of Implementations of Multiple Inheritance

This section analyzes which goals are desirable for an implementation of multiple inheritance. Different examples shall clarify these goals.

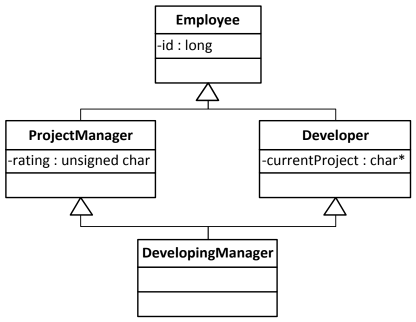

Figure 1: DevelopingManager example hierarchy

First, have a look at the example proposed in Figure 1. It is a typical example of a diamond hierarchy [2] in which a certain class (DevelopingManager) has a non-direct superclass (Employee) inherited via multiple paths (via ProjectManager and via Developer).

Obviously, it is important that a DevelopingManager has a single ID which should be specifiable during construction of a respective class instance. Therefore, we define the first goal to be achieved as

1. **Consistent field initialization:** The fields of Employee (ID) must be initialized consistently. This means, it must be possible to uniquely specify the Employee ID in the constructor of DevelopingManager, and it should not be possible for direct subclasses of Employee

(ProjectManager and Developer) to specify different Employee IDs themselves which they rely on.

Furthermore, each of the classes should have a constructor which is parameterized with values for fields to be initialized:

2. **Constructors with individual parameters:** Constructors must be able to take parameters in order to initialize fields accordingly. Parameters should be specifiable individually for each class.

If this goal is not fulfilled, it might be necessary to initialize fields of the classes manually with initialization methods which should be avoided because after constructor invocation, an object should already have a valid initial state:

3. **No separate initialization methods:** After construction of an object, it must not be necessary to call separate initialization methods in order to guarantee a valid state.

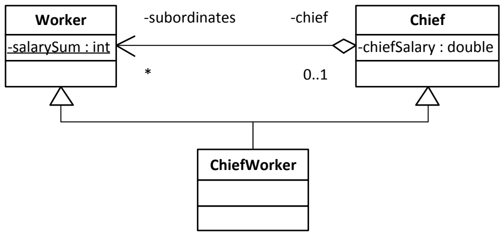

Figure 2: ChiefWorker example hierarchy

The fourth and fifth goals are introduced with the example in Figure 2: Suppose Workers to have a fixed salary. For each instantiated worker, the static field salarySum is increased by this salary. The salary of a Chief is exactly 1/10 of the current value of salarySum when the Chief object is created. A ChiefWorker will earn the sum of a normal worker salary and the chief salary calculated before the salarySum is increased. Therefore, the constructor of Chief must be invoked before the Worker constructor which is the next goal:

4. **Specifiable order of constructor invocations:** The order in which constructors of superclasses are invoked must be specifiable.

For the proposed example to be implementable, constructors must be able to have side effects additionally to their main purpose, field initialization. In this case, the Worker constructor must modify the salarySum field appropriately. Hence, the fifth goal is:

5. **Arbitrary initialization code:** Constructors must not only be able to initialize fields, but also to have other side effects by executing arbitrary code.

Finally, usability of the implementations should also be taken into account. This includes, e.g., reusability of pieces of functionality. As an example, consider the Developer piece of Figure 1 and suppose that there are not only employed developers, but also nonpaid ones. For this kind of workers, a class Volunteer exists. It should be possible to derive a subclass VolunteerDeveloper of Volunteer by inheriting from Developer as well. For that purpose, Developer must not be coupled to Employee which yields the fifth goal:

6. **Superclass decoupling:** It must be possible to define classes that rely on certain functionality without being coupled to a fixed superclass.

Superclass decoupling may, however, lead to another problem: Consider Developer to be decoupled from any superclass as described before. In this case, it is desirable that Developer would adapt to any concrete superclass so that a programmer would not have to write a dedicated new constructor for each possible superclass, e.g., one for Developer as a subclass of Employee in order to initialize the employee ID correctly, and another one for Developer as a subclass of Voluntary in order to initialize the Voluntary fields correctly. The respective goal is:

7. **Constructor reusability:** It should not be necessary to write a new constructor when a class is applied to another superclass.

Another issue of usability concerns the effort of preparation and using a certain solution of multiple inheritance:

8. **Low overhead:** The solution should not impose a significant overhead concerning syntax or memory usage.

This goal is admittedly a subjective goal; however, it shall give a hint on the practicability of solutions.

3. Implementations of Multiple Inheritance

In this section, we will see some implementations of multiple inheritance. They are presented with examples and analyzed with respect to the goals introduced in the previous section. Remarkable results concerning this analysis will be mentioned explicitly; obviously fulfilled or unachieved goals are not mentioned. For a complete overview of fulfilled goals, the reader is referred to the Section 4.

3.1 Conventional C++

When talking about multiple inheritance in C++, one has to distinguish between the two built-in variants of inheritance: *default* inheritance and *virtual* inheritance as described, for example, by Ellis and Stroustrup [4]. Both types mainly differ in the way in which diamond hierarchies are resolved: While with default inheritance, a separate instance of the common superclass for each direct subclass is created, virtual inheritance lets the common superclass be instantiated only once as a direct superclass of the common subclass in the hierarchy.

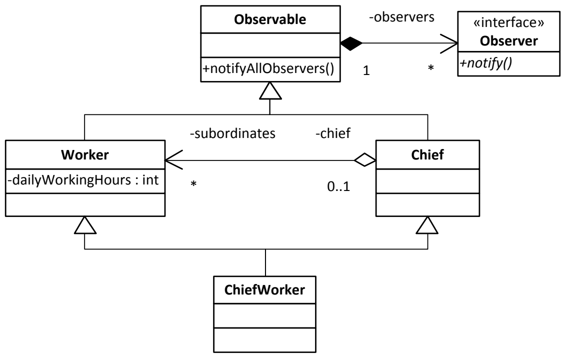

Figure 3: Extended ChiefWorker example hierarchy

As an example for an application of default inheritance, consider the hierarchy proposed in Figure 3: This extends the example of Figure 2 with the common superclass Observable in order to make changes to Worker and Chief objects observable by observers. When a remarkable change occurs, Worker as well as Chief call the `notifyAllObservers()` method of their superclass which in turn iterates over all registered observers and calls `notify()` for each of them. For the subclass ChiefWorker, it should be possible for observers to register only for events concerning one of its superclasses. E.g., an observer might be interested in changes to the list of subordinates of a chief, but they do not care about changes to the number of daily working hours which is specific for the Worker part of a ChiefWorker. This requirement demands separate observer lists and hence separate Observable instances for the Chief and the Worker part of a ChiefWorker. Therefore, C++ default inheritance fits the needs of this example.

Looking back to the example of Figure 1 again, one quickly realizes that two instances of the common superclass Employee are not feasible because a DevelopingManager should not have two different employee IDs. It is therefore a typical example where virtual inheritance must be applied as suggested in Listing 1 (appendix). Note that the constructor of DevelopingManager must call the constructor of Employee because virtual inheritance makes Employee become a virtually direct superclass of DevelopingManager. As a consequence, the invocations of the Employee constructor within the Developer and ProjectManager constructors are ignored when instantiating a DevelopingManager.

The last point contradicts the goal *consistent field initialization* because Developer and ProjectManager might actually provide different IDs to their superclass constructors which would be ignored with virtual inheritance.

The goal *defined order of constructor invocations* is fulfilled because the order in which constructors are called is defined by the order of the inheritance definition of the subclass [10]. In the example implementation in the appendix, the constructor of Developer is invoked before the one of ProjectManager because Developer is listed earlier in the list of superclasses of DevelopingManager.

With C++ virtual inheritance, it is also possible to fulfill the goal *superclass decoupling*: Methods can be declared abstract in a virtually inherited superclass A and implemented in an unknown class B. Therefore, by inheriting from A, a class can be decoupled from B. However, initialization of B is deferred to another class which actually inherits from B. Thus, the goal *constructor reusability* is not fulfilled.

In conclusion, the basic C++ solution is clear and without syntactic overhead. Also, the “virtual” keyword is easily applicable and does not impose much effort for the developer. The goal *low overhead* is therefore fulfilled. Lack of consistent field initialization and constructor reusability is the weakness of basic C++ solutions.

3.2 Mixin Solutions

Mixins are a frequently applied method for extending existing classes with existing functionality[1]. Firstly proposed by Moon [7], a mixin is an abstract snippet of functionality which is not intended to be instantiated itself, but rather to be applied to a concrete class. As a concrete class can be extended with multiple mixins, the mixin solution can be understood as a form of multiple inheritance. It is only a pseudo-form because, in fact, programmers do not apply multiple mixins simultaneously, but one after the other, thereby defining a linear order on the applied mixins.

Mixins aim at dynamic constructs in the sense that the superclass of a mixin – or more general: the functionality a mixin relies on – is a parameter of the mixin which can be substituted as necessary. Therefore, mixins are often defined as abstract subclasses [2].

[1] in contrast to subclassing which extends existing classes with *new* functionality (However, with multiple inheritance, similar effects can be simulated.)

3.2.1 Parameterized Inheritance in C++

Parameterized inheritance (PI) is a method to implement mixins in C++. It has been proposed by Smaragdakis and Batory [9] and it has been realized to modular solutions in previous projects, e.g., by VanHilst and Notkin [12]. In this solution, the mixin is defined as a subclass which is parameterized by the superclass. E.g., in order to define the Developer class of Figure 1 as a mixin, it could be defined as follows:

```
template<class A>
class Developer : public A
{
public:
     Developer(long id, char *currentProject) :
           A(id), currentProject(currentProject) {}
}
```

If ProjectManager is defined analogously, a type which is equivalent to DevelopingManager in Figure 1 could then be defined as `ProjectManager<Developer<Employee>>`.

This solution counteracts the weakness of the conventional C++ solution that the goal *consistent field initialization* was not fulfilled: Due to the linearization property of mixins, diamond hierarchy problems do not arise. Every field to be initialized is well defined in this solution and there are no constructors being ignored as in the virtual inheritance solution of C++.

However, this approach also has considerable restrictions as pointed out by Eisenecker et al. [3]. As one of them, note that the Developer mixin makes the assumption that its unknown superclass A has a constructor which takes an ID as parameter. This is necessary in order to initialize the superclass correctly (in the case that it actually needs an ID for initialization). To make things even worse, the ProjectManager constructor must, in order to be applicable to the superclass `Developer<Employee>`, pass both an ID and the currentProject parameter to its superclass’s constructor. This in turn restricts the ProjectManager mixin to application to such superclasses which actually require an ID and a currentProject parameter in their constructor. If one wanted to apply Developer and ProjectManager to other superclasses or only in reverse order, one would have to define additional constructors passing the correct parameters to the respective superclass. Hence, the goals *constructor reusability* and *low overhead* are not achieved.

Nevertheless, the goal *superclass decoupling* is fulfilled excellently because Developer is not coupled to any certain superclass – it does not even rely on any abstract class which declares the necessary functionality: It would be applicable to any class expecting an ID as constructor parameter.

Eisenecker et al. [3] sum up the following approaches which try to counteract the constructor reusability problem.

3.2.1.1 PI with Virtual Inheritance

One problem of the previous solution is that mixins have to pass all parameters of the original superclass as well as parameters of previously applied mixins to the superclass. Or referring to the DevelopingManager example again: ProjectManager does not only have to pass the ID parameter to its superclass, but also the currentProject parameter, assuming that the Developer mixin is applied before.

A countermeasure to this problem is to change the inheritance mode of mixins to virtual inheritance, e.g., for Developer:

```
template<class A>
class Developer : virtual public A { ... }
```

Now, DevelopingManager could be defined as a new class as follows:

```
class DevelopingManager : public Developer<Employee>,
          public ProjectManager<Employee> { ... }
```

In its constructor, it is responsible for passing mixin-specific constructor parameters such as currentProject for Developer and for passing the ID to the Employee constructor due to virtual inheritance of the extended mixins.

Although this method makes it possible not to care about the order in which mixins are applied (since, with virtual inheritance, the implementing subclass is responsible for initializing them all), the drawback is that an implementing subclass has to be written explicitly for every desired combination of mixins which contradicts the goal *low overhead*. Also, the solution inherits the weaknesses of virtual inheritance discussed in Section 3.1. Therefore, *consistent field initialization* is not achieved with this solution.

3.2.1.2 PI with Argument Class

Another solution to the constructor problem of the PI approach of Section 3.2.1 is to maintain an argument class which is used as parameter for every constructor of a mixin.

The idea of using an argument class was proposed by Stroustrup [11] and taken on by Eisenecker: This class is to constitute the standardized interface for constructors of all classes which may be part of a common (multiple) inheritance hierarchy. It encapsulates all data which is needed in any of these classes so that all their constructors only have to take an instance of this class in order to obtain all relevant data for sure. For the example of Figure 1, the class could look like in Figure 4.

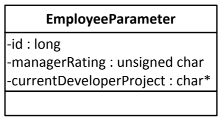
EmployeeParameter
-id : long
-managerRating : unsigned char
-currentDeveloperProject : char*

Figure 4: Argument class for the example in Figure 1

This makes it possible to define mixins which are applicable to any superclass which has a constructor taking an instance of the argument class. Each mixin constructor can then select the data it needs from the argument object and pass it through to the next superclass.

When many mixins are existing which are possibly not all used in a certain hierarchy, the size of the argument class might constitute a significant overhead in memory; thus, the goal *low overhead* is not achieved. Also, *constructors with individual parameters* for certain mixins are not possible, so the argument class must be extended for every new feature which is needed by any constructor of any mixin.

3.2.1.3 PI with Default Constructors

Yet another solution with standardized constructors bases on the convenience to provide a default constructor without parameters in every mixin. Though not exhibiting the overhead of the argument class solution, an additional drawback is that all initialization must be done explicitly with *initialization methods* after construction — an error-prone method as programmers cannot be forced to use initialization methods.

3.2.1.4 PI with Typed List Arguments

As a new and preferable solution to the constructor problem, Eisenecker et al. propose a kind of heterogeneous value lists to be used as constructor arguments. This kind of list was firstly proposed by Järli [5] and allows defining fixed-length linked lists of objects, each having a specific type.

The idea of the solution is visualized in Figure 5: In a traditional mixin hierarchy (left side of the figure), each mixin A, B, C has its specific argument list for its constructor. E.g., mixin A expects arguments of types 1, 2, and 3. The solution proposes a standardized constructor interface (similarly to the argument class solution); however, the type of the expected constructor parameter is now a heterogeneous value list. Based on C++ templates, the type of each list is generated dynamically so that it defines the number and types of expected list elements according to the expected argument types of the mixin and the inherited argument types from other mixins. For the schema in Figure 5, this means that the constructor of class C which originally needed two arguments of types 5 and 6 now expects a list of objects of types 5, 6, 4, 1, 2, 3. When the constructor of C in the new implementation is invoked, it extracts the elements of types 5 and 6 from the list for own usage and passes the rest of the linked list to the constructor of its superclass B which proceeds analogously.

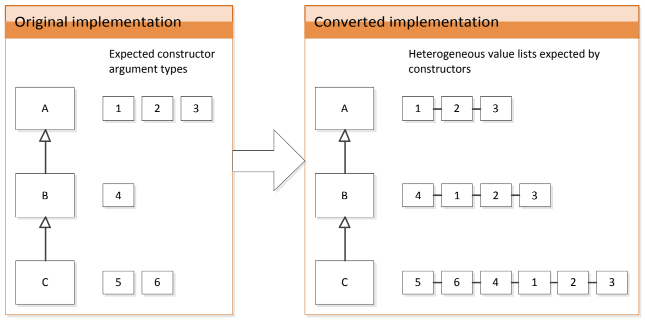
Original implementation
Expected constructor argument types
A
B
C
1
2
3
4
5
6
Converted implementation
Heterogeneous value lists expected by constructors
A
B
C
1
2
3
4
1
2
3
5
6
4
1
2
3

Figure 5: Constructors with heterogeneous value lists

As most important benefit of this solution, *constructor reusability* is now achieved for mixins because changing the superclass of a mixin will not render its constructor useless: The correct type of a heterogeneous value list is automatically generated for each possible hierarchy of mixins by the compiler which also makes the approach type-safe.

The drawback of the solution is the *overhead* for defining the mixins and the infrastructure needed to use the heterogeneous value lists. As a reward for the preparation work, mixins can be plugged together in a seamless manner.

3.2.2 Scala Traits

Compared to Java which does not allow multiple inheritance except for interfaces, Scala [1] extends possibilities of object-oriented programming with advanced inheritance technologies. The restriction to single inheritance with respect to default classes was retained, but so-called “traits” have been introduced which can be used in multiple inheritance hierarchies and which are more than a substitute for Java interfaces: As opposed to these, traits can contain both state and method implementations. A common property is that both cannot be instantiated. Thus, traits are to be understood as mixins which can be added to other classes.

The example of Figure 1 can be implemented as in Listing 2 (appendix). A main disadvantage is that traits cannot be initialized with *constructor parameters*. Hence, *initialization methods* have to be kept in mind when creating mixin-based objects.

Traits are also not responsible for initializing their superclass. Consequently, super constructor calls are impossible. If a trait extends a class without a default constructor (as in the example where Developer and ProjectManager extend Employee which takes an ID parameter), subclasses of the trait

(DevelopingManager in this case) must explicitly invoke the superclass constructor which is similar to the C++ virtual inheritance solution; however, as superclass constructor calls are forbidden, different traits may not pass different arguments to a common superclass. Hence, *consistent field initialization* is guaranteed. In contrast to virtual inheritance, *constructor reusability* is also, at least partially, achieved: A DevelopingManager could be created on-the-fly with `new Employee(42) with ProjectManager with Developer`, thereby reusing the constructor of Employee. A new constructor is only necessary if the functionalities of Employee, ProjectManager, and Developer should be combined into a new class.

The default constructor of traits can contain *arbitrary initialization code* which can just be written into the body of the trait after the opening bracket "{". The *order of trait constructor invocations* is defined by the order in which traits are mixed in. In the example listing, the order of constructors for a DevelopingManager object is: Employee, Developer, ProjectManager, DevelopingManager.

3.3 CZ

CZ is a Java-based extension proposed by Malayeri and Aldrich [6]. It allows for multiple inheritance with "extends" relationships, but forbids diamond hierarchies. It also introduces a new kind of subtyping relationship, namely "requires", which also supports diamonds. If a class A requires class B, class A can always access any visible members of its super type B. However, B is not a superclass in the "extends" sense: Class A is not responsible for creating an instance of and initializing class B; hence, no super constructor call is necessary. Yet, in order to make members of B accessible to A, the following rules for "require" relationships are employed:

- Classes requiring other classes are abstract themselves. They do not call constructors of required classes, but they can use features of the required classes as if they were subclasses.
- If a class A extends B and B requires C, A must require or extend C or a subclass of C. This ensures that B can always rely on the existence of C in a concrete implementation.

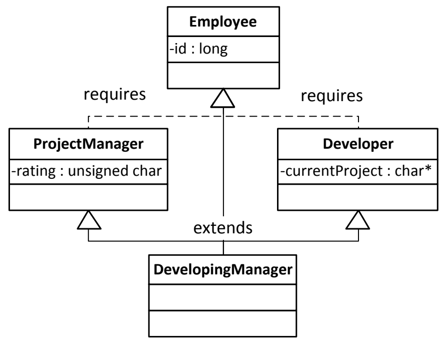

Figure 6: DevelopingManager hierarchy for CZ

The last rule, in short, denotes that "B requires C" is the contract that every concrete subclass of B is a subclass of C.

To give an example, we convert the hierarchy of Figure 1 to CZ as proposed in Figure 6. Similarly to the virtual inheritance solution, the common subclass DevelopingManager is now responsible for instantiating the common superclass Employee. As opposed to the virtual inheritance solution, DevelopingManager must explicitly extend Employee.

An important advantage of CZ is that constructor invocations of required classes are forbidden from the outset. Thus, no constructor calls must be ignored as in the case of virtual inheritance which fulfills the goal *consistent field initialization*.

Superclass decoupling is simulated by only requiring, but not extending a certain class. E.g., Developer can be extended by any direct or indirect subclass C of Employee and, due to dynamic method dispatch, behaves as if it were a subclass of C.

A weakness, compared to the solutions of parameterized inheritance, is that combining multiple classes in a new one always requires defining a new class such as DevelopingManager

Table 1: Comparison of implementations of multiple inheritance (see Section 3) with respect to the goals they achieve (see Section 2)

Solution	1) Consistent field initialization	2) Constructors with individual parameters	3) No separate initialization methods	4) Specifiable order of constructor invocations	5) Arbitrary initialization code	6) Superclass decoupling	7) Constructor reusability	8) No/few syntactic and memory overhead	Further issues
Conventional C++	–	+	+	+	+	+	–	+	(+) Choice between multiple instances and single instance of common superclass
Mixins									(+) Mixing-in functionality possible on-the-fly, i.e., without new class definition as container for mixins
C++ Parameterized Inheritance	+	+	+	+	+	++	–	–	
… with virtual inheritance	–	+	+	+	+	++	–	–	
… with argument class	+	–	+	+	+	++	++	–	
… with default constructors	+	–	–	+	+	++	+	+	
… with typed list arguments	+	+	+	+	+	++	++	–	
Scala Traits	+	–	–	+	+	+	+	+	
CZ	+	+	+	+	+	+	–	+	
Traits	+	–	+	/	–	++	–	+	(–) Traits do neither contain state nor constructors.

in the above example. This class must also explicitly invoke constructors of all extended classes, and there may be many of them if many classes requiring some other classes are extended.

3.4 Traits

Similarly to mixins, traits represent another unit of reuse which has been introduced by Shärli et al. [8]. In this context, the term "trait" is used with a different meaning than in the Scala context where traits are actually mixins. With Shärli, a trait merely defines methods which are subdivided in a group of provided methods and another group of used methods. Used methods are semantically close to abstract C++ methods and hence not implemented. Provided methods constitute the functionality which the trait provides. Furthermore, a trait does not contain state which makes constructors obsolete because traits can also not extend concrete classes, but only other traits.

Traits provide good *superclass decoupling* because traits do not depend on specific classes; they merely define the interface they rely on.

In summary, traits are useful if pure behavioral, i.e. stateless, functionality should be modularized, and they are not when state is involved in units of modularization. Therefore, the examples of Section 2 are not implementable with traits.

4. Comparison

Table 1 lists the presented solutions row-by-row and denotes in its columns whether the respective goal introduced in Section 2 is solved (+, or ++ for particularly good solutions) or not (–).

The main insights from the overview table are the following:

- Parameterized inheritance solutions suffer from high overhead, particularly, with respect to syntactical effort.
- Superclass decoupling is a desirable goal for flexible designs and hence covered by all solutions. Parameterized inheritance solutions cope best with this goal because they do not even rely on a certain interface which declares the functionality they need.
- Simple C++ solutions do not cover reusability of constructors or pay for it with restricted expressiveness, e.g., no individual constructor parameters in the case of argument classes and default constructors. However, argument classes are the best solution with respect to constructor reusability, together with heterogeneous value lists, because reusable constructors actually pass initialization values to the superclass constructors.
- The order of constructor invocations is a rather theoretical problem; all solutions cover this point well.
- Traits are merely useful for combining stateless functionality. Initialization is mainly out of their scope. They have been included into the list in order to distinguish them from other solutions, particularly Scala traits.
- There is no solution covering all discussed goals. With respect to these, the most promising solutions are C++ parameterized inheritance with heterogeneous value lists and CZ.

5. Summary

After introducing eight goals of multiple inheritance, directly or indirectly related to construction of objects, different solutions have been presented: Default C++ does not cope with diamond hierarchies in cases where a single instance of the common superclass is required; virtual inheritance turned out to impose additional pitfalls by ignoring particular constructors. Mixins model plug-in-like inheritance well, though producing problems related to initialization of mixins. Several solutions to the initialization problem have been presented including argument classes and heterogeneous value lists as constructor parameters. Scala traits have been discussed which, however, exhibited the disadvantage of forbidden constructor parameters. This problem has not appeared in the Java-based CZ solution that distinguishes between the traditional "extends" and a new "requires" relationship. For composition of stateless functionality, traits have been pointed out to be the method of choice. In a comparison, an overview of achieved goals has been provided for all presented solutions.

6. References

[1] The scala programming language. http://www.scala-lang.org/.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM.

[3] Ulrich W. Eisenecker, Frank Blinn, and Krzysztof Czarnecki. A solution to the constructor-problem of mixin-based programming in c++, 2000.

[4] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[5] Jaakko Järvi. Tuples and multiple return values in c++. Technical report, Turku Centre for Computer Science, 1999.

[6] Donna Malayeri. Cz: multiple inheritance without diamonds. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 923–924, New York, NY, USA, 2008. ACM.

[7] David A. Moon. Object-oriented programming with flavors. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 1–8, New York, NY, USA, 1986. ACM.

[8] Nathanael Shärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. Technical report, OGI School of Science & Engineering Oregon Health & Science University, 2002.

[9] Yannis Smaragdakis and Don S. Batory. Mixin-based programming in c++. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, GCSE '00, pages 163–177, London, UK, 2001. Springer-Verlag.

[10] Bjarne Stroustrup. Multiple inheritance for c++. In *Proceedings of the Spring '87 European Unix Systems Users's Group Conference*, Helsinki, May 1987.

[11] Bjarne Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[12] Michael VanHilst and David Notkin. Using role components in implement collaboration-based designs. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 359–369, New York, NY, USA, 1996. ACM.

7. APPENDIX

7.1 Listings

```
class Employee
{
public:
         Employee(long id) : id(id) {}
...
};

class Developer : virtual public Employee
{
public:
         Developer(long id, char *currentProject) :
            Employee(id), currentProject(currentProject) {}
...
};

class ProjectManager : virtual public Employee
{
public:
         ProjectManager(long id, unsigned char rating) :
            Employee(id), rating(rating)
...
};

class DevelopingManager :
    public Developer, public ProjectManager
{
public:
         DevelopingManager(long id,unsigned char rating,
                             char *currentProject) :
                  Employee(id), ProjectManager(id, rating),
                  Developer(id, currentProject) {}
};
```

Listing 1: C++ implementation of the example in Figure 1

```
class Employee(val id: Long)

trait Developer extends Employee {
      var currentProject: String = null
      protected def initialize(currentProject: String) =
              this.currentProject = currentProject
}

trait ProjectManager extends Employee {
      var rating: Short = 0
      protected def initialize(rating: Short) =
              this.rating = rating
}

class DevelopingManager(id: Long, currentProject: String,
          rating: Short) extends Employee(id)
          with Developer with ProjectManager {
      initialize(currentProject)
      initialize(rating)
}
```

Listing 2: Scala implementation of the example in Figure 1