Under consideration for publication in Formal Aspects of Computing

The diamond problem solved! Introducing abstract virtual fields as a clean and general solution to multiple inheritance

YuQian Zhou

Independent researcher

Abstract. This paper has two parts: in the first part, we discovered a semantics issue in Eiffel's field renaming mechanism when applied to the diamond problem of multiple inheritance (MI), and we propose to use *abstract virtual fields* to fix the semantics issue; in the second part, we further developed the concepts *abstract virtual fields* and *semantic branching (site)* that we introduced in the first part, and present a new design pattern DDIFI that solved the diamond problem of multiple inheritance cleanly and generally, which can be applied to a wide range of industry-strength OOP languages.

Our proposal of using *abstract virtual fields* is based on the following observation: mathematically with multiple inheritance, the class hierarchy forms an inheritance lattice, where each class node may introduce new fields. Embedding this *lattice* structure (i.e., the memory layout of all classes) into the computer's *linear* memory address space is challenging; additionally, up-casting the bottom class to any of its superclasses while maintaining memory layout integrity becomes awkward.

Our new design pattern DDIFI using *abstract virtual fields* cleanly solves the diamond problem. It can handle the instance fields of the multiple inheritance exactly according to the programmer's intended application semantics. It gives programmers flexibility when dealing with the diamond problem for instance variables: each instance variable can be configured *individually* either as one joined copy or as multiple independent copies in the implementation class. The key ideas are:

1. decouple implementation inheritance from interface inheritance by stopping inheriting data fields;
2. in the regular methods implementation use *virtual property* methods instead of direct raw fields; and
3. after each *semantic branching site* add (and override) the new semantic assigning property.

We will also show that our method is general enough, and can achieve clean multiple inheritance in a wide range of industry-strength OOP languages: e.g. C++, Python, OCaml, Lisp, Eiffel, Java, C#, Scala, D, etc.

Keywords: diamond problem, (uncoordinated) multiple inheritance, abstract virtual fields, semantic branching site, inheritance lattice, virtual property, data interface, data implementation, rename dispatching, view stack, Eiffel language, C++, Java, name clash resolution, program to interfaces, reusability, modularity

Correspondence and offprint requests to: YuQian Zhou

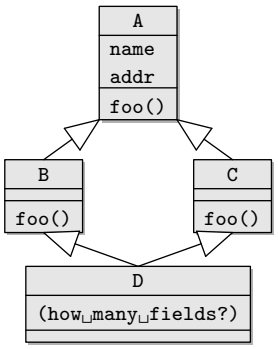
Part 1: Introducing abstract virtual fields and rename dispatching based on type view stack to fix Eiffels field renaming semantics issue

Among all the object-oriented programming (OOP) languages that support multiple inheritance (MI), Eiffel is unique for its distinctive renaming mechanism, which is designed to address name clashes of each class member individually in the derived classes. However, we discovered a semantics issue in Eiffel's field renaming mechanism when applied to the diamond problem of MI, and confirmed it by showing divergent and problematic outputs for the same example code in three major different Eiffel compilers. In particular, ISE EiffelStudio compiler demonstrates *different* field renaming semantics even between its own workbench (i.e debug) mode and finalize (i.e. release) mode, while *both* are problematic. To fix the semantics issue we propose to abandon the renaming's reference identity semantics; we introduce a concept called *abstract virtual fields*, and propose two methods: the first method is manual fix with help from enhanced programming rules, e.g. direct virtual field access is only allowed in accessor methods — among other rules (in the second part of this paper, we will generalize the concept of *abstract virtual fields* as a clean and general solution to the diamond problem in a wide range of industry-strength OOP languages); and the second method is automatic. We introduce rename dispatching based on the object's runtime types' *view stack*, and provide formalized new programming rules. Hence providing an improved solution to multiple inheritance. We also have developed a tool to detect such semantics issue in current Eiffel code, and have given it to the Eiffel user community. At the end of this part we report our detection findings (and bugs) in the real ISE Eiffel source code, and suggest compiler migration plans.
All the source code of this paper can be found in the supplementary material: eiffel_rename.tgz and gobo-detect_diamond.zip.

1. Motivation: the diamond problem

The most well known problem in multiple inheritance (MI) is the diamond problem [Sny87] [Knu88] [Sak89], for example on the popular website wikipedia[1] (for everyday working programmers) it is described as:

> The "diamond problem" is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?



This Wikipedia definition of the diamond problem pertains to methods, actually in real-world engineering practice, for any *method*'s ambiguity e.g. foo(), it is *relatively easy* to resolve *by the programmers* (since there is no *auto-magical* way for the compiler to *guess* which method has the correct *application semantics* that the programmers have in their *mind*; in Sec 12.3 we will discuss this in more detail and give an example):

- just *override* it in D.foo(), or
- explicitly use fully quantified method names, e.g. A.foo(), B.foo(), or C.foo().

The *more difficult* problem is how to handle the *data members (i.e. fields)* inherited from A:

[1] `https://en.wikipedia.org/wiki/Multiple\_inheritance\#The\_diamond_problem`

1. Shall D have one joined copy of A's fields? or
2. Shall D have two separate copies of A's fields? or
3. Shall D have mixed fields from A, with some fields being joined, and others separated?

For instance in C++ [Str89a] [Str94] [Str03] (1) is called virtual inheritance, and (2) is default (regular) inheritance. But C++ does not completely solve this problem, it is difficult to achieve (3). This *diamond problem on fields* will be the main focus of this paper.
For example let's build an object model for PERSON, STUDENT, FACULTY, and RESEARCH_ASSISTANT in a university:

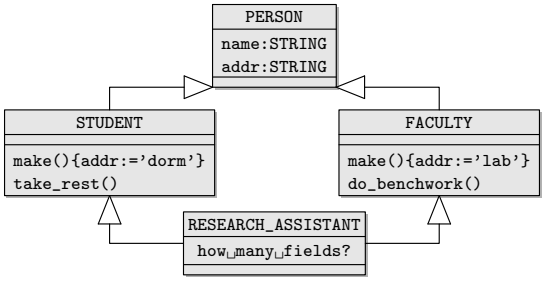


Fig. 1: the diamond problem in multiple inheritance

The intended semantics is that a RESEARCH_ASSISTANT should have only 1 name field, but 2 address fields: one "dorm" as STUDENT to take_rest(), and one "lab" as FACULTY to do_benchwork(); so in total 3 fields. However, in C++'s plain MI we can either do:

1. virtual inheritance: RESEARCH_ASSISTANT will have 1 `name`, and 1 `addr`; or
2. default inheritance: RESEARCH_ASSISTANT will have 2 `name`s, and 2 `addr`s.

Hence with C++'s plain MI mechanism, RESEARCH_ASSISTANT will have either one whole copy (2 fields in total), or two whole copies (4 fields in total) of PERSON's all data members. This leaves something better to be desired.
Among all the OOP languages that support MI, Eiffel is unique in that it provides a renaming mechanism designed to resolve name clashes of each class member *individually* in the derived classes. Let's check how Eiffel models this example in the next Section 2, then we will examine the semantics of Eiffel's renaming mechanism in Section 2.2, and show the semantics issue we discovered in 2.3. In Section 3 we will fix the semantics issue with a manual approach, by introducing the concept of abstract virtual fields and proposing some enhanced programming rules. In Section 4 we will introduce rename dispatching based on view stack to fix the semantics issue automatically with an experimental compiler, and describe our implementation in Section 5. In Section 6 we will discuss a detection tool that we have developed, which is the best thing we can do for now to help the Eiffel programmers with existing code to alert and avoid fields renaming in diamond inheritance before the renaming semantics is fixed in the Eiffel language specification. In Section 7 we will discuss Eiffel's renamed methods, and compare virtual method dispatch and rename dispatching. In Section 8 we will discuss some related works. Finally, we summarize our work in Section 9.

2. Eiffel's renaming mechanism[2]

We want to simulate *uncoordinated* MI, which means when a programmer works on class C, s/he cannot make any changes to any of C's superclasses. So the programming language has to provide good feature[3] adaptation mechanisms to allow the programmer of class C to achieve the intended semantics. Let us assume each class is developed by different software vendors independently – hence uncoordinated, but in the topological order of inheritance directed acyclic graph (*DAG* for MI; while for single inheritance, it's inheritance *tree*), starting from the top base classes. And if class B inherits from A, we say B('s level) is *below* A.

[2] Note: all the code listings in this section are compilable and executable, so it's a bit verbose.
[3] In Eiffel, a feature means a field or a method.

The first vendor developed class PERSON, each person has a name and an address:

Listing 1: person.e (Eiffel)

```
class PERSON
inherit ANY redefine default_create end  -- needed by ISE and GOBO compiler; but not by SmartEiffel

create {ANY}
   default_create

feature {ANY}
   name: STRING
   addr: STRING

   get_addr():STRING  is do Result := addr end  -- accessor method, to read
   set_addr(a:STRING) is do addr   := a    end  -- accessor method, to write

   default_create is  -- the constructor
     do
       name := "name"
       addr := "addr"
     end
end
```

The second vendor developed class STUDENT: added set/get_student_addr() accessors, and take_rest() method, since PERSON has the `addr` field already, the second vendor just uses it instead of adding another field:

Listing 2: student.e (Eiffel)

```
class STUDENT
inherit PERSON

create {ANY}
   default_create


feature {ANY}
   get_student_addr():STRING  is do Result := get_addr() end  -- assign dorm semantics to addr
   set_student_addr(a:STRING) is do set_addr(a) end

   take_rest() is
     do
       io.put_string(name + " take_rest in the: " + get_student_addr() + "%N");
     end
end
```

At the *same time* as the second vendor, the third vendor developed class FACULTY: added set/get_faculty_addr() accessors, and do_benchwork() method, in the same way to reuse the inherited field `PERSON.addr` instead of adding another field:

Listing 3: faculty.e

```
class FACULTY
inherit PERSON

create {ANY}
   default_create


feature {ANY}
   get_faculty_addr():STRING  is do Result := get_addr() end  -- assign lab semantics to addr
   set_faculty_addr(a:STRING) is do set_addr(a) end

   do_benchwork() is
     do
       io.put_string(name + " do_benchwork in the: " + get_faculty_addr() + "%N");
     end
end
```

Definition 1 (semantic branching site of field) *At this point, we can see the two different inheritance branches of PERSON have assigned different semantics to the same inherited field* `addr`*, we call class PERSON as the* semantic branching site of field `addr`.

By contrast, in the whole inheritance DAG of this example, there is no semantic branching site of field `name`.

2.1. Eiffel MI: individual feature renaming

With Eiffel's renaming mechanism to treat each inherited feature (field or method) individually from the super-classes, we implement RESEARCH_ASSISTANT as listing 4.

Listing 4: research_assistant.e

```
1  class RESEARCH_ASSISTANT
2  inherit
3          STUDENT rename addr as student_addr end  -- field student_addr inherit the dorm semantics
4          FACULTY rename addr as faculty_addr end  -- field faculty_addr inherit the lab  semantics
5          -- then select, NOTE: not needed by SmartEiffel, but needed by GOBO and ISE compiler
6          PERSON  select addr end
7
8  create {ANY}
9     make
10
11 feature {ANY}
12    print_ra() is  -- print out all 3 addresses
13       do
14         io.put_string(name +" has 3 addresses: <"+ addr +", "+ student_addr +", "+ faculty_addr + ">%N")
15       end
16
17    make is  -- the constructor
18       do
19         name := "ResAssis"
20         addr := "home"          -- the home semantics
21         student_addr := "dorm"  -- the dorm semantics
22         faculty_addr := "lab"   -- the lab  semantics
23       end
24 end
```

Definition 2 (renaming site of a field) *If a class A has renamed any of its super-class(es)' field, we call A the* renaming site *of the field.*

For example, RESEARCH_ASSISTANT is the renaming site for both STUDENT.addr and FACULTY.addr.

Please note, we have made RESEARCH_ASSISTANT inherited from PERSON 3 times: 2 times indirectly via STUDENT and FACULTY, and 1 time directly from PERSON. Thus, RESEARCH_ASSISTANT has 1 `name` field (inherited fields with the same name are joined in Eiffel by default), and 3 address fields. The extra inheritance from PERSON is to make the inheritance from STUDENT and FACULTY symmetric, which helps easy exposition of the next Section 4 when we discuss view stacks.

The Eiffel `select`[4] clause allows the programmer to *explicitly* resolve name clash on each inherited feature *individually*, which we believe is a better solution than imposing the *same* method resolution order (MRO) [BCH+96] to *all* features as in many other OOP languages, e.g. Python [vR10]: the base classes' order in the inheritance clause should *not* matter.

[4] Note: strictly speaking the `select addr end` clause on line 6 is not needed, because after the two renamings on line 3 and 4, there is no more name clash on RESEARCH_ASSISTANT's field `addr`, hence no ambiguity. However some Eiffel compilers (e.g. GOBO and ISE) enforce the presence of this `select` clause which we think is wrong, while others (e.g. SmartEiffel) do not.

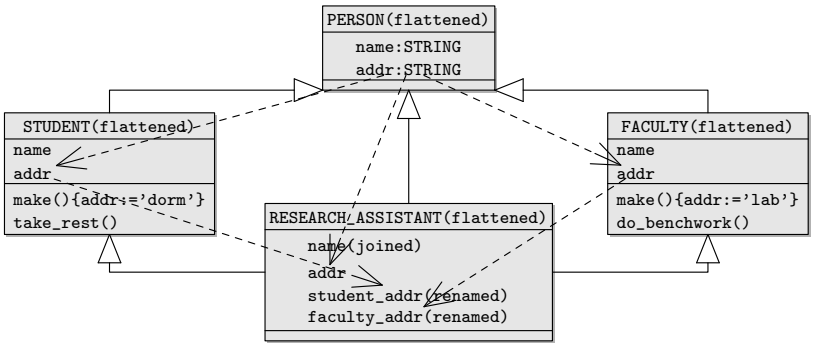


Fig. 2: flattened fields view, and feature renaming DAG of field 'addr' (dashed arrows)

The above diagram shows the field flattened view of the classes implemented in Eiffel; in particular, the dashed arrows show how the field `addr` is inherited (and renamed) in the class hierarchy, we call it *feature renaming DAG* of `addr`.

2.2. Examine the semantics of the renamed fields

We want to study how the renamed field behaves in this diamond inheritance. In ECMA-367 [ECM06][5] Section 8.16.2, it says:

Let D be a class and $B_1, B_n (n \geq 2)$ be parents of D based on classes having a common ancestor A. ... Any two of these features inherited under a different name yield two features of D.

Which means the purpose of the renaming mechanism is to have *separate* copies for the name-clashed field. On the eiffel.com tutorial website, we found the following "Repeated Inheritance rule" [6]:

- A feature inherited multiply under one name will be shared: it is considered to be just one feature in the repeated descendant.
- A feature inherited multiply under different names will be replicated, yielding as many variants as names.

So without renaming, all fields with the same name from the base classes are joined. However for fields that need to be separated, joining will cause semantic error. E.g. `STUDENT.addr` v.s. `FACULTY.addr` in `RESEARCH_ASSISTANT`:

- for the inherited methods from `STUDENT`, e.g. `STUDENT.take_rest()`, they expect `addr` to have "dorm" semantics; while
- for the inherited methods from `FACULTY`, e.g. `FACULTY.do_benchwork()`, they expect `addr` to have "lab" semantics.

But if these two fields with different semantics are joined under the same name in `RESEARCH_ASSISTANT`, they will share the same data member `addr`, it will cause disastrous bugs in the resulting program.
Then in ECMA-367 Section 8.6.16, we find a very brief description of the renaming's semantics:

Renaming principle: Renaming does not affect the semantics of an inherited feature.

so we have to look further elsewhere: from [Mey97] Section 15.2 page 544-545, we found a concrete example:

```
class SANTA_BARBARA inherit
  LONDON   rename foo as fog end
  NEW_YORK rename foo as zoo end
feature
  ...
end
```

[5] It serves as Eiffel language standard specification.
[6] `https://archive.eiffel.com/doc/online/eiffel50/intro/language/tutorial-10.html`

...
`l: LONDON; n: NEW_YORK; s: SANTA_BARBARA`
Then l.foo and s.fog are both valid; after a polymorphic assignment l := s they would have the same effect, since the feature names represent the same feature. Similarly n.foo and s.zoo are both valid, and after n := s they would have the same effect.
None of the following, however, is valid:

- l.zoo, l.fog, n.zoo, n.fog since neither LONDON nor NEW_YORK has a feature called fog or zoo.
- s.foo since as a result of the renaming SANTA_BARBARA has no feature called foo.[7]

From these descriptions, we can conclude in Eiffel renaming is a *reference identity* relation between the original field and the new renamed field, let's use `=` to denote this relationship (In Eiffel syntax, = is used to test reference identity).
However, we think there are two problems here: suppose both `LONDON` and `NEW_YORK` actually inherited `foo` from their common super-class `CITY` (i.e. form a diamond inheritance relationship), and `c: CITY`, then after `c := s` assignment:

1. *No feature separation achieved*. Because
   - `c.foo = l.foo = s.fog`, and
   - `c.foo = n.foo = s.zoo`

   then it follows: `s.fog = s.zoo`, which means there is *no feature separation (i.e. two distinct features) achieved* at all! We think this simple reference identity semantics of field renaming is a semantics issue in Eiffel, as demonstrated in this diamond problem.
2. *No needed feature renaming dispatch found*. Another problem is: suppose there is a method `m()` in `CITY` that accesses feature `foo` in `CITY`, then when `m()` is invoked on a `SANTA_BARBARA` object, i.e `c.m()`, which `s.fog` or `s.zoo` will be accessed? We have not find any discussion of this needed feature renaming dispatch (i.e. from `c.foo` to either `s.fog` or `s.zoo`) in Eiffel literature.

To confirm our suspicion, we will verify it with the actual Eiffel compilers.

2.3. Verify the semantics issue with real Eiffel compilers

Let's create a RESEARCH_ASSISTANT object, and call the method do_benchwork() and take_rest() on it, and check the outputs:

Listing 5: app.e

```
-- to build with SmartEiffel: compile  app.e -o app
class APP inherit INTERNAL

create {ANY}
   make

feature {ANY}
   ra: RESEARCH_ASSISTANT
   p:  PERSON
   s:  STUDENT
   f:  FACULTY

   -- problematic implementation: direct field access
   print_student_addr_direct_field(u: STUDENT) is
      do io.put_string(u.name + " as STUDENT.addr: " + u.addr + "%N") end
   print_faculty_addr_direct_field(u: FACULTY) is
      do io.put_string(u.name + " as FACULTY.addr: " + u.addr + "%N") end

   -- correct implementation: use semantic assigning accessor
   print_student_addr_via_accessor(u: STUDENT) is
      do io.put_string(u.name + " as STUDENT.addr: " + u.get_student_addr() + "%N") end
   print_faculty_addr_via_accessor(u: FACULTY) is
      do io.put_string(u.name + " as FACULTY.addr: " + u.get_faculty_addr() + "%N") end
```

[7] This example actually also demonstrates why the `select addr end` clause in class RESEARCH_ASSISTANT is not necessary: after two renamings, there is no more name clash on `addr`.

```
   make is
      do
         create  p.default_create
         create  s.default_create
         create  f.default_create
         create ra.make

         ra.print_ra()
         io.put_string("PERSON  size: " +physical_size(p ).out+ "%N")
         io.put_string("STUDENT size: " +physical_size(s ).out+ "%N")
         io.put_string("FACULTY size: " +physical_size(f ).out+ "%N")
         io.put_string("RESEARCH_ASSISTANT size: " +physical_size(ra).out+ "%N")

         ra.do_benchwork()  -- which addr field will this call access?
         ra.take_rest()     -- which addr field will this call access?

         io.put_string("-- print_student|faculty_addr_direct_field%N")
         print_student_addr_direct_field(ra)
         print_faculty_addr_direct_field(ra)

         io.put_string("-- print_student|faculty_addr_via_accessor%N")
         print_student_addr_via_accessor(ra)
         print_faculty_addr_via_accessor(ra)

         io.put_string("-- check reference identity%N")
         if                  ra.addr  = ra.faculty_addr
         then io.put_string("ra.addr  = ra.faculty_addr%N")
         else io.put_string("ra.addr != ra.faculty_addr%N") end

         if                  ra.addr  = ra.student_addr
         then io.put_string("ra.addr  = ra.student_addr%N")
         else io.put_string("ra.addr != ra.student_addr%N") end

         if                  ra.student_addr  = ra.faculty_addr
         then io.put_string("ra.student_addr  = ra.faculty_addr%N")
         else io.put_string("ra.student_addr != ra.faculty_addr%N") end

         io.put_string("-- test: suppose ra moved both lab2 and dorm2%N")
         ra.set_faculty_addr("lab2")
         ra.print_ra()
         ra.set_student_addr("dorm2")
         ra.print_ra()
      end
end
```

We have tested all the three major Eiffel compilers that we can find on the internet:

1. the latest ISE EiffelStudio[8] 23.09.10.7341 (released in 2023)
2. the open source Gobo Eiffel compiler gec version 22.01.09.4[9] (released in 2022)
3. the open source GNU SmartEiffel version 1.1[10] (released in 2003)

As will be shown in the following subsections, all three compilers generate problematic outputs.

2.3.1. ISE compiler

First, let us test ISE EiffelStudio compiler in workbench (i.e. debug) mode:

Listing 6: ISE EiffelStudio workbench (i.e. debug) mode output: most of the lines are wrong

```
ResAssis has 3 addresses: <home, home, home>  -- all 3 addr is "home"! no feature separation at all!
PERSON  size: 32
STUDENT size: 32
FACULTY size: 32
RESEARCH_ASSISTANT size: 48
```

[8] https://www.eiffel.com/company/leadership/ the company with Eiffel language designer Dr. Bertrand Meyer as CEO and Chief Architect, Founder.

[9] https://github.com/gobo-eiffel/gobo

[10] in later years, SmartEiffel was divergent from the ECMA Eiffel standard, so we choose to test version 1.1 (the last version of 1.x).

```
6  ResAssis do_benchwork in the: home
7  ResAssis take_rest in the: home
8  -- print_student|faculty_addr_direct_field
9  ResAssis as STUDENT.addr: home
10 ResAssis as FACULTY.addr: home
11 -- print_student|faculty_addr_via_accessor
12 ResAssis as STUDENT.addr: home
13 ResAssis as FACULTY.addr: home
14 -- check reference identity
15 ra.addr  = ra.faculty_addr
16 ra.addr  = ra.student_addr
17 ra.student_addr  = ra.faculty_addr
18 -- test: suppose ra moved both lab2 and dorm2
19 ResAssis has 3 addresses: <lab2, lab2, lab2>
20 ResAssis has 3 addresses: <dorm2, dorm2, dorm2>
```

From ISE workbench (i.e. debug) mode output, we can see our suspicion is confirmed: it demonstrates a few problems:

1. line 2 – 5 show the object size in bytes, we can see indeed `RESEARCH_ASSISTANT` is bigger than both `STUDENT` and `FACULTY`, which means it has more data fields; while line 15 – 17 (check reference identity) [11], report `ra.addr = ra.faculty_addr` and `ra.addr = ra.student_addr`, which means the `rename` language construct does not help to achieve feature separation at all, this is a language semantics issue.
2. Moreover, the 1st line of the ISE compiler output is three "home" strings! if we check the constructor `RESEARCH_ASSISTANT.make()` in listing 4, we can see the last assignment statement is `faculty_addr := "lab"`, even with reference identity semantics this output is wrong, so we think this is an ISE compiler bug (also compare it with the line 19 – 20).
3. line 6 & 7, do_benchwork() and take_rest() all output "home", it failed to fulfill the programmer's intention.
4. again, the last two lines 19 – 20, although it achieved the renaming's reference identity semantics, it does not achieve feature separation.

Then, let us test ISE EiffelStudio compiler in finalized (i.e. release) mode:

Listing 7: ISE EiffelStudio finalized (i.e. release) mode output: most of the lines are wrong

```
1  ResAssis has 3 addresses: <home, dorm, lab>
2  PERSON  size: 32
3  STUDENT size: 32
4  FACULTY size: 32
5  RESEARCH_ASSISTANT size: 48
6  ResAssis do_benchwork in the: home
7  ResAssis take_rest in the: home
8  -- print_student|faculty_addr_direct_field
9  ResAssis as STUDENT.addr: home
10 ResAssis as FACULTY.addr: home
11 -- print_student|faculty_addr_via_accessor
12 ResAssis as STUDENT.addr: home
13 ResAssis as FACULTY.addr: home
14 -- check reference identity
15 ra.addr != ra.faculty_addr
16 ra.addr != ra.student_addr
17 ra.student_addr != ra.faculty_addr
18 -- test: suppose ra moved both lab2 and dorm2
19 ResAssis has 3 addresses: <lab2, dorm, lab>
20 ResAssis has 3 addresses: <dorm2, dorm, lab>
```

We noticed the finalized mode output is *different* from workbench mode output:

1. line 1, and 15 – 17 (check reference identity) show in finalized mode it achieved feature separation, however
2. line 19 – 20, for assignment:
   - `ra.set_faculty_addr("lab2")`, and
   - `ra.set_student_addr("dorm2")`

[11] In Eiffel, "=" means reference identity testing

it wrongly changed the value of `RESEARCH_ASSISTANT.addr`, while the very reason the programmer introduced renaming is want to

- modify `RESEARCH_ASSISTANT.student_addr` on the `ra` object, and
- modify `RESEARCH_ASSISTANT.faculty_addr` on the `ra` object

2.3.2. GOBO compiler and SmartEiffel compiler

Listing 8: GOBO output

```
ResAssis has 3 addresses: <home, dorm, lab>
PERSON  size: 24
STUDENT size: 24
FACULTY size: 24
RESEARCH_ASSISTANT size: 40
ResAssis do_benchwork in the: home
ResAssis take_rest in the: home
-- print_student|faculty_addr_direct_field
ResAssis as STUDENT.addr: home  -- wrong read
ResAssis as FACULTY.addr: home  -- wrong read
-- print_student|faculty_addr_via_accessor
ResAssis as STUDENT.addr: home
ResAssis as FACULTY.addr: home
-- check reference identity
ra.addr != ra.faculty_addr
ra.addr != ra.student_addr
ra.student_addr != ra.faculty_addr
-- test: suppose ra moved both lab2 and dorm2
ResAssis has 3 addresses: <lab2, dorm, lab>   -- wrong
ResAssis has 3 addresses: <dorm2, dorm, lab>  -- wrong
```

Listing 9: SmartEiffel output

```
1  ResAssis has 3 addresses: <home, dorm, lab>
2  PERSON  size: 12
3  STUDENT size: 12
4  FACULTY size: 12
5  RESEARCH_ASSISTANT size: 20
6  ResAssis do_benchwork in the: home
7  ResAssis take_rest in the: home
8  -- print_student|faculty_addr_direct_field
9  ResAssis as STUDENT.addr: home
10 ResAssis as FACULTY.addr: home
11 -- print_student|faculty_addr_via_accessor
12 ResAssis as STUDENT.addr: home
13 ResAssis as FACULTY.addr: home
14 -- check reference identity
15 ra.addr != ra.faculty_addr
16 ra.addr != ra.student_addr
17 ra.student_addr != ra.faculty_addr
18 -- test: suppose ra moved both lab2 and dorm2
19 ResAssis has 3 addresses: <lab2, dorm, lab>
20 ResAssis has 3 addresses: <dorm2, dorm, lab>
```

From line 1 & 14 – 17, we can see GOBO indeed separate the 3 address fields (note: this output shows it does *not* implement the reference identity semantics), but it failed to achieve the programmer's intended semantics:

1. line 6 & 7, do_benchwork() and take_rest() all output "home", same problem as ISE
2. line 19 – 20, for assignment:

   - `ra.set_faculty_addr("lab2")`, and
   - `ra.set_student_addr("dorm2")`

   it wrongly changed the value of `RESEARCH_ASSISTANT.addr`, while the very reason the programmer introduced renaming is want to

   - modify `RESEARCH_ASSISTANT.student_addr` on the `ra` object, and
   - modify `RESEARCH_ASSISTANT.faculty_addr` on the `ra` object

SmartEiffel's output is mostly the same as GOBO output (except the object size which is compiler dependent), and both compilers do *not* implement the reference identity semantics.
In particular, we can see for all the three compilers: the do_benchwork() method calls all print out the *problematic* address "home", while the programmer's intention is "lab"; and take_rest() print out "home" instead of "dorm".
Please also note: currently FACULTY.do_benchwork() calls FACULTY.get_faculty_addr(), and then PERSON.get_addr() to access the field `addr`; even if we change FACULTY.do_benchwork() or FACULTY.get_faculty_addr() to access the field `addr` directly, the output is still the same, which we have tested; and interested readers are welcome to verify it.
We choose to define these three pairs of seemingly redundant accessor methods:

- `PERSON.set/get_addr()`
- `STUDENT.set/get_student_addr()`

- `FACULTY.set/get_faculty_addr()`

for the purpose of easy exposition of the next Section 3. In the next two sections, we will fix the semantics issue we have found with two different methods.

3. Abstract virtual fields and enhanced accessor rules

Our first method to fix the Eiffel rename semantic issue is we will add enhanced accessor rules of the renamed fields to the compiler, and help the programmer to access these fields with discipline.

3.1. Memory layout

Let us examine the flattened view of the fields of each class. Since most Eiffel compilers generate C code as target, let's use C to make our discussion more concrete.

The intended memory layout of each flattened class with multiple inheritance and renaming is shown on the right:

```c
struct Person {
  char* name;
  char* addr;
};
struct Student {
  char* name;  // from Person
  char* addr;  // from Person
};
struct Faculty {
  char* name;  // from Person
  char* addr;  // from Person
};
struct ResearchAssistant {
  char* name;  // from Person, Student&Faculty (joined)
  char* addr;  // from Person, no renaming
  char* student_addr;  // from Student and renaming
  char* faculty_addr;  // from Faculty and renaming
};
```

Rule 1 (remove reference identity) *To fix the semantics issue, first we remove the reference identity relationship among all the renamed fields.*

For example, all the three `*addr` fields in the above example.
When a RESEARCH_ASSISTANT is passed as a PERSON object to a method call or assignment target, and then need to access its `addr` field, depending on its execution context (which we will explain later), this *field access* need to be dispatched to one of:

- `ResearchAssistant.addr`
- `ResearchAssistant.student_addr`
- `ResearchAssistant.faculty_addr`

3.2. Abstract virtual fields and accessing rules

In traditional OOP languages, we have virtual method dispatch depends on the actual object type, but here the *field access* also needs to be dispatched to the intended renamed new field. Therefore, we would like to introduce the following concept:

Definition 3 (abstract virtual fields) *If a class field is renamed (anywhere) in the inheritance DAG, we call it a* virtual field.

For *uncoordinated* MI, i.e. the programmer can inherit any existing class, and make any feature (here field) adaptations to create a new class, so any field in any class can be a virtual field. To fix the semantics issue, we introduce the following enhanced compiler rules:

Rule 2 (abstract virtual fields accessing rule)

1. *the programmer must* add new semantic assigning *accessor methods for every virtual field in each class that is immediately below the field's semantic branching site.*
2. *at each renaming site of a field, the programmer must* override *the new virtual semantic assigning accessor method of that field added in (1), to use the field with the new name.*

3. *only accessor methods can make direct access (read or write) to those actual fields; while any other methods must use these semantic accessor methods to access those actual fields, instead of accessing those actual fields directly.*

For examples:

- by Rule 2.(1): PERSON is the semantic branching site of field `addr`, so
  - STUDENT must add new accessors get / set_student_addr()
  - FACULTY must add new accessors get / set_faculty_addr()
- by Rule 2.(2): RESEARCH_ASSISTANT is the renaming site, so it must override
  - STUDENT.get / set_student_addr() to read / write the renamed field `student_addr`
  - FACULTY.get / set_faculty_addr() to read / write the renamed field `faculty_addr`

Adding new semantic assigning accessors is very important: e.g.

- FACULTY.get_addr() / set_addr() (via PERSON) v.s.
- FACULTY.get_faculty_addr() / set_faculty_addr()

if we only override get_addr() / set_addr() in RESEARCH_ASSISTANT, it will affect both FACULTY and STUDENT class' methods that calls get_addr() / set_addr(), hence still mix the two different semantics; while adding and overriding get_faculty_addr() can establish the semantics of the renamed field FACULTY.addr → RESEARCH_ASSISTANT.faculty_addr.
Furthermore for any other method defined in FACULTY that need the `faculty_addr` semantics of field `addr` (which was only renamed and available in class RESEARCH_ASSISTANT level and down below), it needs to call these new accessor methods instead of the FACULTY.get_addr() / set_addr().
Now let's update class RESEARCH_ASSISTANT to comply with these rules:

Listing 10: research_assistant.e with virtual accessor method override

```
class RESEARCH_ASSISTANT
inherit
        STUDENT rename addr as student_addr      -- field student_addr inherit the dorm semantics
                redefine get_student_addr, set_student_addr
        end
        FACULTY rename addr as faculty_addr      -- field faculty_addr inherit the lab  semantics
                redefine get_faculty_addr, set_faculty_addr
        end
        -- then select, NOTE: not need by SmartEiffel, but needed by GOBO and ISE compiler
        PERSON  select addr end

create {ANY}
   make

feature {ANY}
   get_student_addr():STRING is  do Result := student_addr end  -- override and read the renamed field!
   get_faculty_addr():STRING is  do Result := faculty_addr end  -- override and read the renamed field!
   set_student_addr(a:STRING) is do student_addr := a end   -- override and write to the renamed field!
   set_faculty_addr(a:STRING) is do faculty_addr := a end   -- override and write to the renamed field!

   print_ra() is  -- print out all 3 addresses
     do
       io.put_string(name +" has 3 addresses: <"+ addr +", "+ student_addr +", "+ faculty_addr + ">%N")
     end

   make is
     do
       name := "ResAssis"
       addr := "home"            -- the home semantics
       set_student_addr("dorm")  -- the dorm semantics
       set_faculty_addr("lab")   -- the lab  semantics
     end
end
```

Please pay special attention to the redefined (override) methods get_student_addr(), get_faculty_addr(),

set_student_addr() and set_faculty_addr() to see how they implemented the renamed field's *intended* accessor semantics. With these manual overrides, let's run the updated program, and check the new results: As we noted in Listing 6, the ISE compiler in workbench mode implemented the problematic reference identity semantics of the renamed fields, *since we cannot change their compiler, our method is not applicable to the ISE compiler* in workbench mode. However, we can apply it in finalized mode:

Listing 11: ISE finalized mode: most problems fixed

```
1  ResAssis has 3 addresses: <home, dorm, lab>
2  PERSON  size: 32
3  STUDENT size: 32
4  FACULTY size: 32
5  RESEARCH_ASSISTANT size: 48
6  ResAssis do_benchwork in the: lab
7  ResAssis take_rest in the: dorm
8  -- print_student|faculty_addr_direct_field
9  ResAssis as STUDENT.addr: home
10 ResAssis as FACULTY.addr: home
11 -- print_student|faculty_addr_via_accessor
12 ResAssis as STUDENT.addr: dorm
13 ResAssis as FACULTY.addr: lab
14 -- check reference identity
15 ra.addr != ra.faculty_addr
16 ra.addr != ra.student_addr
17 ra.student_addr != ra.faculty_addr
18 -- test some assignment: suppose ra moved both lab2 and dorm2
19 ResAssis has 3 addresses: <home, dorm, lab2>
20 ResAssis has 3 addresses: <home, dorm2, lab2>
```

We can see most of the problems are fixed, except the two lines 9 & 10 in the middle where the programmer made direct field access (which violates the new programming rules);
Both GOBO and SmartEiffel do not use this reference identity semantics, so we can check their new outputs.

Listing 12: GOBO: most problems fixed

```
ResAssis has 3 addresses: <home, dorm, lab>
PERSON  size: 24
STUDENT size: 24
FACULTY size: 24
RESEARCH_ASSISTANT size: 40
ResAssis do_benchwork in the: lab
ResAssis take_rest in the: dorm
-- print_student|faculty_addr_direct_field
ResAssis as STUDENT.addr: home
ResAssis as FACULTY.addr: home
-- print_student|faculty_addr_via_accessor
ResAssis as STUDENT.addr: dorm
ResAssis as FACULTY.addr: lab
-- check reference identity
ra.addr != ra.faculty_addr
ra.addr != ra.student_addr
ra.student_addr != ra.faculty_addr
-- test: suppose ra moved both lab2 and dorm2
ResAssis has 3 addresses: <home, dorm, lab2>
ResAssis has 3 addresses: <home, dorm2, lab2>
```

Listing 13: SmartEiffel: most problems fixed

```
1  ResAssis has 3 addresses: <home, dorm, lab>
2  PERSON  size: 12
3  STUDENT size: 12
4  FACULTY size: 12
5  RESEARCH_ASSISTANT size: 20
6  ResAssis do_benchwork in the: lab
7  ResAssis take_rest in the: dorm
8  -- print_student|faculty_addr_direct_field
9  ResAssis as STUDENT.addr: home
10 ResAssis as FACULTY.addr: home
11 -- print_student|faculty_addr_via_accessor
12 ResAssis as STUDENT.addr: dorm
13 ResAssis as FACULTY.addr: lab
14 -- check reference identity
15 ra.addr != ra.faculty_addr
16 ra.addr != ra.student_addr
17 ra.student_addr != ra.faculty_addr
18 -- test: suppose ra moved both lab2 and dorm2
19 ResAssis has 3 addresses: <home, dorm, lab2>
20 ResAssis has 3 addresses: <home, dorm2, lab2>
```

With GOBO Eiffel compiler, most of the problems are fixed, except the two lines 9 & 10 in the middle where the programmer made direct field access (which violates the new programming rules); and again, SmartEiffel's new output is the same as GOBO Eiffel.
Note, even without direct field access, it's better for a non-accessor method to call only accessor method defined in the *same* class, for example, suppose:

Listing 14: problematical call to accessor method from the base class

```
class FACULTY
  do_benchwork() is
    do
```

```
    io.put_string(name + " do_benchwork in the: " + get_addr() + "%N");  -- i.e PERSON.get_addr()
  end
```

- FACULTY.do_benchwork() directly call accessor PERSON.get_addr() (instead of FACULTY.get_faculty_addr()), and
- STUDENT.take_rest() directly call PERSON.get_addr() (instead of STUDENT .get_student_addr())

that will be a programming error, since no matter how RESEARCH_ASSISTANT.get_addr() is implemented (or even not overridden at all), there can only be one implementation in RESEARCH_ASSISTANT, so at least one of RESEARCH_ASSISTANT.do_benchwork() and RESEARCH_ASSISTANT.take_rest() will get a wrong address.
Therefore we add the following accessor calling level rule:

Rule 3 (virtual field accessor calling level warning rule – i.e. violations are warnings instead of errors)

1. *only accessor method can call super-class' accessor method: e.g. FACULTY.get_faculty_addr() call PERSON.get_addr()*
2. *any non-accessor method can only call virtual field accessor method defined in the* same *class*

But sometimes, the programmers do need to make direct field access or call accessor methods from the base classes, too much such warning messages can be annoying. Therefore we also introduce a new syntax for such cases to silence the compiler warning messages:

Rule 4 (programmer manually verified direct field access or accessor method from the base class) *new feature access operator !:*

- `object!direct_field_access` *or*
- `object!accessor_method_from_base_class()`

3.3. One step further beyond manual fix

The compiler can be enhanced to issue error / warning messages when detecting these rule violations, and alert the programmer to check and fix them just as we did in our example. While this process works, the programmer needs to re-examine *manually* all the existing code to ensure their semantics are correct.
Eiffel first appeared in 1986, and won the ACM software system awards in 2006, there are many existing users[12] with a possible very big code base. Re-examine and fix all these code base manually is a very complex task. Can we do better to avoid this tedious manual approach and make the existing code work *as it is*? The main purpose of MI is to encourage code reuse, so we would like to make the method FACULTY.do_benchwork() work correctly according to the programmer's intention *without* adding the extra virtual accessor as we introduced in this section.
Another method is to change the compiler to implement the intended semantics of the rename fields. In the next section we will introduce a new concept called: *rename dispatching based on view stacks* to fix the semantics issue.

4. Rename dispatching based on view stack

(Note: in the following sections, code listings are for language design discussion purpose, hence may not be compilable or executable. The new method we will introduce in this section is independent of the previous section; in fact, we assume the previous section, in particular the updated research_assistant.e listing 10 with virtual accessor override does not exist at all. We only assume Eiffel's reference identity semantics of renamed field is *removed*.)
For any new method implemented in class RESEARCH_ASSISTANT (and its descendants) which is related

[12] For example, `https://www.eiffel.com/company/customers/` listed even many heavy-weight industry customers ranging from aerospace defense (e.g. Boeing Co), to finance (e.g. CBOE), and to tele-communications (e.g. Alcatel-Lucent), etc.

to its FACULTY role, it can use the renamed new field `faculty_addr`, but how about *existing* methods inherited from the super-classes e.g. FACULTY.do_benchwork() method in Listing 3, which accesses the original field via the old name `addr`?

4.1. Virtual field access dispatch

Ideally, when a super-class's method e.g `FACULTY.do_benchwork()` is called on a `RESEARCH_ASSISTANT` object, the method needs to access the renamed `addr` field with "lab" semantics, i.e `RESEARCH_ASSISTANT.faculty_addr` as the programmer introduced in the renaming clause. However in the current Eiffel, it still accesses the old field `PERSON.addr` which has "home" semantics; similarly `STUDENT.take_rest()` also wrongly access the field `PERSON.addr` in `RESEARCH_ASSISTANT`, whose "home" semantics is not what `take_rest()` expected "dorm" semantics. So we end up in the situation that these two methods from different super-classes which expect different semantics of `addr` still *share* the same field `RESEARCH_ASSISTANT.addr`, and this does not achieve feature separation at all.
So after a feature renaming, when an inherited method is called on a derived class object, it still accesses the original feature by the *old* name, which generates problematic semantics different from the programmer's intention by using renaming. The needed dispatch to the features with *new* names is neither discussed in the existing Eiffel language literature, nor implemented by any of the Eiffel compilers that we have tested. What needed is a semantical dispatch of renamed field, so we introduce the following principle:

Definition 4 (semantical dispatch principle of renamed features) *the purpose of feature renaming is to resolve feature name clash while achieving the programmer's renaming intention; when the original feature (by the old name) is accessed (both read and write) on a sub-class object in the super-class' method, that feature access needs to be dispatched to the renamed feature (by the new name) in the sub-class.*

This renamed feature dispatching semantics is different from Eiffel's original reference identity semantics: e.g. with reference identity semantics, all these three notation

1. `RESEARCH_ASSISTANT.addr`
2. `RESEARCH_ASSISTANT.student_addr`
3. `RESEARCH_ASSISTANT.faculty_addr`

refer to the same field; while with renamed feature dispatching semantics these three are *separated* fields (with different physical memory locations), and for any method call or statement that need access to the `addr` field, if the execution context is in the:

1. PERSON branch, it needs to be dispatched to `RESEARCH_ASSISTANT.addr`
2. STUDENT branch, it needs to be dispatched to `RESEARCH_ASSISTANT.student_addr`
3. FACULTY branch, it needs to be dispatched to `RESEARCH_ASSISTANT.faculty_addr`

NOTE: In traditional OOP languages without Eiffel's renaming mechanism, e.g. in C++, virtual *method* dispatch just needs to know the static type of the receiver and the runtime type of the actual object. However, for Eiffel's renamed virtual *field* dispatch this is not enough, e.g. in the same `PERSON.get_addr()` method call, the field access `PERSON.addr` can be dispatched to one of `RESEARCH_ASSISTANT.addr`, (renamed) `RESEARCH_ASSISTANT.student_addr` or `RESEARCH_ASSISTANT.faculty_addr`, depending on the actual program execution path.

4.2. Virtual field access context is object view stack dependent

In this subsection we will show that field access context is object view stack dependent.

4.2.1. Execution context: call stack

Let's examine the execution context when `STUDENT.set_student_addr()` is called on a `RESEARCH_ASSISTANT` object: the method is defined in class STUDENT.e in Listing 2, which in turn calls `PERSON.set_addr()`

method, and the final assignment statement there writes to the `addr` field, so the call stack at the assignment site is (from the top to the bottom)[13]:

Listing 15: the actual call stack of ra.set_student_addr()

```
set_addr()          --  PERSON.e line 9  with Current type: PERSON
set_student_addr()  -- STUDENT.e line 6, with Current type: STUDENT
ra.set_student_addr("dorm")           -- with Current type: RESEARCH_ASSISTANT
```

As we have just discussed, this write needs to be performed on the `student_addr` field of `RESEARCH_ASSISTANT` (please refer to the renaming DAG of Fig 2) hence[14]:

Listing 16: write to the renamed field

```
1  ra.set_student_addr("dorm");        -- write to the STUDENT.addr field
2  assert(ra.student_addr == "dorm");  -- read the renamed student_addr field
3
4  ra.set_faculty_addr("lab");         -- write to the FACULTY.addr field
5  assert(ra.faculty_addr == "lab");   -- read the renamed faculty_addr field
```

For line 1: the actual assignment statement to the field `(PERSON.)addr` is in the method `PERSON.set_addr()`, and at that assignment site (i.e. line 9 of PERSON.e of Listing 1), the call stack of the `Current` object's type (i.e. the list we get by projecting `Current`'s type out of the call stack) from bottom to the top is:

$$\text{callStackTypes} = [\text{RESEARCH\_ASSISTANT}, \text{STUDENT}, \text{PERSON}]$$

For line 4: similarly the call stack is:

$$\text{callStackTypes} = [\text{RESEARCH\_ASSISTANT}, \text{FACULTY}, \text{PERSON}]$$

Note the *same* (by-name) field `PERSON.addr` is modified by the *same* method `PERSON.set_addr()`, and is invoked on the *same* object (`ra`), but the actual assignments need to be made on two *different* actual fields: `ra.student_addr` or `ra.faculty_addr`, due to the different call stacks. Conversely, to read a renamed field e.g. `ra.get_student_addr()`, the actual call stack is:

Listing 17: the actual *call stack* of ra.get_student_addr()

```
get_addr()          --  PERSON.e line 8  with Current type: PERSON
get_student_addr()  -- STUDENT.e line 5, with Current type: STUDENT
ra.get_student_addr()                 -- with Current type: RESEARCH_ASSISTANT
```

and the *actual* field which is needed to be read here is `ra.student_addr`.

4.2.2. Execution context: view stack

Now, let's consider the following assignment statements sequence:

Listing 18: assignment chain, and *view stack*

```
1  ra: RESEARCH_ASSISTANT;
2  ra_as_student: STUDENT := ra;
3  ra_as_student_as_person: PERSON := ra_as_student;
4  ra_as_student_as_person.addr := "dorm";
```

At the last line 4, the actual assignment site, there is no method call stack; however, compare it with the previous code listing 15 of the *call stack* there, the variable `ra_as_student_as_person` here has a *view stack* of the object `ra` it holds:

$$\text{viewStackTypes} = [\text{RESEARCH\_ASSISTANT}, \text{STUDENT}, \text{PERSON}]$$

[13] The Eiffel keyword `Current` is just like `this` in C++ & Java, or `self` in Python, which represents the current object instance.
[14] Here we use "==" to mean reference identity testing (as in C++/Java).

Definition 5 (object view stack) *We generalize the concept of method call stack to* object view stack*, at the bottom of the stack is the object's actual type RESEARCH_ASSISTANT, and the* `ra` *object is assigned to variable* `ra_as_student` *of type STUDENT, and then to* `ra_as_student_as_person` *of type PERSON. At this point, the object is used (viewed) as if it's a PERSON type through a stack of lenses all the way down to the actual object of type RESEARCH_ASSISTANT.*

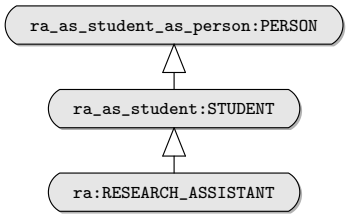


Fig. 3: view stack as a set of "lenses" on the object `ra`

Example 1 *In the following, the same* `RESEARCH_ASSISTANT ra` *object is held by different variables of its super-class type, hence has different view stacks:*

Listing 19: different views of the same object, direct field access

```
1   ra: RESEARCH_ASSISTANT  -- all the variables refer to this same RESEARCH_ASSISTANT object
2   ra_as_student: STUDENT := ra;
3   ra_as_faculty: FACULTY := ra;
4   person: PERSON := ra;
5
6   -- accessing the field by the same name does not mean accessing the same field!
7   assert(ra_as_student.addr == "dorm");  -- [RESEARCH_ASSISTANT, STUDENT] view of ra object
8   assert(ra_as_faculty.addr == "lab" );  -- [RESEARCH_ASSISTANT, FACULTY] view of ra object
9   assert(person.addr == "home");         -- [RESEARCH_ASSISTANT, PERSON]  view of ra object
10
11  -- view stack of assignment chain
12  person := ra_as_student;               -- assign ra to PERSON via STUDENT
13  assert(person.addr == "dorm");         -- [RESEARCH_ASSISTANT, STUDENT, PERSON] view of ra object
14
15  person := ra_as_faculty;               -- assign ra to PERSON via FACULTY
16  assert(person.addr == "lab" );         -- [RESEARCH_ASSISTANT, FACULTY, PERSON] view of ra object
```

note on line 13, assert(person.addr == "dorm"); while on line 16, assert(person.addr == "lab"). This shows view stack is different from usual method call stack, i.e. even in the same scope assigning to a local variable of a different type will change the view stack of the object.

Summary: every access (write and read) of a renamed field needs to be dispatched based on its renaming DAG, *and* the view stacks of the variable at the access site. To the best of the author's knowledge, this behavior has never been documented in any previous OOP literature. We call it *rename dispatching based on the view stack*.

Example 2 *As a consequence of such renamed field dispatch, now the following accessor method calls will return the intended renamed field:*

Listing 20: different views of the same object, accessor method call

```
ra_as_student.get_addr();  -- now return "dorm"
ra_as_faculty.get_addr();  -- now return "lab"
       person.get_addr();  -- now return "home"
```

which means we can make the existing code work as it is by adding rename dispatching to the compiler.

4.3. Implementation: variable as object and view stack holder

An object can be held by different variables, and each variable holds a different view (history) of the object. For example, a same `RESEARCH_ASSISTANT` object can be passed to both the methods call `do_benchwork()` and `take_rest()` simultaneously, e.g on two different threads. So the view stack cannot be hold by the object itself (then will be shared by two different threads); instead each execution context need to maintain its own separate view stack of the object.

Definition 6 *Each variable holds a "fat pointer", which has two components:*

1. *the actual object*

2. *the view stack of the object*

And we introduce the following rules to update view stacks:

Rule 5 (view stack updating rule) *Note, in the following rules,* variables *also include compiler generated temporary variables (not visible by the programmer)*

1. *object creation: var:V = new O(), when an object of type O is created and assigned to var of type V, then*

$$view\_stack(var) = [O,\ V]$$

2. *assignment: var:V = u, when a variable u is assigned to a variable of type V, then*

$$view\_stack(var) = view\_stack(u) + [V]$$

i.e, push type V on to the top of the view stack.

3. *function call: function(var:V) be called as function(u), when a variable u is passed to a function as a parameter of type V, then*

$$view\_stack(var) = view\_stack(u) + [V]$$

i.e. push type V on to the top of the view stack.

Example 3 (assignment chain) `ra` *is first assigned to* `ra_as_student`*, and then to* `person`*, then:*

$$view\_stack(person) = [RESEARCH\_ASSISTANT,\ STUDENT,\ PERSON]$$

4.4. Rename dispatching

Each variable carries a type stack, which holds the type information of the object's view history.

Definition 7 (Ordered Subsequence (order-preserving but may not be contiguous)) *An ordered sequence $A = (a_1, a_2, ..., a_k)$ is a subset of an ordered sequence $B = (b_1, b_2, ..., b_n)$ if there exists an increasing sequence of indices $1 \leq i_1 < i_2 < ... < i_k \leq n$ such that:*

$$a_1 = b_{i_1}, a_2 = b_{i_2}, ..., a_k = b_{i_k}$$

and we denote it as $A \subseteq B$.

Rule 6 (Virtual field dispatching rule) *Given an object's view stack S, find the shortest path P in the object's field's renaming DAG from the top(S) to the bottom(S), such that $S \subseteq reverse(P)$,*

1. *if there is only one shortest path, dispatch to the field's final rename in the renaming DAG.*
2. *if there are multiple such shortest paths, raise a run-time exception.*

Now let us review our previous write and read access to the virtual field `addr`:

Example 4

1. *In Listing 15, at the assignment site (line 9 of class PERSON in Listing 1):*

$$view\_stack(Current) = [RESEARCH\_ASSISTANT,\ STUDENT,\ PERSON]$$

we can find the corresponding path in Fig 2 of the renaming DAG of field `addr` *is:*

$$PERSON \rightarrow STUDENT \rightarrow RESEARCH\_ASSISTANT$$

and the final name of the field is `student_addr`*, so the assignment of string "dorm" in set_addr() to virtual field* `addr` *will be made on the actual field* `ra.student_addr`*.*

2. *and for line 7 of Listing 19,*

$$view\_stack(ra\_as\_student) = [RESEARCH\_ASSISTANT,\ STUDENT]$$

we can find the corresponding path in Fig 2 of the renaming DAG of field `addr` *is:*

$$STUDENT \rightarrow RESEARCH\_ASSISTANT$$

and the final name of the field is `student_addr`*, so the field access of virtual field* `addr` *will be dispatched to the actual field* `ra.student_addr`*, thus the assertion holds:*

```
assert(ra_as_student.addr == "dorm");  --  [RESEARCH_ASSISTANT, STUDENT] view of ra object
```

4.5. Normalized view stacks, and dispatch optimization

With multiple inheritance, an object's view stack can be more complex than just along a single linear branch, let's consider the following MI DAG: suppose there is an object of type Child, this object can be assigned (with compiler generated type checking) to a reference variable of any of its base class type, and in any sequence. In Eiffel this is called assignment attempt, and in other languages (e.g. C++ / Java) it is called (type checked) cast. For example, consider the following assignment attempts:

Listing 21: type cast (Eiffel)

```
1  child: Child
2  base_b: Base_B
3  derived_a: Derived_A
4
5  base_b := child
6  derived_a ?= base_b  -- type cast
7  if derived_a /= Void then
8    derived_a.some_method_call()
9  end
```

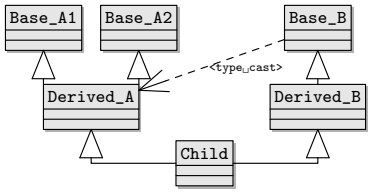


Fig. 4: type cast from Base_B to Derived_A

All the assignment attempts will succeed, so the view_stack(derived_a) at line 8 is [Child, Base_B, Derived_A], and in particular Base_B can be totally unrelated to Derived_A. This example shows in MI, it is possible to cast to a variable of type A which is on a *different* branch in the inheritance DAG from the object's current holding variable's declaration type B. To simplify view stack management, let's define the *normalized* view stack:

Definition 8 (Normalized view stack) *Let view stack*

$$S = [T_0, T_1, \ldots T_{n-1}]$$

and T_0 be the object's actual type at the bottom of the stack, we say S is a normalized stack, if for any $j > i$, T_j is a superclass of T_i. Or in short, all the types in a normalized view stack needs to be in the strict monotonic super-classing total order.

To normalize an arbitrary view stack, we introduce the following rules:

Rule 7 *If two adjacent elements of the view stack are the same [A, A], normalize it to [A].*

Rule 8 (type cast) *Suppose [A, B] are the top two elements of a view stack, i.e. VS + [A, B], then update the view stack:*

1. *first cast to the* greatest common derived class *gcd(A, B) of A and B: VS + [gcd(A, B)]*
2. *then cast from gcd(A, B) to B: VS + [gcd(A, B) + B]*

Example 5 *normalize view stack [RA, FACULTY, STUDENT]:*

1. $[RA, RA]$*, since RA is the gcd(FACULTY, STUDENT)*
2. $[RA, RA, STUDENT]$
3. $[RA, STUDENT]$

Example 6 *cast up and down along the same branch: normalize view stack [RA, FACULTY, RA]*

1. $[RA, RA]$*, since RA is the gcd(FACULTY, RA)*
2. $[RA]$

The normalized view stack can only effectively grow in the direction to the root class of the inheritance DAG.

Theorem 1 (limited view stack length) *The max view stack length is the max depth of the inheritance DAG.*

The most straightforward implementation of the dispatch method is at least linear to the stack length; to speed up, we can create a hash-table by enumerating all the possible type view stacks during the compilation, and reduce the runtime dispatching cost to O(1) (since both the full inheritance and field renaming DAG are known at compile time, the compiler can create a perfect hash-table).
In Eiffel [Mey97] Section 15.4 repeated inheritance is allowed, to support it in virtual field dispatch we need the following rewrite rule

Rule 9 (Repeated inheritance rewrite) *Repeated inheritance can be easily supported by the following simple rewrite rule:*

```
class B inherit A, A   -- repeatedly inherit A multiple times by the same sub-class

-- rewrite to
class A1 inherit A
class A2 inherit A
class B  inherit A1, A2
```

5. Implementation: virtual field dispatch based on view stacks

Due to paper length limit, the specifics of our implementation of what we discussed in the previous section, and the full code of demo.yi (which is equivalent of the Eiffel code of Section 2), are in the Supplementary Material 10. Our new output is:

As we can see, the results are all what the programmer has expected now. We also demonstrate *on purpose* that two distinct fields `Student._student_age` and `Faculty._faculty_age` are joined into one single field `ResearchAssistant._age`, which the three Eiffel compilers all failed to join with the message: "Error: two or more features have the same name (age)."

Listing 22: demo.yi output

```
ResAsst has 3 addresses: <home dorm lab>                          1
ResAsst do_benchwork in the: lab                                  2
ResAsst take_rest in the: dorm                                    3
-- print_student|faculty_addr_direct_field                        4
ResAsst as STUDENT.addr: dorm, age=18                             5
ResAsst as FACULTY.addr: lab, age=18                              6
-- print_student|faculty_addr_via_accessor                        7
ResAsst as STUDENT.addr: dorm                                     8
ResAsst as FACULTY.addr: lab                                      9
-- test some assignment: suppose ra moved both lab2 and dorm2     10
ResAsst has 3 addresses: <home dorm lab2>                         11
ResAsst has 3 addresses: <home dorm2 lab2>                        12
```

5.1. Legacy Eiffel code migration plan: combine the two methods

As we can see in this second method, every field access (except the raw access operator with "!") becomes a method call, and for every variable assignment (including parameter passing in method call) the language runtime needs to maintain the object view stacks. These operations will increase both the memory and runtime overhead of the compiled program. While the first method we introduced in the previous Section 3 is more efficient at runtime. Actually to migrate legacy Eiffel code, we can combine these two methods:

1. first, use the second method of this section enhanced compiler to generate test cases for the intended semantics.
2. then, auto generate new and override semantic assigning accessor methods according to Rule 2 & 3 for the corresponding abstract virtual fields, and use the first method enhanced compiler to validate these test cases,

6. Empirical study and evaluation

We have done some experiments to benchmark the performance of virtual field dispatch of Sections 4 and 5 v.s. regular virtual method dispatch: call a virtual field dispatch method, and a regular virtual method each 1,000,000 times and measure 3 times, and take the average, the results are:

- virtual field dispatch: [1165, 1172, 1189] milliseconds
- regular virtual method dispatch: [6, 6, 6] milliseconds

(The testing system is Ubuntu 22.04.3 LTS, x86_64 GNU/Linux, running on AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx, 16GB memory.)
So the virtual field dispatch is about 195x slower than the regular virtual method dispatch. As previously mentioned, the new abstract virtual fields we just described in the preceding Sections 4 and 5 is mainly intended to be used as a migration tool for existing code but not as release runtime in real production environment, since its runtime performance is severely limited by what can be done given the constraint not to change any existing customer code.
We have communicated with Prof. Bertrand Meyer, the chief designer of Eiffel, and he has acknowledged the semantic issues reported in this paper are real problems and need to be fixed. However, even when the fields renaming semantics be made precise, and updated in the Eiffel language specification in the future (which may take *years* to be done given the past experience), it will certainly be a *breaking* change to the existing code (e.g. just consider the discrepancies between ISE compiler's workbench mode and finalized mode, at least one of them will break – actually both are wrong in terms of the ideal application semantics).
On the other hand, in Part 2 of this paper a new clean and general solution to the diamond problem of multiple inheritance has been developed, which is applicable across a wide spectrum of industry-strength OOP languages. In particular, it is also applicable to Eiffel *without* relying on Eiffel's renaming mechanism, which provides an alternative method to migrate existing Eiffel code.
Until such changes occur, the best thing we can do for now to help the current Eiffel users is to develop a tool that can detect the potential feature renaming problem and alert the Eiffel programmers accordingly, so they can be made aware of the issues and avoid fields renaming in diamond inheritance. Following discussions with the Eiffel community leaders and compiler implementers, we have developed and provided this tool to the Eiffel user community as of March 2024. The new tool can be found in the supplemented file `gobo-detect_diamond.zip`.

6.1. Problems found in real code

We have run the new detection tool we developed on the source code distributed with the official ISE Eiffel latest 23.09 release, found problems in the real code:

- In total there are total 11958 classes, and the tool reported 24 unique renaming problems in diamond inheritance.
- In particular, in EiffelVision 2 GUI library[15], there are 1194 classes, the tool reported 7 unique renaming problems in diamond inheritance.

The detailed listing is in the supplementary vfield_sup.pdf.
For example, in the following ISE 23.09 source code directory:

Listing 23: rename problems found in diamond inheritance in real code

```
$ cd $ISE_EIFFEL/examples/docking/simple/
$ $GOBO/tool/gedoc/detect_diamond.py .
check:  .
...
======================================== ./docking_simple.ecf total fields:  15892
diamond found, full paths:
   EV_PIXMAPABLE.implementation  =>  SD_PLACE_HOLDER_ZONE.{implementation, implementation_upper_zone}
   implementation {('EV_PIXMAPABLE.implementation',
                    'EV_CONTAINER.implementation',
```

[15] https://www.eiffel.org/doc/solutions/EiffelVision_2

```
                    'EV_CELL.implementation',
                    'SD_DOCKING_ZONE.implementation',
                    'SD_PLACE_HOLDER_ZONE.implementation')}
   implementation_upper_zone {('EV_PIXMAPABLE.implementation',
                               'EV_CONTAINER.implementation',
                               'SD_UPPER_ZONE.implementation_upper_zone',
                               'SD_PLACE_HOLDER_ZONE.implementation_upper_zone')}
new_fields: SD_PLACE_HOLDER_ZONE.{implementation, implementation_upper_zone}
diamond core:  EV_CONTAINER.implementation
   => SD_PLACE_HOLDER_ZONE.{implementation, implementation_upper_zone}
   implementation [('EV_CONTAINER.implementation',
                    'EV_CELL.implementation',
                    'SD_DOCKING_ZONE.implementation',
                    'SD_PLACE_HOLDER_ZONE.implementation')]
   implementation_upper_zone [('EV_CONTAINER.implementation',
                               'SD_UPPER_ZONE.implementation_upper_zone',
                               'SD_PLACE_HOLDER_ZONE.implementation_upper_zone')]
```

Basically, the report says: the same field from the diamond top class `EV_PIXMAPABLE.implementation`, become two different fields in the diamond bottom class: `SD_PLACE_HOLDER_ZONE.implementation` and `SD_PLACE_HOLDER_ZONE.implementation_upper_zone`, via fields renaming on two different diamond-inheritance paths.
When we run the finalized build of this `docking_simple` code example (tested on Ubuntu 22.04.3 LTS, x86_64), the program will fail:

```
/Eiffel_23.09/examples/docking/simple$ ./EIFGENs/docking_simple/F_code/docking_simple

docking_simple: system execution failed.
Following is the set of recorded exceptions:

****************************** Thread exception ******************************
In thread           Root thread            0x0 (thread id)
******************************************************************************
------------------------------------------------------------------------------
Class / Object      Routine                Nature of exception           Effect
------------------------------------------------------------------------------
VISION2_APPLICATION root's creation        Segmentation fault:
<00007FC1A8207558>                         Operating system signal.      Exit
------------------------------------------------------------------------------
VISION2_APPLICATION root's creation
<00007FC1A8207558>                         Routine failure.              Exit
------------------------------------------------------------------------------
```

6.2. Eiffel compiler migration plan

Based on the work of previous sections, we suggest the following compiler migration plan to the Eiffel community:

1. Deprecate `rename`, because of the semantics issues we have found in this paper.
2. In particular, *prohibit* `rename` in diamond inheritance, using the detection tool in the previous section.
3. For ISE compiler: remove `rename`'s reference identity semantics in its workbench mode.
4. Use the following alternative methods to implement diamond inheritance if the programmers have to:
   (a) the new programming rule we developed in Section 3, or
   (b) the DDIFI design pattern that we developed in Part 2 of this paper.

7. Discussions

7.1. Renamed methods: dispatch first by view stack then by virtual table

In Eiffel, methods can also be renamed, and the renamed methods need to be treated in the same way as renamed fields.

In 2015, an Eiffel programmer found and raised a similar question regarding renamed methods on the web: `https://stackoverflow.com/questions/32498860/`

> I am struggling to understand the interplay of multiple inheritance with replication and polymorphism. Please consider the following classes forming a classical diamond pattern.
>
> If I attach an instance of D to an object `ob_as_c` of type C, then `ob_as_c.c` prints "c" as expected. However, if attach the instance to an object `ob_as_b` of type B, then `ob_as_b.b` will also print "c".
>
> Is this intended behavior? Obviously, I would like `ob_as_b.b` to print "b".

```
deferred class A
  feature
    a  deferred end
  end

deferred class B
  inherit A
    rename a as b end
  end

deferred class C
  inherit A
    rename a as c end
  end

class D
  inherit
    B
    C select c end
feature
  b do print("b") end
  c do print("c") end
end
```

Both Bertrand Meyer and Emmanuel Stapf (the lead developer of EiffelStudio) replied to that question, but they only mentioned that in (virtual) dynamic binding, there is only one version of the method on D, can be called.
Instead we think, firstly, this is another problem in Eiffel's `select` clause as we have mentioned earlier: after the two feature renamings (separation) there is *no* more name clash on `a`, however they are forced to be joined again (by the `inherit C select c`), i.e `ob_as_b.b` actually calls `d.c`. As the programmer who asked question finally put it: "Unfortunately, this makes Eiffel's inheritance features much less useful to me."
Secondly, just same as our treatment of abstract virtual fields: the compiler should keep methods `D.b & D.c` separate, and the `select` clause is no longer needed. Now dispatch to `ob_as_b.b`, `ob_as_c.c` and even `ob_as_a.a` by the renaming DAG and the view stack first, and then dispatch as a virtual method (if there is any further override), as we did in Section 4. This will achieve what the user wanted in the original question.

7.2. Compare virtual method dispatch and rename dispatching

Virtual *field* dispatch is different from virtual *method* dispatch in that virtual field dispatch depends on *the whole view stack* from the holding variable type down to the actual object type; while virtual method dispatch depends on *only* the actual object type.

8. Related work

There is a long history of inheritance and polymorphism have been studied since 1980s, and there are copious literature on multiple inheritance, e.g. [CP93], [Car88], [vLM96], [Tai96], [BC90]. In the following we will only compare some of the earlier work that has some similarity as our work, and highlight the difference.
In [CG90] Carré and Geib introduced the point-of-view (PoV) notion of multiple inheritance. While both their work and our work try to solve the same problem of dynamic dispatching of class fields, our work differs from their PoV in the following ways:

1. Our view stack approach dynamically tracks the *full object-to-target runtime path*, while their approach only considers the <source object, target type> *two points pair*.
2. In their paper, they did not give the detailed rules on *how* to do selection (i.e dispatch), especially if there are *multiple paths* between PoV<C, O>, (and its even not clear if they handle such cases at all); while we give very specific Rule 6 on how to dispatch, even for *multiple paths*.
3. Their approach only uses the *global* inheritance lattice for *all* the Pov(C, O) calculation; while in our

approach, *each renamed field has its own renaming DAG* (which can be different from the global inheritance DAG), so in our stack view dispatch calculation we use each fields own renaming DAG (and for never-renamed field, e.g. `name`, there is no need for dispatch).

4. In their approach, all inherited fields are separated; while in our approach fields can be shared or separated (renamed) according to the programmers application semantics.

In [CUCH82], the sender path tiebreaker rule of SELF can only handle the case where if "only one of the parents is an ancestor or descendant of the object containing the method that is sending the message (the sending method holder)", which means this rule cannot handle the inheritance relationship in the diamond problem, while our approach can handle it.
In [BI82], Borning and Ingalls discussed the multiple inheritance in Smalltalk-80. They also considered when the same method is inherited via several paths, an error will be reported to the user. In their paper, they did not give the detail of the method resolution method that they used. While our approach gives every specific rules on how to do runtime dispatch, and when to raise exceptions.
In [BMCP18], an orthogonal policy Object Type Integrity (OTI) is proposed to dynamically track C++ object types. So instead of allowing a set of targets for each dynamic dispatch on an object, only the single, correct target for the objects type is allowed. Their work is based on C++ class model where there is no feature renaming, while our work is toward fixing Eiffel's renaming mechanism.
In [Ngo21], Ngomo described a non-linear and non-deterministic approach of the semantics of multiple inheritance in their problem domain, i.e. multiple specification of logical objects. While our approach is deterministic, so the programmers can easily reason about the programs.
In [SP20], Suchánek and Pergl studied the method resolution order in Python, and showed that inheritance can be implemented with minimisation of combinatorial effect using the inheritance implementation patterns.

9. Conclusions

We found the reference identity semantics of Eiffel's field renaming mechanism is problematic, as demonstrated in the diamond problem of MI. We introduced a new concept called abstract virtual fields and proposed two methods to solve it:

1. always add new and use semantic assigning accessors, avoid direct field access.
2. rename dispatching based on view stack.

Some future work directions:

- On the theoretical aspects: study the interplay of abstract virtual fields with other language constructs. In OOP virtual method is a well studied concept, while we have not found any discussion of abstract virtual fields. The interplay of abstract virtual fields with other constructs of the OOP language is yet to be explored for new opportunities (or new problems).
- On the practical aspects: design more efficient implementations. Design and implement more efficient management strategy to reduce view stack memory requirement and dispatch runtime overhead.

In the second part of this paper, we will generalize our first method of *abstract virtual fields* to a new design pattern, which cleanly and generally solves the diamond problem in a wide range of industry strength OOP languages: C++, Python, OCaml, Lisp, Eiffel, Java, C#, Scala, D, etc. *without* using Eiffel's renaming mechanism.

10. Supplementary Material

The source code of this paper can be found in the supplementary material: eiffel_rename.zip and gobo-detect_diamond.zip, including the demo of the Eiffel rename semantics issue we found, the two proposed fixing methods, and the detailed description doc vfield_sup.pdf.

Part 2: Decoupling implementation inheritance from interface inheritance as a *clean* and *general* solution to multiple inheritance

The most difficult challenge in multiple inheritance (MI) is exemplified by the well-known "diamond problem", leading to MI's avoidance in most contemporary OOP languages, such as Java, C#, and Scala etc., which primarily advocate for single inheritance. Nevertheless, to address the absence of MI while maximizing code reuse, alternative techniques such as object composition, mixins / traits have been employed. However, these existing approaches have fallen short in effectively resolving naming conflicts, especially for inherited *fields* conflicts. In particular, they have not provided a satisfactory mechanism for programmers to specify, on an *individual* basis, how each inherited field should be joined or separated. In this paper we introduce a novel method that can achieve this, hence providing a *clean* and *general* solution to multiple inheritance. Our method is applicable across a wide spectrum of industry-strength OOP languages. We will compare the advantages of our method with those of existing methods, including (C++'s) virtual inheritance, object composition, and mixins / traits.

Traditionally in class based OOP languages, *both the fields and methods* from the super-classes are inherited by the sub-classes. However we believe this is the root cause of many problems in multiple inheritance, e.g. most notably the diamond problem. In this paper, we propose to *stop inheriting data fields* and Decouple Implementation Inheritance From Interface Inheritance (DIIFII), which is a derived principle from the more general principle: Decoupling Data Interface From data Implementation (DDIFI), as a clean and general solution to such problems. We first present a design pattern called DDIFI [16] which cleanly solves the diamond problem in C++. It can handle the instance fields of the multiple inheritance exactly according to the programmer's intended application semantics. It gives programmers flexibility when dealing with the diamond problem for instance variables: each instance variable can be configured *individually* either as one joined copy or as multiple independent copies in the implementation class. The key ideas are:

1. decouple implementation inheritance from interface inheritance by stopping inheriting data fields;
2. in the regular methods implementation use *virtual* property methods instead of direct raw fields; and
3. after each semantic *branching* add (and override) the new semantic assigning property.

Then we show our method is general enough, and can also achieve clean multiple inheritance in any OOP languages:

- that natively support multiple inheritance (e.g. C++, Python, OCaml, Lisp, Eiffel, etc.)
- single inheritance languages that support default interface methods (e.g. Java, C# etc.)
- single inheritance languages that support mixins (e.g. D), or traits (e.g. Scala)

(The supplementary DDIFI.tgz contains the implementation source code of this design pattern DDIFI in these 9 languages.)

Moreover, in this paper we introduce a `<Person, Student, Faculty, ResearchAssistant>` inheritance Problem 1 as a working example of our method. We believe if one can solve this problem, then one can solve *all* the problems that can arise in multiple inheritance. Finally, we would like to propose the following challenges to the whole programming language research community:

1. Design another alternative cleaner solution (than ours) to the diamond problem of ResearchAssistant Problem.
2. Devise another multiple inheritance problem (be it diamond problem or not), which *cannot* be solved by using the method of this paper.

We hope the success of DDIFI will guide the future OOP languages design and implementations.

[16] The DDIFI design pattern is patent pending.

11. Motivation: the diamond problem

Let us continue using the example from Part 1 of this paper to build an object model for Person, Student, Faculty, and ResearchAssistant in a university:

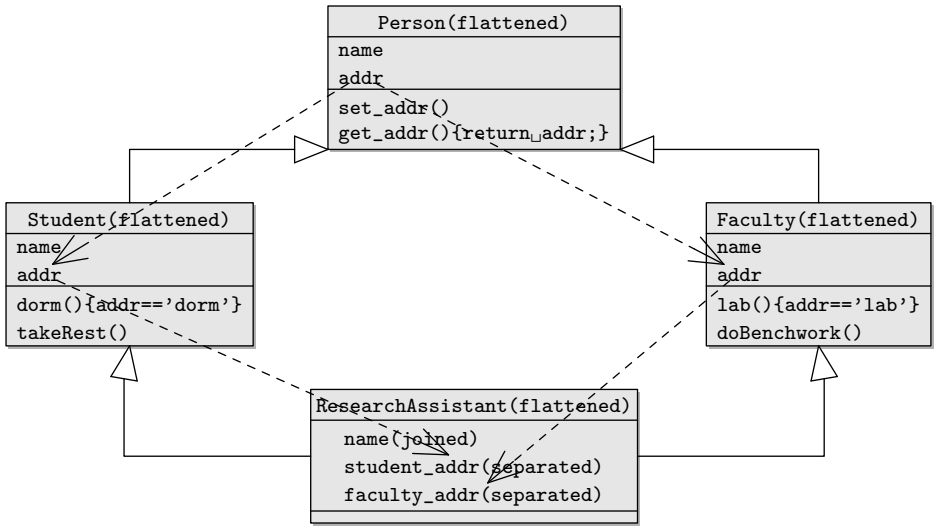


Fig. 5: the diamond problem: the ideal semantics of *fields* `name` & `addr`, which is not achievable in C++'s plain MI mechanism: with `name` joined into one field, and `addr` separated into two fields

We would like to formulate this ResearchAssistant inheritance problem as:

Problem 1 (The intended application semantics) *In the diamond inheritance of* `<Person, Student, Faculty, ResearchAssistant>`*, a* `ResearchAssistant` *should have*

- *only 1* `name` *field,*
- *but 2 different address fields:*
  1. *one* `"dorm"` *as Student to* `takeRest()`*, and*
  2. *one* `"lab"` *as Faculty to* `doBenchwork()`

so in total 3 fields.

In the following we will use this problem as the working example to illustrate our method to simultaneously achieve both the field joining and separation in multiple inheritance, just as the dining philosophers problem as the working example in concurrency algorithm design.

12. Literature survey and current engineering practices

12.1. C++'s virtual inheritance

In C++'s plain MI we can do either:

1. virtual inheritance: ResearchAssistant will have 1 name, and 1 addr; in total 2 fields, or
2. default inheritance: ResearchAssistant will have 2 names, and 2 addrs; in total 4 fields

It is difficult to achieve the intended application semantics. As is shown in the following C++ code:

Listing 24: plain_mi.cpp

```
1 #include <iostream>
2 #include <string>
3 typedef std::string String;
```

```
4
5  #define VIRTUAL // virtual  // no matter we use virtual inheritance or not, it's problematic
6
7  class Person {
8   protected:
9    String _name;  // need to be joined into one single field in ResearchAssistant
10   String _addr;  // need to be separated into two addresses in ResearchAssistant
11  public:
12   virtual String name() {return _name;}
13   virtual String addr() {return _addr;}
14 };
15
16 class Student : public VIRTUAL Person {
17  public:
18   virtual String dorm() {return _addr;}  // assign dorm semantics to _addr
19   void takeRest() {
20     std::cout << name() << " takeRest in the " << dorm() << std::endl;
21   }
22 };
23
24 class Faculty : public VIRTUAL Person {
25  public:
26   virtual String lab() {return _addr;}  // assign lab semantics to _addr
27   void doBenchwork() {
28     std::cout << name() << " doBenchwork in the " << lab() << std::endl;
29   }
30 };
31
32 class ResearchAssistant : public VIRTUAL Student, public VIRTUAL Faculty {
33 };
34
35
36 int main() {
37   std::cout << "sizeof(Person)  = " << sizeof(Person)  << std::endl;
38   std::cout << "sizeof(Student) = " << sizeof(Student) << std::endl;
39   std::cout << "sizeof(Faculty) = " << sizeof(Faculty) << std::endl;
40   std::cout << "sizeof(ResearchAssistant) = " << sizeof(ResearchAssistant) << std::endl;
41 }
```

Hence if the programmers use C++'s multiple inheritance mechanism plainly as it is, `ResearchAssistant` will have either one whole copy, or two whole copies of `Person`'s all data members. This leaves something better to be desired. E.g this is why the Google C++ Style Guide [Goo22] (last updated: Jul 5, 2022) gives the following negative advice about the diamond problem in MI:

> Multiple inheritance is especially problematic, ... because it risks leading to "diamond" inheritance patterns, which are prone to ambiguity, confusion, and outright bugs.

Because the C++ inheritance mechanism (virtual or not) always treats all the fields from the super-class as a *whole*, no matter how we combine virtual and non-virtual inheritance in any possible way, it will not achieve the goal that we want: i.e. support both field join and separation flexibly according to any application semantics the programmers needed. Moreover, Wasserrab et al. [WNST06] presented an operational semantics and type safety proof for multiple inheritance in C++, and they concluded the combination of virtual and non-virtual inheritance caused additional complexity at the semantics level.

12.2. Other industry strength OOP languages

Other OOP languages have designed different mechanisms, among the most popular OOP languages (besides C++) used in the industry:

- In Python [VRDJ14] all the inherited fields are joined by name (a Python object's fields are keys of an internal dictionary), hence there is no direct language inheritance mechanism to achieve field separation.
- Java [GJSB00] and C# [HWG03] get rid of multiple inheritance in favor of the simple single inheritance and multiple interfaces, and advise programmers to use composition to simulate multiple inheritance when needed.

12.3. Method resolution order

The Eiffel [Mey93] `select` clause allows the programmer to *explicitly* resolve name clash on each inherited feature *individually*, which we believe is a better solution than imposing the *same* method resolution order (MRO) [BCH+96] to *all* features as in many other OOP languages, e.g. Python [vR10]: the base classes' order in the inheritance clause should *not* matter.
Also, using any kind of *fixed* precedence order (of one class over another class) does not always work: for example, suppose in the application semantics the `ResearchAssistant` simultaneously wants

- `Faculty.library_privilege()` precedence over `Student.library_privilege()`, *and*
- `Student.cafe_discount_benefit()` precedence over `Faculty.cafe_discount_benefit()`

Actually this example shows why the Wikipedia definition of the diamond problem on methods is *not really* a problem as stated there: no compiler can *auto-magically guess* which method has the correct *application semantics* that the programmers have in their *mind*; the programmers must *override* the method in the bottom class D to make it explicit.

12.4. Eiffel's feature (attribute in particular) renaming mechanism

Eiffel [ECM06] also provides a feature renaming mechanism which is designed to solve the name clashing of the inherited features in the subclasses: inherited features with the same name are joined, while inherited features with different names are separated. However, as we discovered in Part 1 that there is a language semantics issue in Eiffel's field renaming mechanism when applied to the diamond problem of multiple inheritance, and it has been confirmed by showing divergent and problematic outputs for the same example code in three major different Eiffel compilers. Actually it is this discovery that prompted us to develop our method reported in the second part of this paper.

12.5. Multiple inheritance via composition

For OOP languages which do not directly support multiple inheritance, it is usually suggested to simulate multiple inheritance via composition. As illustrated in the following:

Listing 25: MI via composition in Java

```
// multiple inheritance via composition
class ResearchAssistant implements StudentI, FacultyI {  // suffix 'I' means Interface
  Student _theStudentSubObject; // composition
  Faculty _theFacultySubObject; // composition

  // Problem 1, code duplication: manual forwarding for *every* methods is very tedious
  void doBenchWork()   { _theFacultySubObject.doBenchWork(); }
  void takeRest()      { _theStudentSubObject.takeRest(); }
  String lab()  { return _theFacultySubObject._addr; }
  String dorm() { return _theStudentSubObject._addr; }

  // Problem 2, data duplication: need mutex, and keep *multiple duplicate* fields in sync
  Object set_name_mtx; // need extra mutex var

  String name() {
    synchronized (set_name_mtx) {
      String r = _theStudentSubObject._name;
      return r;
    }
  }

  String name(String name) {
    synchronized (set_name_mtx) {
      _theStudentSubObject._name = name; // dup fields
      _theFacultySubObject._name = name;
    }
  }
};
```

First, *logically* speaking we think this method is abusing "Has-A" relationship as "Is-A" relationship. (i.e. a ResearchAssistant "Is-A" both Student and Faculty object, not "Has-A" both Student and Faculty objects). Mixing two distinct concepts "Has-A" and "Is-A" into one language construct can hinder the reasoning and comprehension of both the code authors and readers, particularly within typical multi-person teams in industrial settings.
Furthermore, with *manual* method forwarding, which is not only very tedious, but also incur data duplcation, e.g.

- Each `ResearchAssistant` object will contain both a `Student` and `Faculty` sub-object, so the fields `name` are duplicated in both the sub-objects.
- In multi-threaded environment, assign new value to the two `name` fields in these two sub-objects need to be protected by a synchronization lock, which increases the complexity of the software.

12.6. Mixins / traits

In some other single inheritance OOP languages, various forms of mixins are introduced to remedy the lack of MI. Informally a mixin / trait is a named compilation unit which contains fields and methods to be *"inlined (copy/pasted)"* [17] rather than *inherited* by the client class to avoid the inheritance relationship, e.g.:

- Mixins [BC90] in Dart, D [BAP20], Ruby [FM08], etc.
- Traits [SDNB03] [DNS+06] in Scala [Ode10], PHP [Loc15], Pharo Language [TDP+20], Hack [Ott18] etc.

However, the problems with mixins and traits are:

1. There is no clean and flexible way to resolve conflicts between fields with the same name that are included from *multiple* different mixins, making it difficult to join or separate fields arbitrarily to support diverse application semantics. This limitation is particularly evident in mainstream, industry-grade OOP languages. Our method aims to address and resolve this issue.
2. Furthermore, an object of the type of the including class cannot be cast to, and be used as the named mixin type (i.e no subtype relationship), which means it paid the price of the inheritance ambiguity of (e.g. as C++'s plain) MI, but does not enjoy the benefit of it.

12.7. Other experimental languages / constructs

The LAVA programming language [Kni99] is an extension of Java with a new language construct called wrappers [BW00], [TJJV04] presented a method in LAVA to solve the diamond problem using a delegation-based object system with wraps clauses. However, object delegation increases the complexity of the overall system with many extra participating objects at runtime. It also has the same issues as composition we discussed above.
[MA09] proposed a new language CZ, which supports multiple inheritance but forbids diamond inheritance, and showed how to convert a diamond inheritance scheme to one without diamonds. While this works for their new language, we think it is counter-intuitive since diamonds arise naturally in the languages that support multiple inheritance. Instead of restricting what the programmers cannot do, we want a solution that can work with most current mainstream OOP languages.
Additional surveys of previous work are available in the related work section of [MA09], and we encourage interested readers to explore their paper.
In short, while these experimental languages are of interest to the academic researchers, we want to find a practical solution that can be directly applied in the current industry-strength languages. In this paper we have designed a new design pattern which can cleanly achieve multiple inheritance according to the programmers intended semantics. It gives programmers flexibility when dealing with the diamond problem for instance variables: each instance variable can be configured *individually* either as one joined copy or as multiple independent copies in the implementation class.
In Section 13 we will demonstrate our method in C++ using the previous example step by step; in Section

[17] E.g. from Hack Documentation: `https://docs.hhvm.com/hack/traits-and-interfaces/using-a-trait`

14 we will formalize our design pattern and present new programming rules that make our method also work in some other OOP languages; in Section 15 we will demonstrate our method in Java with the same example using these rules. In Section 16 we will compare our method with MI via composition, and other approaches like mixins / traits. In Section 17 we will discuss the advantages and disadvantages of DDIFI, and empirical evaluation based on a case study. Finally, in the Supplement A, we will show our method in Python and C#.

13. Decoupling data interface from data implementation

One of the most important OOP concepts is encapsulation, which means bundling *data* and *methods* that work on that data within one unit (i.e. class). As noted, inherited method conflicts are relatively easy to solve by the programmers by either overriding or using fully quantified names in the derived class.

13.1. Troublemaker: the inherited fields

But for fields, traditionally in almost all OOP languages, if a base class has field `f`, then the derived class will also have this field `f`. The reason that the inherited data members (fields) from the base classes causing so much troubles in MI is because fields are the actual memory implementations, which are hard to be adapted to the new derived class, e.g.:

- Should the memory layouts of all the different base classes' fields be kept intact in the derived class? and in which (linear memory) order?
- How to handle if the programmers want *some* of the inherited fields from different base classes to be merged into one field (e.g. `name` in the above example), and *others* separated (e.g. `addr` in the above example) according to the application semantics?
- What are the proper rules to handle all the combinations of these scenarios?

Mathematically with multiple inheritance, the class hierarchy forms an inheritance lattice, where each class node may introduce new fields. Embedding this *lattice* structure (i.e., the memory layout of all classes) into the computer's *linear* memory address space is challenging; additionally, up-casting the bottom class to any of its superclasses while maintaining memory layout integrity becomes awkward.
So the key inspiring question: since instance fields are the troublemakers for MI, can we just remove them from the inheritance relation? or delay their implementation to the last point?

13.2. The key idea: reduce the data dependency on fields to methods dependency on properties

Let us step back, and check what is the minimal dependency of the class methods on the class data? Normally there are two ways for a method to read / write instance fields:

1. directly read / write the raw fields
2. read / write through the getter / setter methods

Definition 9 (getter and setter method) *In OOP we have*

- *The getter method returns the value of a instance field.*
- *The setter method takes a parameter and assigns it to a instance field.*

In the following, we call getter and setter as property method *or just* property*; and we call the collection of properties of a class as the* data interface *of the class; In contrast we call the other non-property class methods as* regular methods *or just* methods.

In Fig.5 and `plain_mi.cpp`, we can see the field `Person._addr` has been assigned two different meanings in the two different inheritance branches: in class `Student` it's assigned "dorm" semantics, while in class `Faculty` it's assigned "lab" semantics.

Definition 10 (semantic branching site of property) *If a class* `C`*'s property* `p` *has more than one semantic meanings in its immediate sub-classes, we call* `C` the semantic branching site *of* `p`*; If class A inherits from class B, we call A is* below *B.*

In our previous example, class `Person` is the semantic branching site of property `addr`; and class `Student` is below `Person`.

Since properties are methods which can be overridden in the sub-classes — intuitively this means properties are more manipulatable than the raw data fields by the programmers (as in most OOP languages there is no mechanism for the programmers to change the semantics or implementation of an inherited data field in the sub-class). And by using properties, we reduce the data dependency on fields to methods dependency on properties, by only using fields' getter and setter in the regular methods.

Traditionally, the getter and setter methods are defined in the *same* scope as the field is in, i.e. in the same class body (as we can see from the `class Person` in `plain_mi.cpp` of the previous example). But due to the troubles the instance fields caused us in MI, we would like to isolate them into another scope (as data implementation). Then to make other regular methods in the original class continue to work, we will add abstract property definitions to the original class (as data interface). For example, as shown in the class UML diagram on the right:

To make the distinction more clear, we intentionally colored the data interface class `Person` with lighter background color, and the data implementation class `PersonImpl` with darker background.

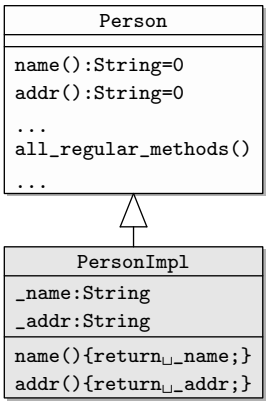


Fig. 6: decouple data interface (class Person with abstract property methods) from data implementation (class PersonImpl where the fields and property methods are actually defined)

The key point here is: the programmers have the freedom to either add new or override existing property *methods* in the derived class' data interface to achieve any application semantics, without worrying about the *data implementation*, which will be eventually defined in the implementation class. Thus remove the data dependency of the derived class' implementation on the base classes' implementation. We call this design pattern DDIFI: i.e. Decoupling Data Interface From data Implementation

The following UML shows applying DDIFI design pattern on the ResearchAssistant example:

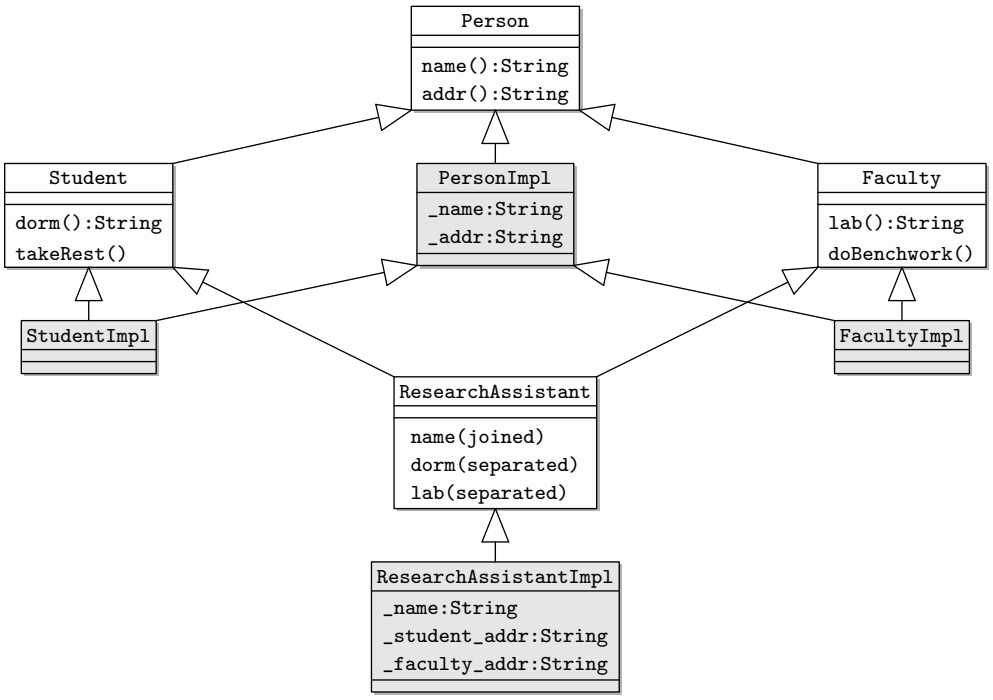


Fig. 7: DDIFI achieved the ideal semantics of *fields* `name` & `addr`: with `name` joined into one field, and `addr` separated into two fields

Please note: implementation inheritance is still an *option*, e.g. `StudentImpl` inherits `PersonImpl`, and `FacultyImpl` inherits `PersonImpl` for maximal code reuse; but it is not mandatory, e.g. `ResearchAssistantImpl` is totally *independent* of `StudentImpl`, `FacultyImpl`, and `PersonImpl`.

In fact, we can draw the above DDIFI UML classes in a 3D fashion: the interface classes are at the top, and the implementation classes are at the bottom

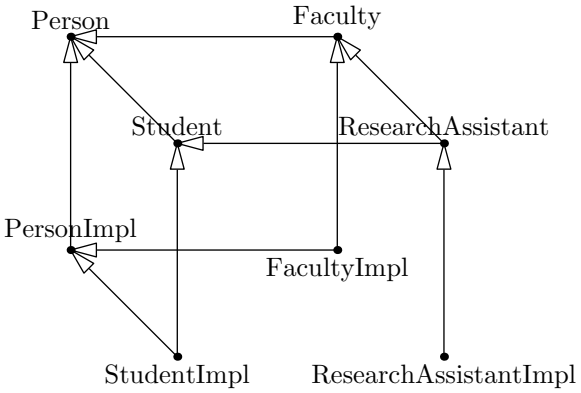


Fig. 8: DDIFI 3D UML

Please compare the diamond inheritance relationship among the interface classes on the top surface, and the *non*-diamond inheritance relationship among the implementation classes on the bottom surface (esp. the two missing edges). These two inheritance relationship are completely *independent*, and this is the key to solve the diamond problem cleanly.

This 3D UML actually shows a derived programming principle from DDIFI: i.e. Decoupling Implementation Inheritance From the Interface Inheritance (DIIFII).

13.3. DDIFI implementation in C++

In the following we will demonstrate how this data interface and implementation decoupling can solve the diamond problem in a clean way with concrete C++ code.

Listing 26: person.h

```
// define abstract virtual property, in Person's data-interface
class Person {
 public:
  virtual String name() = 0;  // C++ abstract virtual method
  virtual String addr() = 0;  // C++ abstract virtual method

  // all_public_or_protected_regular_methods() are defined in the data-interface
  // to be inherited and code-reused
};

// define fields and property method, in Person's data-implementation
class PersonImpl : Person {
 protected:
  String _name;
  String _addr;
 public:
  virtual String addr() override { return _addr; }
  virtual String name() override { return _name; }
};
```

First, split `person.h` into two classes: `Person` as data interface (with regular methods), and move fields definition into `PersonImpl` as data implementation.

Listing 27: student.h

```
class Student : public Person {
 public:
  // add new semantic assigning virtual property
  virtual String dorm() {  // give it a new exact name matching its new semantics
    return addr();         // but the implementation here can be just super's addr()
  }

  // regular methods' implementation
  void takeRest() {
    cout << name() << " takeRest in the "
         << dorm()  // MUST use the new property, not the inherited addr() whose semantics has branched!
         << endl;
  }
};


class StudentImpl : public Student, PersonImpl {
  // no new field: be memory-wise efficient, while function-wise flexible
};
```

We do the same for `student.h`, please also note:

1. We added a *new* semantic assigning virtual property dorm(), which currently just return addr(); but can be overridden in the derived classes.
2. We implemented all other regular methods in the data-interface class `Student`, which when needed can read / write (but not direct access) any instance field via the corresponding (abstract) property method.
3. Please also take notice here `Student.takeRest()` calls `dorm()` (which in turn calls `addr()`), instead of calling `addr()` directly. We will discuss this treatment of semantic branching property in the next section.
4. `StudentImpl` inherits all the data fields from `PersonImpl`, this is just for convenience; alternatively, the programmer can choose to let `StudentImpl` define its own data implementation totally *independent* of `PersonImpl`, as we will show in the following `ResearchAssistantImpl`. This is the key to solve the inherited field conflicts of the diamond problem.

Listing 28: faculty.h

```
class Faculty : public Person {
 public:
```

```
  // add new semantic assigning virtual property
  virtual String lab() {  // give it a new exact name matching its new semantics
    return addr();         // but the implementation here can be just super's addr()
  }

  // regular methods' implementation
  void doBenchwork() {
    cout << name() << " doBenchwork in the "
         << lab()  // MUST use the new property, not the inherited addr() whose semantics has branched!
         << endl;
  }
};

class FacultyImpl : public Faculty, PersonImpl {
  // no new field: be memory-wise efficient, while function-wise flexible
};
```

We do the same also for `faculty.h`, and added a new semantic assigning property lab().

Listing 29: ra.h

```
class ResearchAssistant : public Student, public Faculty {  // MI with regular-methods code reuse!
};

class ResearchAssistantImpl : public ResearchAssistant {  // only inherit from ResearchAssistant interface
 protected:
  // define 3 fields, NOTE: totally independent to those fields in PersonImpl, StudentImpl and FacultyImpl
  String _name;
  String _faculty_addr;
  String _student_addr;
 public:
  ResearchAssistantImpl() {  // the constructor
    _name = NAME;
    _faculty_addr = LAB;
    _student_addr = DORM;
  }

  // override the property methods
  virtual String name() override { return _name; }
  virtual String addr() override { return dorm(); }  // use dorm as ResearchAssistant's main addr
  virtual String dorm() override { return _student_addr; }
  virtual String  lab() override { return _faculty_addr; }
};

ResearchAssistant* makeResearchAssistant() {  // the factory method
  ResearchAssistant* ra = new ResearchAssistantImpl();
  return ra;
}
```

Finally, we define research assistant, please note:

1. The fields of `ResearchAssistantImpl`: `_name`, `_faculty_addr`, and `_student_addr` are totally *independent* of the fields in `PersonImpl`, `StudentImpl`, and `FacultyImpl`. This is what we mean: removing the data dependency of the derived class' data implementation on the base classes' data implementations
2. Now indeed each ResearchAssistant object has exactly 3 fields: 1 name, 2 addrs!
3. We added a factory method to create new ResearchAssistant objects.

Let's create a ResearchAssistant object, also assign it to `Faculty*`, `Student*` variables, and make some calls of the corresponding methods on them:

```
#include <iostream>
#include <string>
typedef std::string String;
using namespace std;

String NAME = "ResAssis";
String HOME = "home";
String DORM = "dorm";
String  LAB = "lab";

#include "person.h"
#include "student.h"
#include "faculty.h"
```

```
#include "ra.h"
#include "biora.h"

int main() {
  ResearchAssistant* ra = makeResearchAssistant();
  Faculty* f = ra;
  Student* s = ra;

  ra->doBenchwork();  // ResAssis doBenchwork in the lab
  ra->takeRest();     // ResAssis takeRest in the dorm

  f->doBenchwork();   // ResAssis doBenchwork in the lab
  s->takeRest();      // ResAssis takeRest in the dorm

  return 0;
}
```

As we can see, all the methods generate expected correct outputs.
To the best of the authors' knowledge, this design pattern that we introduced in this section to achieve multiple inheritance so cleanly has never been reported in any previous OOP literature. It is the first design pattern that cleanly solves the diamond problem in a number of mainstream industry-strength OOP programming languages[18], e.g. C++ [Str91], Java, C#, Python, Ocaml [LDF+21], D, etc, which we will show in the following sections.

13.4. Virtual property

It is very important to define the property method as *virtual*, this gives the programmers the freedom to choose the appropriate implementation of the concrete representation in the derived class. Properties can be:

- fields (data members) with memory allocation, or
- methods via computation if needed.

For example, a biology research assistant may alternate between two labs (labA, labB) every other weekday to give the micro-organism enough time to develop. We can implement `BioResearchAssistantImpl` as the following (please pay special attention to the new lab() property):

Listing 30: biora.h

```
#include "util.h"

String LAB_A = "labA";
String LAB_B = "labB";

// only inherit from ResearchAssistant, but not from any other xxxImpl class
class BioResearchAssistantImpl : public ResearchAssistant {
 protected:
  // define 2 fields, NOTE: totally independent to those fields in PersonImpl, StudentImpl and FacultyImpl
  String _name;
  String _student_addr;
 public:
  BioResearchAssistantImpl() {  // the constructor
    _name = NAME;
    _student_addr = DORM;
  }

  // override the property methods
  virtual String name() override { return _name; }
  virtual String addr() override { return dorm(); }  // use dorm as ResearchAssistant's main addr
  virtual String dorm() override { return _student_addr; }
  virtual String  lab() override {
    int weekday = get_week_day();
    return (weekday % 2) ? LAB_A : LAB_B;  // alternate between two labs
  }
};
```

[18] By the TIOBE Programming Community index `https://www.tiobe.com/tiobe-index/`

```
27
28 ResearchAssistant* makeBioResearchAssistant() {  // the factory method
29   ResearchAssistant* ra = new BioResearchAssistantImpl();
30   return ra;
31 }
```

Note: both `ResearchAssistantImpl` and `BioResearchAssistantImpl` are at the bottom point of the diamond inheritance, but their actual fields are quite different. In our approach the derived class data implementation does *not* inherit the actual fields from the base classes' data implementation, but only inherits the data interface of the base classes (i.e. the property methods, and will override them). This is the key difference from C++'s plain MI mechanism. That's why our approach is so flexible that it can achieve the intended semantics the programmers needed.
In the next section we will summarize the new programming rules to formalize our approach to achieve general MI.

14. New programming rules

Rule 10 (split data interface class and data implementation class) *To model an object foo, define two classes:*

1. *class Foo as* data interface*, which does* not *contain any field; and Foo can inherit multiple-ly from any other data-interfaces.*
2. *class FooImpl inherit from Foo, as* data implementation*, which contains fields (if any) and implement property methods.*

For example, we can see from person.h and Fig. 6: class Person and PersonImpl in the previous section.

Rule 11 (data interface class) *In the data-interface class Foo:*

1. *define or override all the (abstract) properties, and always make them virtual (to facilitate future uncoordinated MI).*
2. *implement all the (especially public and protected) regular methods, using the property methods when needed, as the default regular methods implementation.*
3. *add a static (or global) Foo factory method to create FooImpl object, which the client of Foo can call without exposing the FooImpl's implementation detail.*

Note: although Foo is called *data* interface, the regular methods are also *implemented* here, because:

- it's good engineering practice to program to (the data) interfaces, instead of using the raw fields directly
- other derived classes will inherit from Foo, (instead of FooImpl which is data implementation specific), so these regular methods can be reused to achieve the other OOP goal: maximal code reuse.

Of course, for the *private* regular methods, the programmer may choose to put them in FooImpl to hide their implementation.

Rule 12 (data implementation class) *In the data-implementation class FooImpl:*

1. *implement all the properties in the class FooImpl: a property can be either*
   (a) *via memory, define the field and implement the getter and setter, or*
   (b) *via computation, define property method*
2. *implement at most the private regular methods (or just leave them in class Foo by the* program to (the data) interfaces *principle, instead of directly accessing the raw fields).*

So, because of Rule 11 all the data-interface classes (which also contains regular method implementations) can be multiple-ly inherited by the derived interface class without causing fields conflict. And because of Rule 12 each data-implementation class can provide the property implementations exactly as the intended application semantics required.

Rule 13 (sub-classing) *To model class bar as the subclass of foo:*

1. *make Bar inherit from Foo, and override any virtual properties according to the application semantics.*
2. *make BarImpl inherit from Bar, but BarImpl can be implemented independently from FooImpl (hence no data dependency of BarImpl on FooImpl).*

Rule 14 (add and use new semantic assigning property after branching) *If class* `C` *is the semantic branching site of property* `p`*, in every data-interface class* `D` *that is immediate below* `C`*:*

1. *add a new semantic assigning virtual property* `p'` *(of course,* `p'` *and* `p` *are different names),*
2. *all other regular methods of* `D` *should choose to use* `p'` *instead of* `p` *according to the corresponding application semantics when applicable.*

The following is an example of applying this Rule 14:

- Class `Person` is the semantic branching site of property `addr`.
- In class `Student`, we added a new semantic assigning property `dorm()`;
  and `Student.takeRest()` uses property `dorm()` instead of `addr()`.
- In class `Faculty`, we added a new semantic assigning property `lab()`;
  and `Faculty.doBenchwork()` uses property `lab()` instead of `addr()`.

The reason to add new semantic assigning virtual property after branching is to facilitate fields separation, as we have shown in `ra.h`, the derived class `ResearchAssistant` implementation can override the new properties (i.e. `dorm()`, and `lab()`) differently; otherwise without adding such new properties, `ResearchAssistant` can only override the single property `addr()`, then at least one of the inherited method `takeRest()` or `doBenchwork()` will be wrong (since they can only both call `addr()` in that case).
In summary: the goal is to make fields joining or separation as flexible as possible, to allow programmers to achieve any intended semantics (in the derived data implementation class) that the application needed:

- field joining can be achieved by overriding the corresponding virtual property method of the same name from multiple base classes
- field separation can be achieved by implementing / overriding the new semantic assigning property introduced in Rule 14.

15. Java with interface default methods

Despite many modern programming languages (Java, C#) tried to avoid multiple inheritance (MI) by only using single inheritance + multiple interfaces in their initial design and releases, to remedy the restrictions due to the lack of MI, they introduced various other mechanisms in their later releases, e.g.

1. Java v8.0 added default interface methods in 2014 [Ora14]
2. C# v8.0 added default interface methods in 2019 [Mic22])

Actually, the programming rules we introduced in the previous section works perfectly well with Java's (>= v8.0) interface default methods, which now allows methods be implemented in Java interfaces. In the following, we show how the previous example can be coded in Java.

Listing 31: MI.java

```
1  interface Person {
2    public String name();  // abstract property method, to be implemented
3    public String addr();  // abstract property method, to be implemented
4    // no actual field
5  }
6
7  class PersonImpl implements Person {
8    // only define fields and property methods in data implementation class
9    String _name;
10   String _addr;
11   @Override public String name() { return _name; }
12   @Override public String addr() { return _addr; }
13 }
14
15 interface Faculty extends Person {
```

```
16    default String lab() {return addr();}  // new semantic assigning property
17
18    // regular methods
19    default void doBenchwork() {
20      System.out.println(name() + " doBenchwork in the " + lab());
21    }
22  }
23
24  class FacultyImpl extends PersonImpl implements Faculty {
25    // nothing new needed, so just extends PersonImpl
26  }
27
28  interface Student extends Person {
29    default String dorm() {return addr();}  // new semantic assigning property
30
31    // regular methods
32    default void takeRest() {
33      System.out.println(name() + " takeRest in the " + dorm());
34    }
35  }
36
37  class StudentImpl extends PersonImpl implements Student {
38    // nothing new needed, so just extends PersonImpl
39  }
40
41  interface ResearchAssistant extends Student, Faculty {
42    // factory method
43    static ResearchAssistant make() {
44      ResearchAssistant ra = new ResearchAssistantImpl();
45      return ra;
46    }
47  }
48
49  class ResearchAssistantImpl implements ResearchAssistant {
50    // define 3 fields, NOTE: totally independent to those fields in PersonImpl, StudentImpl and FacultyImpl
51    String _name;
52    String _faculty_addr;
53    String _student_addr;
54
55    ResearchAssistantImpl() {   // constructor
56      _name = "ResAssis";
57      _faculty_addr = "lab";
58      _student_addr = "dorm";
59    }
60
61    // property methods
62    @Override public String name() { return _name; }
63    @Override public String addr() { return dorm(); }  // use dorm as addr
64    @Override public String dorm() { return _student_addr; }
65    @Override public String  lab() { return _faculty_addr; }
66  }
67
68
69  public class MI {
70    public static void main(String[] args) {
71      ResearchAssistant ra = ResearchAssistant.make();
72      Faculty f = ra;
73      Student s = ra;
74
75      ra.doBenchwork();  // ResAssis doBenchwork in the lab
76      ra.takeRest();     // ResAssis takeRest in the dorm
77
78      f.doBenchwork();   // ResAssis doBenchwork in the lab
79      s.takeRest();      // ResAssis takeRest in the dorm
80    }
81  }
```

16. Related works

16.1. Compare our method with MI via composition

With the technique introduced in this paper, there are some boilerplate property implementation code needed for each virtual property. However for any non-trivial program, typically the number of regular class methods is far more than the number of fields. So our approach is better than MI via composition in terms of the needed supporting boilerplate code. More importantly, with MI via composition the programmers still need to solve the field joining problem. While with our approach, the fields joining or separation problems are solved perfectly by overriding the corresponding virtual property methods to read / write the same (e.g. `_name`) or different (e.g. `_faculty_addr`, or `_student_addr`) fields in the data implementation class. Additionally, our boilerplate property implementation code can be auto-generated by a pre-processor.

16.2. Compare our method with mixins / traits

Firstly, mixins / traits' drawbacks have been discussed in Section 12.6, DDIFI does not have any of those drawbacks. Furthermore, the programmers need to learn the new language concept and rules associated with mixins or traits; while in our method, DDIFI only use the existing language mechanisms, which the programmers have already mastered.

16.3. *Virtual* fields (properties) are all you need to achieve clean MI

In many OOP languages, e.g. C#, JavaScript, Python, Scala, Swift, D, and Lua etc., properties are special class member methods that provide controlled access to an object's data by getters and setters, which encapsulate data with logic for validation, computation, or transformation. Our method make heavy use of *virtual* fields (properties), which is the key to achieve clean MI.

17. Discussions and future works

17.1. The ResearchAssistant inheritance problem challenge

In Section 11 we introduced the ResearchAssistant inheritance Problem 1 as a working example of our method. We believe if one can solve this problem, then one can solve *all* the problems that can arise in multiple inheritance. We would like to propose the following challenges to the whole programming language research community:

1. Design another alternative cleaner solution (than ours) to the diamond problem of `<Person, Student, Faculty, ResearchAssistant>`.
2. Devise another multiple inheritance problem (be it diamond problem or not), which *cannot* be solved by using the method in this paper.

17.2. Advantages and disadvantages of DDIFI

We would like to summarize the advantages and disadvantages of DDIFI as the following:
Advantages:

1. Clean: completely solved the diamond problem cleanly.
2. General: works in not only in native multiple inheritance languages like C++, Python, OCaml, Lisp, and Eiffel etc., but also works in single inheritance languages like Java, C#, D, Scala etc. (I.e. DDIFI achieved clean multiple inheritance in these native single inheritance languages).

Disadvantages:

1. Each class now split into two classes: one as data-interface (which also contains regular methods implementation), and the other as data-implementation.
   However we think:
   (a) These splits *only* happen in parts of the class hierarchy within the whole software system where diamond (multiple) inheritance is needed.
   (b) The concept of *program to interface* is a good practice in almost any serious software project already, which is well-understood by the developers.
2. The regular methods must access fields using property methods which will incur lots of virtual function call cost in performance.
   However we think:
   (a) Again, these extra virtual dispatch (compared to a system built without DDIFI) *only* happen in parts of the class hierarchy within the whole software system where diamond (multiple) inheritance is needed.
   (b) "Premature optimization is the root of all evil" [19]: making the multiple inheritance logic correct is more important than micro optimizations.
   (c) Virtual methods is the corner-stone of OOP (since its start in 1960s), it is heavily optimized by modern compilers already. "If one does not want use virtual functions, probably s/he should not use OOP in the first place."[20]
   (d) Also we can always use local temporary variables to reduce the number of virtual property method calls needed in any regular method.

Finally, we want to emphasize that DDIFI operates on a pay-as-you-go basis: programmers are not obliged to use it in every class of their software system. Instead, they can employ DDIFI selectively, utilizing it only when the native OOP language's inheritance mechanism falls short in achieving the desired application semantics.

17.3. Case study: empirical evaluation

We have incorporated this design pattern into our proprietary software system in the D programming language, which consists of 213 classes. The system has only two instances of diamond inheritance, implemented using DDIFI, with a total of 26 abstract virtual fields. We conducted a comparison of both the resulting binary's code size and runtime efficiency with the previous system, which is constructed using multiple inheritance via composition, and found the difference to be negligible. However, the structural clarity and logic of the new system with DDIFI far surpasses that of its predecessor.
Another example is what we have discussed in Section 6.1 of Part 1, there are 7 diamond inheritances in Eiffel's EiffelVision 2 GUI library (1194 classes in total), all these 7 diamond inheritances can be re-implemented using the DDIFI design pattern.

17.4. Cfront 2.0 challenge

In retrospect, when C++ first introduced multiple inheritance in its version 2.0 (then called Cfront Release 2.0 [Str89b]), all the native language mechanisms to achieve DDIFI are available. Since the original source code [Str89c] only works on the old system back in 1980's, we would like to propose the following challenge: if we can port the source code onto a modern Unix/Linux system, then we can try DDIFI in C++ 2.0!

[19] `https://hans.gerwitz.com/2004/08/12/premature-optimization-is-the-root-of-all-evil.html`
[20] by Walter Bright (designer of the D programming language [BAP20]), private communication on the DDIFI design pattern presented in this paper.

17.5. Programming paradigms evolution: procedural, OOP, DDIFI

In the following table, we compare three different ways of programming using C++ side by side:

1. Procedural programming, where data and functions are separate.
2. Object oriented programming (OOP), where data and methods are bundled together in one unit (class).
3. OOP with Decoupling Data Interface From data Implementation (DDIFI), where each class is split into a *data*-interface class and a *data*-implementation class.

In summary, with DDIFI:

- all the class behaviors (method bodies) are defined by using virtual properties in the data-interface classes, which allows maximum code reuse via inheritance; while
- only the states (fields) are defined in the data-implementation class, which can be tailored by the programmers to fit the application semantics on an individual bases.

Procedural programming	Object oriented programming	OOP with DDIFI

`struct Person {`
`  String name;`
`  String addr;`
`};`
 	
`void a_function(Person* p) {`
`  print(p->addr);`
`}`
	`class Person {`
	`  String name;`
	`  String addr;`
	
	`public:`
	`  void a_regular_method() {`
	`  print(this->addr);`
	`  }`
	`};`
		`class Person {`
		`public:`
		`  virtual String name() = 0;`
		`  virtual String addr() = 0;`
		
		`  void a_regular_method() {`
		`  print(this->addr());`
		`  }`
		`};`
		
		`class PersonImpl : Person {`
		`private:`
		`  String _name;`
		`  String _addr;`
		
		`public:`
		`  virtual String name() {`
		`  return _name;`
		`  }`
		
		`  virtual String addr() {`
		`  return _addr;`
		`  }`
		`};`

17.6. Integrate into existing language compilers

The new programming rules we introduced in Section 14 can also be added to existing OOP language compilers (e.g. as automatic code transformers), maybe with a new command-line option, to help the programmers to achieve clean multiple inheritance in these languages.
Also pre-processors can be developed for each of the OOP language we discussed, to auto-generate the boilerplate code for the DDIFI fields implementation.
It is our hope that the success of DDIFI will guide the future OOP languages design and implementation.

Acknowledgments

The author wishes to thank Prof. Tony Hoare for introducing me to the study of OOP, and Prof. Bertrand Meyer for the discussions on Eiffel (despite the semantic issue discovered in Part 1, we believe Eiffel remains an inspiring OOP language). The author also thanks Eric Bezault for assisting in the development of the Eiffel field renaming detection tool, and Walter Bright for his comments on the DDIFI design pattern.

A. Supplement

The supplementary DDIFI.tgz contains the full implementation source code of this design pattern DDIFI in the following 9 languages: C++, Java, C#, Python, OCaml, Scala, Eiffel, Lisp (CLOS [Ste90]) and D etc.

A.1. Our approach demo in C#

C#'s (>= v8.0) default interface methods are essentially the same as Java's [Mic22]. The following is the equivalent C# program of our Java example:

Listing 32: MI in C#

```
using System;

interface Person {
  public string name();  // abstract property method, to be implemented
  public string addr();  // abstract property method, to be implemented
  // no actual field
}

class PersonImpl : Person {
  // only define fields and property methods in data implementation class
  string _name = null;
  string _addr = null;
  public string name() { return _name; }
  public string addr() { return _addr; }
}

interface Faculty : Person {
  string lab() {return addr();}  // new semantic assigning property

  // regular methods
  void doBenchwork() {
    Console.WriteLine(name() + " doBenchwork in the " + lab());
  }
}

class FacultyImpl : PersonImpl, Faculty {
  // nothing new needed, so just extends PersonImpl
}

interface Student : Person {
  string dorm() {return addr();}  // new semantic assigning property

  // regular methods
  void takeRest() {
    Console.WriteLine(name() + " takeRest in the " + dorm());
  }
}

class StudentImpl : PersonImpl, Student {
  // nothing new needed, so just extends PersonImpl
}

interface ResearchAssistant : Student, Faculty {
  // factory method
  public static ResearchAssistant make() {
    ResearchAssistant ra = new ResearchAssistantImpl();
    return ra;
  }
}

class ResearchAssistantImpl : ResearchAssistant {
  // define 3 fields, NOTE: totally independent to those fields in PersonImpl, StudentImpl and FacultyImpl
  string _name;
  string _faculty_addr;
  string _student_addr;

  public ResearchAssistantImpl() {   // constructor
    _name = "ResAssis";
    _faculty_addr = "lab";
    _student_addr = "dorm";
```

```
  }

  // property methods
  public string name() { return _name; }
  public string addr() { return dorm(); }  // use dorm as addr
  public string dorm() { return _student_addr; }
  public string  lab() { return _faculty_addr; }
}


public class MI {
  public static void Main(string[] args) {
    ResearchAssistant ra = ResearchAssistant.make();
    Faculty f = ra;
    Student s = ra;

    ra.doBenchwork();  // ResAssis doBenchwork in the lab
    ra.takeRest();     // ResAssis takeRest in the dorm

    f.doBenchwork();   // ResAssis doBenchwork in the lab
    s.takeRest();      // ResAssis takeRest in the dorm
  }
}
```

A.2. Our approach demo in Python

The following is the equivalent Python program of our Java example:

Listing 33: MI.py

```
import abc

class Person:
  @abc.abstractmethod
  def name(self):  # abstract property method, to be implemented
    pass

  @abc.abstractmethod
  def addr(self):  # abstract property method, to be implemented
    pass

  # no actual field


class PersonImpl(Person):
  # only define fields and property methods in data implementation class
  def __init__(self):
    self._name = "name";
    self._addr = "addr";

  def name(self): return self._name;
  def addr(self): return self._addr;


class Faculty(Person):
  def lab(self): return self.addr();  # new semantic assigning property

  # regular methods
  def doBenchwork(self):
    print(self.name() + " doBenchwork in the " + self.lab());


class FacultyImpl(PersonImpl, Faculty):
  # nothing new needed, so just: PersonImpl
  pass


class Student(Person):
  def dorm(self): return self.addr();  # new semantic assigning property

  # regular methods
  def takeRest(self):
```

```
43      print(self.name() + " takeRest in the " + self.dorm());
44
45
46  class StudentImpl(PersonImpl, Student):
47    # nothing new needed, so just: PersonImpl
48    pass
49
50
51  class ResearchAssistant(Student, Faculty):
52    # factory method
53    @staticmethod
54    def make():
55      ra = ResearchAssistantImpl();
56      return ra;
57
58
59  class ResearchAssistantImpl(ResearchAssistant):
60    # define 3 fields, NOTE: totally independent to those fields in PersonImpl, StudentImpl and FacultyImpl
61    def __init__(self):   # constructor
62      self._name = "ResAssis";
63      self._faculty_addr = "lab";
64      self._student_addr = "dorm";
65
66    # property methods
67    def name(self): return self._name;
68    def addr(self): return self.dorm();  # use dorm as addr
69    def dorm(self): return self._student_addr;
70    def  lab(self): return self._faculty_addr;
71
72
73  def main():
74    ra:ResearchAssistant = ResearchAssistant.make();
75    f:Faculty = ra;
76    s:Student = ra;
77
78    ra.doBenchwork();  # ResAssis doBenchwork in the lab
79    ra.takeRest();     # ResAssis takeRest in the dorm
80
81    f.doBenchwork();   # ResAssis doBenchwork in the lab
82    s.takeRest();      # ResAssis takeRest in the dorm
83
84
85  if __name__ == '__main__':
86    main()
```

References

[BAP20] Walter Bright, Andrei Alexandrescu, and Michael Parker. Origins of the d programming language. *Proceedings of the ACM on Programming Languages, Volume 4, Issue HOPL*, pages 1–38, June 2020.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, 1990.

[BCH+96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. *OOPSLA '96 Conference Proceedings. ACM Press.*, pages 69–82, 1996.

[BI82] Alan H. Borning and Daniel H. H. Ingalls. Multiple inheritance in smalltalk-80. *Proceedings of the Second AAAI Conference on Artificial Intelligence (AAAI'82). AAAI Press*, pages 234–237, 1982.

[BMCP18] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. Cfixx: Object type integrity for c++ virtual dispatch. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[BW00] Martin Büchi and Wolfgang Weck. Generic wrappers. In *ECOOP 2000 Object-Oriented Programming: 14th European Conference Sophia Antipolis and Cannes, France, June 12–16, 2000 Proceedings 14*, pages 201–225. Springer, 2000.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Inf. Comput. 76.2/3*, pages 138–164, 1988.

[CG90] Bernard Carré and Jean-Marc Geib. The point of view notion for multiple inheritance. *ACM Sigplan Notices 25.10*, pages 312–321, 1990.

[CP93] Adriana B. Compagnoni and Benjamin C. Pierce. Multiple inheritance via intersection types. *Computing Science Institute, Department of Informatics, Faculty of Mathematics and Informatics, [Katholieke Universiteit Nijmegen]*, 1993.

[CUCH82] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in self. *LISP and Symbolic Computation 4*, pages 207–222, 1982.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

[ECM06] ECMA. *Eiffel: Analysis, Design and Programming Language*. ECMA International, 2006.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language: Everything You Need to Know*. "O'Reilly Media, Inc.", 2008.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java language specification*. Addison-Wesley Professional, 2000.

[Goo22] Google. Google c++ style guide, Jul 5, 2022.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[Kni99] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP99 Object-Oriented Programming: 13th European Conference Lisbon, Portugal, June 14–18, 1999 Proceedings 13*, pages 351–366. Springer, 1999.

[Knu88] Jørgen Lindskov Knudsen. Name collision in multiple classification hierarchies. In *European Conference on Object-Oriented Programming*, pages 93–109. Springer, 1988.

[LDF+21] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.13: Documentation and user's manual*. PhD thesis, Inria, 2021.

[Loc15] Josh Lockhart. *Modern PHP: New features and good practices*. " O'Reilly Media, Inc.", 2015.

[MA09] Donna Malayeri and Jonathan Aldrich. Cz: multiple inheritance without diamonds. *ACM SIGPLAN Notices*, 44(10):21–40, 2009.

[Mey93] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1993.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction, 2nd Ed.* Prentice Hall, 1997.

[Mic22] Microsoft. Default interface methods, 08/12/2022.

[Ngo21] Macaire Ngomo. Multiple inheritance mechanisms in logic objects approach based on a multiple specialisation of objects. *International Journal of Computer Trends and Technology, vol. 69, no. 10, pp. 1-11, 2021*, 2021.

[Ode10] Martin Odersky. The scala language specification, version 2.9. *EPFL (May 2011)*, 2010.

[Ora14] Oracle. Default methods, 2014.

[Ott18] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

[Sak89] M Sakkinen. Disciplined inheritance. In *ECOOP 89*, pages 39–56. Cambridge University Press, 1989.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. *Lecture Notes in Computer Science 2743*, pages 248–274, 2003.

[Sny87] Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming, Series in Computer Systems*, pages 165–188. The MIT Press, 1987.

[SP20] Marek Suchánek and Robert Pergl. Evolvability analysis of multiple inheritance and method resolution order in python. *PATTERNS 2020 : The Twelfth International Conference on Pervasive Patterns and Applications*, 8:11, 2020.

[Ste90] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

[Str89a] Bjarne Stroustrup. Multiple inheritance for c++. *Computing Systems*, 2(4):367–395, 1989.

[Str89b] Bjarne Stroustrup. C++ historical sources archive, June, 1989.

[Str89c] Bjarne Stroustrup. Cfront release 2.0, June, 1989.

[Str91] Bjarne Stroustrup. *The C++ Programming Language (Second Edition)*. Addison-Wesley, 1991.

[Str94] B Stroustrup. The design and evolution of c++. addison weasley. *Reading*, 1994.

[Str03] Bjarne Stroustrup. The c++ standard: Incorporating technical corrigendum no. 1, 2003.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Comput. Surv. 28, 3 (Sept. 1996)*, pages 438–479, 1996.

[TDP+20] Pablo Tesone, Stéphane Ducasse, Guillermo Polito, Luc Fabresse, and Noury Bouraqadi. A new modular implementation for stateful traits. *Science of Computer Programming*, 195, 2020.

[TJJV04] Eddy Truyen, Wouter Joosen, Bo Nørregaard Jørgensen, and Pierre Verbaeten. A generalization and solution to the common ancestor dilemma problem in delegation-based object systems. In *Dynamic aspects workshop (daw04)*, volume 6, 2004.

[vLM96] Marc van Limberghen and Tom Mens. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Syst. 3.1*, pages 1–30, 1996.

[vR10] Guido van Rossum. The history of python: Method resolution order, June 23, 2010.

[VRDJ14] Guido Van Rossum and Fred L Drake Jr. The python language reference. *Python Software Foundation: Wilmington, DE, USA*, 2014.

[WNST06] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in c++. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 345–362, 2006.