A new design pattern DDIFI: Decoupling *Data* Interface From *data* Implementation as a clean and general solution to multiple inheritance[1]

YUQIAN ZHOU,, USA

Traditionally in class based OOP languages, *both the fields and methods* from the super-classes are inherited by the sub-classes. However this may cause some serious problems in multiple inheritance, e.g. most notably the diamond problem. In this paper, we propose to *stop inheriting data fields* as a clean and general solution to such problems. We first present a design pattern to cleanly achieve multiple inheritance in C++, which can handle class fields of the diamond problem exactly according to the programmers' intended application semantics. It gives programmers flexibility when dealing with the diamond problem for instance variables: each instance variable can be *configured* either as one joined copy or as multiple independent copies in the bottom class. The key ideas are: 1) decouple *data* interface from *data* implementation; 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 also applicable to any OOP languages that natively support multiple inheritance (e.g. C++, Python, Eiffel, etc.), or single inheritance languages that support default interface methods (e.g. Java, C# etc.).

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*.

Additional Key Words and Phrases: multiple inheritance, diamond problem, program to interfaces, virtual property, data interface, data implementation, semantic branching site, reusability, modularity

ACM Reference Format:
YuQian Zhou. 2023. A new design pattern DDIFI: Decoupling *Data* Interface From *data* Implementation as a clean and general solution to multiple inheritance. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2023), 15 pages.

1 MOTIVATION: THE DIAMOND PROBLEM

The most well known problem in multiple inheritance (MI) is the diamond problem, let's quote from wikipedia[1]:

> 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?

Actually in the real world engineering practice, for any *method*'s ambiguity e.g. foo(), it is *relatively easy* to resolve *by the programmers*:

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

[1]The work reported in this paper is patent pending.
[1]https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

Author's address: YuQian Zhou, Mercer Island, WA, 98040, USA, zhou@joort.com.

2023. 2475-1421/2023/1-ART1 $15.00
https://doi.org/

The *more difficult* problem is how to handle the *data members (i.e. fields)* inherited from A: shall D have one joined copy or two separate copies of A's fields (or mixed fields with some are joined, and others separated)? For example, in C++ [Stroustrup 1991], the former is called virtual inheritance, and the latter is default (regular) inheritance. But C++ does not completely solve this problem, for example let's build an object model for Person, Student, Faculty, and ResearchAssistant in a university:

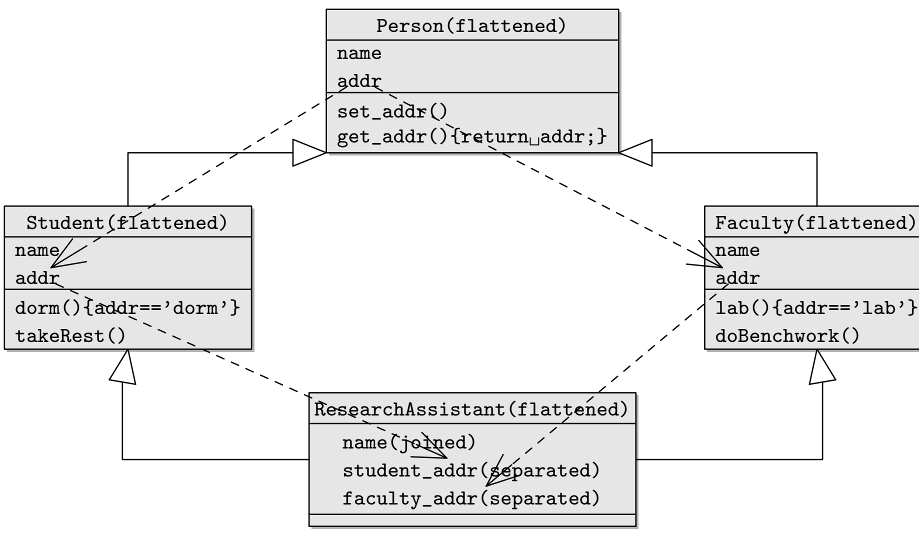

Fig. 1. the diamond problem: the ideal semantics of *fields* `name` & `addr`, which is not achievable in C++'s plain MI mechanism: with `name` joined into one field, and `addr` separated into two fields

The intended application semantics is that a ResearchAssistant should have only 1 name field, but 2 address fields: one "dorm" as Student to takeRest(), and one "lab" as Faculty to doBenchwork(); so in total 3 fields. However, 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

As is shown in the following C++ code:

Listing 1. plain_mi.cpp

```
#include <stdio.h>
typedef char* String;

#define VIRTUAL // virtual  // no matter we use virtual inheritance or not, it's problematic

class Person {
 protected:
  String _name;  // need to be joined into one single field in ResearchAssistant
  String _addr;  // need to be separated into two addresses in ResearchAssistant
 public:
  virtual String name() {return _name;}
  virtual String addr() {return _addr;}
};

class Student : public VIRTUAL Person {
 public:
  virtual String dorm() {return _addr;}  // assign dorm semantics to _addr
  void takeRest() {
    printf("%s takeRest in the %s\n", name(), dorm());
```

```
  }
};

class Faculty : public VIRTUAL Person {
 public:
  virtual String lab() {return _addr;}  // assign lab semantics to _addr
  void doBenchwork() {
    printf("%s doBenchwork in the %s\n", name(), lab());
  }
};

class ResearchAssistant : public VIRTUAL Student, public VIRTUAL Faculty {
};


int main() {
  printf("sizeof(Person)  = %ld\n", sizeof(Person));
  printf("sizeof(Student) = %ld\n", sizeof(Student));
  printf("sizeof(Faculty) = %ld\n", sizeof(Faculty));
  printf("sizeof(ResearchAssistant) = %ld\n", sizeof(ResearchAssistant));
}
```

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 [Google 2022] (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.

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

- in Python all the inherited fields are joined by name (a Python object's fields are keys of a internal dictionary)
- while Java / C# get rid of multiple inheritance in favor of the simple single inheritance, and advise programmers to use composition to simulate multiple inheritance when needed.

However, in this paper we will show we have designed a new method which can cleanly achieve multiple inheritance according to the programmers intended semantics. In Section 2 we will demo our method in C++ using the previous example step by step; in Section 3 we will summarize and present new programming rules that make our method also work in some other OOP languages; in Section 4 we will demo our method in Java with the same example using these rules. in Section 5 we will compare our method with MI via composition, and other approaches like mixins / traits. Finally, in the Appendix, we will show our method in Python and C#.

2 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.

2.1 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?

The key inspiring question: since class fields are the troublemakers for MI, can we just remove them from the inheritance relation? or delay their implementation to the last point?

2.2 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 class fields:

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

Definition 1 (getter and setter method)**.** In OOP we have

- The getter method returns the value of a class field.
- The setter method takes a parameter and assigns it to a class 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.1 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 2 (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 are more manipulatable than the raw fields, we can 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 (as we can see from the `class Person` in `plain_mi.cpp` of the previous example). But due to the troubles the class 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:

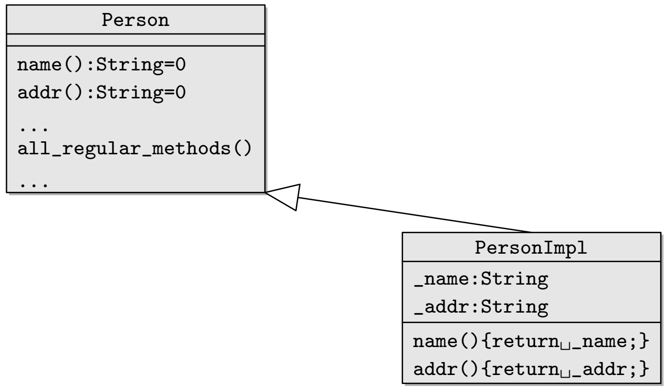

Fig. 2. decouple data interface (class Person with abstract property methods) from data implementation (class PersonImpl where the fields and property methods are actually defined)

The key point here is that: 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.

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

Listing 2. person.h

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

  // all_regular_methods()
};


class PersonImpl : Person {  // define fields and property method, as Person's (data) implementation
 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 3. student.h

```
class Student : public Person {
 public:
  virtual String dorm() {return addr();}  // new semantic assigning property

  // regular methods' implementation
  void takeRest() {
    printf("%s takeRest in the %s\n", name(), dorm());
  }
};


class StudentImpl : public Student, PersonImpl {
  // no new field
};
```

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 class 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 4. faculty.h

```
class Faculty : public Person {
 public:
  virtual String lab() {return addr();}  // new semantic assigning property

  // regular methods' implementation
  void doBenchwork() {
    printf("%s doBenchwork in the %s\n", name(), lab());
  }
};

class FacultyImpl : public Faculty, PersonImpl {
  // no new field
};
```

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

Listing 5. ra.h

```
class ResearchAssistant : public Student, public Faculty {  // inherit from both interface
};

class ResearchAssistantImpl : public ResearchAssistant {  // only inherit from ResearchAssistant interface
 protected:
  // define three 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 <stdio.h>
typedef char* String;

char NAME[] = "ResAssis";
char HOME[] = "home";
char DORM[] = "dorm";
char  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.

2.3 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 6. biora.h

```
#include "util.h"

char  LAB_A[] = "labA";
char  LAB_B[] = "labB";

class BioResearchAssistantImpl : public ResearchAssistant {  // only inherit from ResearchAssistant interface
 protected:
  // define two 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
  }
};

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

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 it can achieve the intended the semantics the programmers needed.

In the next section we will summarize the new programming rules to formalize our approach to achieve general MI.

3 NEW PROGRAMMING RULES

Rule 1 (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 multiply 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. 2: class Person and PersonImpl in the previous section.

Rule 2 (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 unplanned 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 3 (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 2 all the data-interface classes (which also contains regular method implementations) can be multiply inherited by the derived interface class without causing fields conflict. And because of Rule 3 each data-implementation class can provide the property implementations exactly as the intended application semantics required.

Rule 4 (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 5 (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 5:

- 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 5.

4 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 [Oracle [n. d.]]
(2) C# v8.0 added default interface methods in 2019 [Microsoft 2022])

Actually, the programming rules we introduced in the previous section works perfect 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 7. MI.java

```java
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 implements Person {
  // only define fields and property methods in data implementation class
  String _name;
  String _addr;
  @Override public String name() { return _name; }
  @Override public String addr() { return _addr; }
}

interface Faculty extends Person {
  default String lab() {return addr();}  // new semantic assigning property

  // regular methods
  default void doBenchwork() {
    System.out.println(name() + " doBenchwork in the " + lab());
  }
}

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

interface Student extends Person {
  default String dorm() {return addr();}  // new semantic assigning property

  // regular methods
  default void takeRest() {
    System.out.println(name() + " takeRest in the " + dorm());
  }
}

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

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

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

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

  // property methods
  @Override public String name() { return _name; }
  @Override public String addr() { return dorm(); }  // use dorm as addr
  @Override public String dorm() { return _student_addr; }
  @Override 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
  }
}
```

5 DISCUSSION

5.1 Compare our method with MI via composition

For OOP languages which do not direct support MI, it is usually suggested to simulate MI via composition, with *manual* method forwarding, which is very tedious sometimes. 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.

5.2 Compare our method with mixins / traits

In some other single inheritance OOP languages, various forms of mixins are introduced to remedy the lack of MI. Basically a mixin is a named compilation unit which contains fields and methods to be *included* rather than *inherited* by the client class to avoid the inheritance relationship, e.g.:

- Mixins [Bracha and Cook 1990] in Dart, D, Ruby, etc.
- Traits [Schärli et al. 2003] in Scala, PHP, etc.

However, the problems with mixins are:

(1) There is no clean and flexible way to resolve field (of the same name) conflicts included from multiple different mixins, as our method has achieved.
(2) Furthermore, an object of the type of the including class cannot be cast to, and be used as the named mixin type, 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.

5.3 Integrate into existing language compilers

The new programming rules we introduced in Section 3 can also be added to existing OOP language compilers (e.g. C++/Python/Java/C#/Eiffel etc.), maybe with a new command-line option, to help the programmers to achieve clean MI in these languages.

5.4 Programming paradigms: 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 an interface class and an implementation class.

<table>
<tr><th>Procedural programming</th><th>Object oriented programming</th><th>OOP with DDIFI</th></tr>
<tr>
<td><pre>
struct Person {
  String name;
  String addr;
};

void a_function(Person* p) {
  print(p->addr);
}
</pre></td>
<td><pre>
class Person {
  String name;
  String addr;

 public:
  void a_regular_method() {
    print(this->addr);
  }
};
</pre></td>
<td><pre>
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;
  }
};
</pre></td>
</tr>
</table>

ACKNOWLEDGMENTS

A APPENDIX

The source code of this paper is available at https://github.com/joortcom/DDIFI.

A.1 Our approach demo in C#

C#'s (>= v8.0) default interface methods are essentially the same as Java's [Microsoft 2022]. The following is the equivalent C# program of our Java example:

Listing 8. 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 three 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 9. MI.py

```
import abc

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

```python
    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):
    print(self.name() + " takeRest in the " + self.dorm());


class StudentImpl(PersonImpl, Student):
  # nothing new needed, so just: PersonImpl
  pass


class ResearchAssistant(Student, Faculty):
  # factory method
  @staticmethod
  def make():
    ra = ResearchAssistantImpl();
    return ra;


class ResearchAssistantImpl(ResearchAssistant):
  # define three fields: NOTE: totally independent to those fields in PersonImpl, StudentImpl, and FacultyImpl
  def __init__(self):   # constructor
    self._name = "ResAssis";
    self._faculty_addr = "lab";
    self._student_addr = "dorm";

  # property methods
  def name(self): return self._name;
  def addr(self): return self.dorm();  # use dorm as addr
  def dorm(self): return self._student_addr;
  def  lab(self): return self._faculty_addr;


def main():
  ra:ResearchAssistant = ResearchAssistant.make();
  f:Faculty = ra;
  s:Student = 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


if __name__ == '__main__':
  main()
```

REFERENCES

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

Google. Jul 5, 2022. *Google C++ Style Guide*. https://google.github.io/styleguide/cppguide.html

Microsoft. 08/12/2022. *Default interface methods*. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods

Oracle. [n. d.]. *Default Methods*. https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behaviour. *Lecture Notes in Computer Science 2743* (2003), 248–274.

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