The Diamond Problem **Solved**!

A new design pattern **DDIFI** (Decoupling ***Data Interface*** From data Implementation)
as a **clean** and **general** solution to multiple inheritance: by using ***virtual properties***

- **Clean**: solve the diamond (*name clashing*) problem very cleanly
- **General**: works in C++ / Python / Java / C# / Ocaml / Lisp / Scala / Eiffel / D, etc. …
  - YES: with DDIFI, we can achieve clean multiple inheritance in Java! — the so-called single inheritance language!

NOTE the key point: it's ***DATA*** interface, not (just) *method* interface.

YuQian Zhou (zhou@joort.com), June 24, 2023

Disclosure: This work is patent pending.

Talk outline

- Intro: about me
- Review how plain multiple inheritance currently work in C++
    - The diamond problem, why it is hard:
        1. C++ memory model is messy (a very brief discussion)
        2. Semantic branching
    - Current less-ideal solution: by composition
- My design pattern DDIFI, which solved the diamond problem cleanly
    - Stop inheriting data fields; instead, use virtual property to define regular methods
    - A new concept: semantic branching site
- Walk-thru DDIFI in C++
- General programming rules / guidelines of DDIFI
- Quick walk-thru DDIFI in Java!
- Q & A

Disclosure: This work is patent pending.

About me, my experience with languages

- Startup founder
    - Always looking for better developing tools,
    - including better programming languages
    - C++, D, Rust, Dart, Python, Java, Lisp, Go
- Google engineering
    - 3 main lang: C++, Java & Python
    - Invited Walter Bright to Google HQ in 2005 to give a talk about D pre-v1.0
        - EVP then Alan said the new language need to be mature & stable
        - … Google later developed Go (2009) … by Robert Griesemer, Rob Pike, Ken Thompson
- D.Phil, Oxford Univ., thesis advisor: Prof. Tony Hoare
    - Process algebra, CSP (later Go is based on)/ OOP (Eiffel)

Disclosure: This work is patent pending.

Overview: Multiple Inheritance (MI)

Historically:

- MI is considered complex (e.g. since C++, v2.0 1989), caused lots of headache
  - E.g. Google C++ coding style strongly advised against it.
- Most notably: **the diamond problem**
- Such that, later languages Java(1995)/C#(2000)/D(2001)/…: only allow single inheritance + multiple interfaces (i.e. only method prototype declaration without implementation code).

BUT MI is still very useful for code reuse: programmers do want to ***reuse the implementation code*** (not just the method interface), so people invented other mechanisms to make remedy, e.g:

- Trait: Scala, PHP, etc.
- Mixin: Ruby, Dart, D (multiple <interface + `alias this` + mixin template>, MI creeps in already)
- However, there is no clean solution for the name-clashes, esp for data fields.

Not anymore: with DDIFI

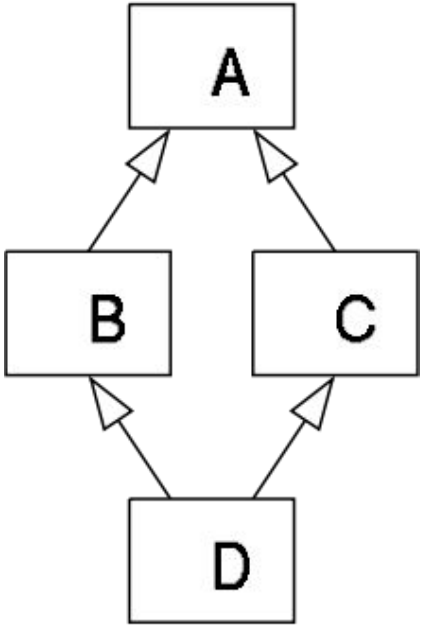
- **Clean**: solve the diamond (name clashing) problem very cleanly
- **General**: works in C++ / Python / Java / C# / Ocaml / Lisp / Scala / Eiffel / D, etc. …

Disclosure: This work is patent pending.

Motivation: the diamond problem

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?

From: https://en.wikipedia.org/wiki/Multiple_inheritance



Actually, this is application semantics, no compiler rule can help the programmers to choose *auto-magically*.

For the programmers, the answer is right in the problem description:

- Just **override** it!, or
- Use **fully quantified** method names, e.g. A.foo(), B.foo(), or C.foo().

Conclusion: for method name clash resolution, it's very easy.

The more difficult problem is: fields resolution. Let's see a concrete example:

Disclosure: This work is patent pending.

The more difficult problem: fields resolution

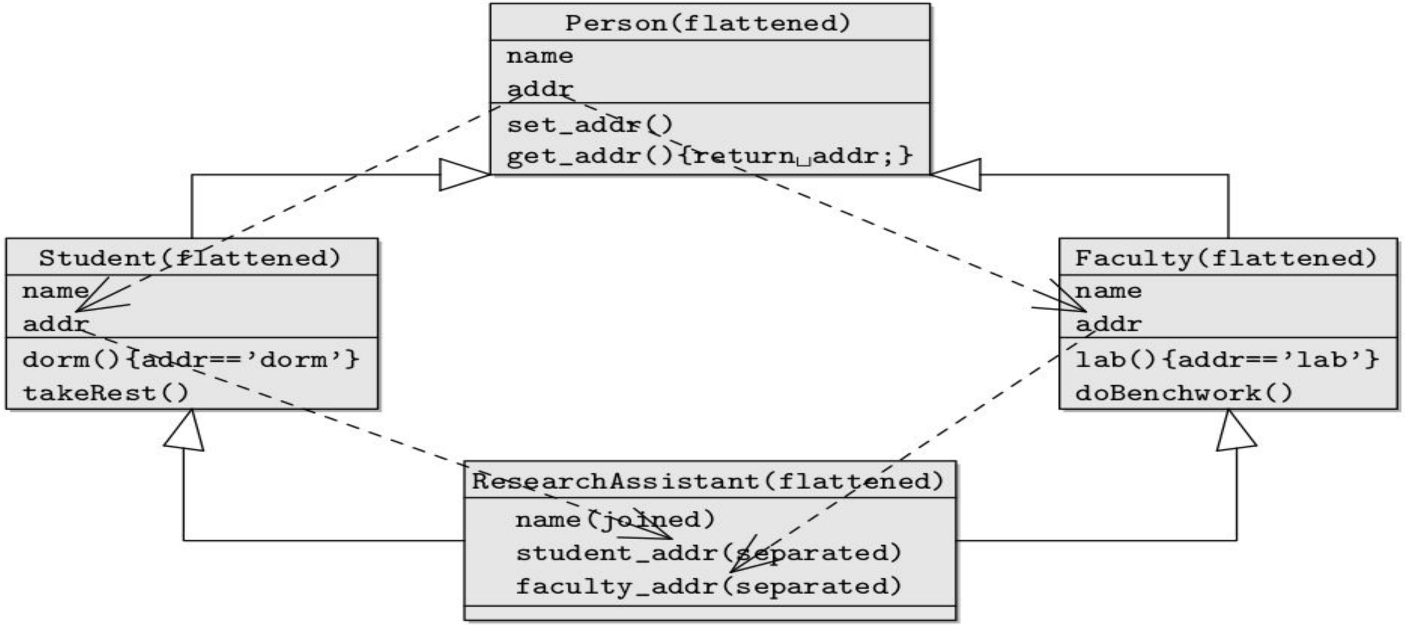


Fig. 1. the diamond problem: the ideal semantics of *fields* name & addr, which is not achievable in C++'s plain MI mechanism: with name joined into one field, and addr separated into two fields

Disclosure: This work is patent pending.

The more difficult problem: fields resolution

For fields in A, that are inherited by B and C, and then in D. If the application semantics requires:

- Some of the fields (`name`) be joined, while
- Other fields (`addr`) be separated, how to achieve this?

How to handle *both* scenarios?

  - Separation is relatively easy, e.g. use fully quantified names
  - but how to join fields, e.g. `Student.name` & `Faculty.name` into `ResearchAssistant.name`?

In the remaining of the talk, we will only discuss ***fields***.

Let's work on this example application problem in C++, test-drive the `` `virtual` `` inheritance keyword.

Disclosure: This work is patent pending.

C++ plain MI: to virtual or not to virtual?

```
#define VIRTUAL  // virtual

class Person {

  String _name;

  String _addr;

};

class Student : public VIRTUAL Person {};

class Faculty : public VIRTUAL Person {};


class ResearchAssistant :

  public VIRTUAL Student , public VIRTUAL Faculty {};
```

Disclosure: This work is patent pending.

C++ plain MI: to virtual or not to virtual?

```
#define VIRTUAL virtual
```

(A) virtual inheritance: ResearchAssistant will have:

- 1 name
- 1 addr
- in total 2 fields

```
#define VIRTUAL  // empty
```

(B) default inheritance: ResearchAssistant will have:

- 2 names,
- 2 addrs
- in total 4 fields

None of them achieved the application semantics!

- The super-class’ fields are shared / separated as a **whole**
- Cannot treat each field **individually**: i.e `name` shared, but `addr` separated

Let’s check C++ MI memory layout.

Disclosure: This work is patent pending.

C++ MI memory layout … as clear as mud!

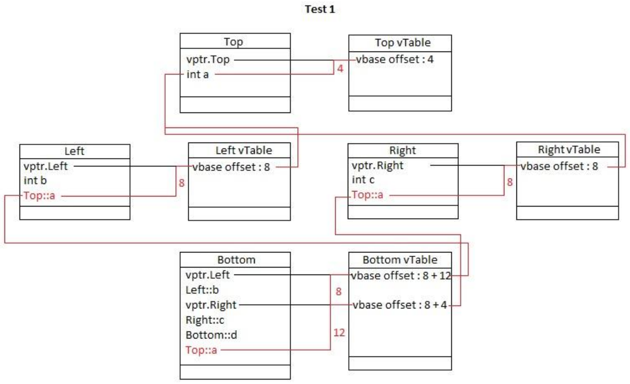
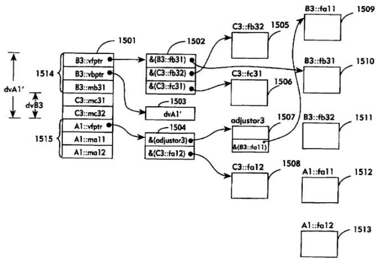


Figure 15

From a patent by Microsoft about MI: (US5754862A)



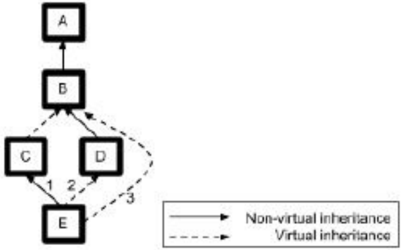

```
class A {
public:
   A(){
        cout << "A::A()" << endl;
   }
   A(int x){
        cout << "A::A(int)" << endl;
   }
};
```

```
class B : public A {
public:
   B(){
        cout << "B::B()" << endl;
   }
   B(int x){
        cout << "B::B(int)" << endl;
   }
};
```

```
class C : public virtual B {
public:
   C(){
        cout << "C::C()" << endl;
   }
   C(int x){
        cout << "C::C(int)" << endl;
   }
};
```

```
class D : public B {
public:
   D(){
        cout << "D::D()" << endl;
   }
   D(int x) : B(x){
        cout << "D::D(int)" << endl;
   }
};
```

```
class E : public C, public virtual
D, public virtual B {
public:
   E(){
        cout << "E::E()" << endl;
   }
   E(int x) : D(x){
        cout << "E::E(int)" << endl;
   }
};
```



Disclosure: This work is patent pending.

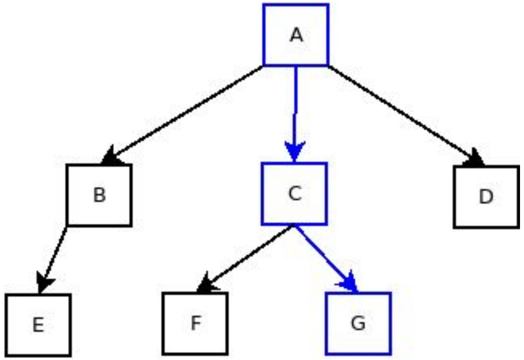
Problem 1: C++ MI memory layout … it’s messy!

Traditionally, all the fields from the all base classes are inherited.
BUT in the derived class:

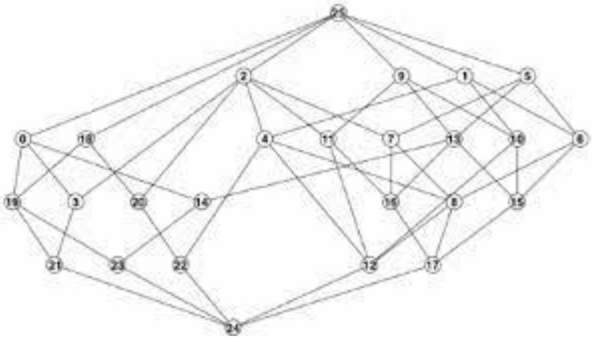
- Should the memory layouts of all the different base classes’ fields be kept intact in the derived class? and ***in which (linear memory) order***?
- How to handle: if the programmers want **some** of the inherited fields from different base classes to be **merged** into one field (e.g. name in the 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?

Disclosure: This work is patent pending.

The idea: stop inheriting data fields



Compare SI vs MI: for fields memory layout of any given class:
- In single inheritance, the path to root is linear, just tile them
  - E.g. for class G: [class A, class B, class G]
- In multiple inheritance, the path to root(s) is a lattice
  - However, the memory space is linear!
  - How to properly layout a lattice, with:
    - some joined (e.g. `name`)
    - others separated? (e.g. `addr`)



Inherited fields are too messy! … for both the
1. Compiler writers to get them right,
    a. … and to handle all kinds of *application semantics*
2. Developers to even understand

So let's just ***get rid of them***!

He who fights with monsters might take care lest he thereby become a monster.
— Friedrich Nietzsche, Beyond Good and Evil

Disclosure: This work is patent pending.

Problem 2: semantic branching

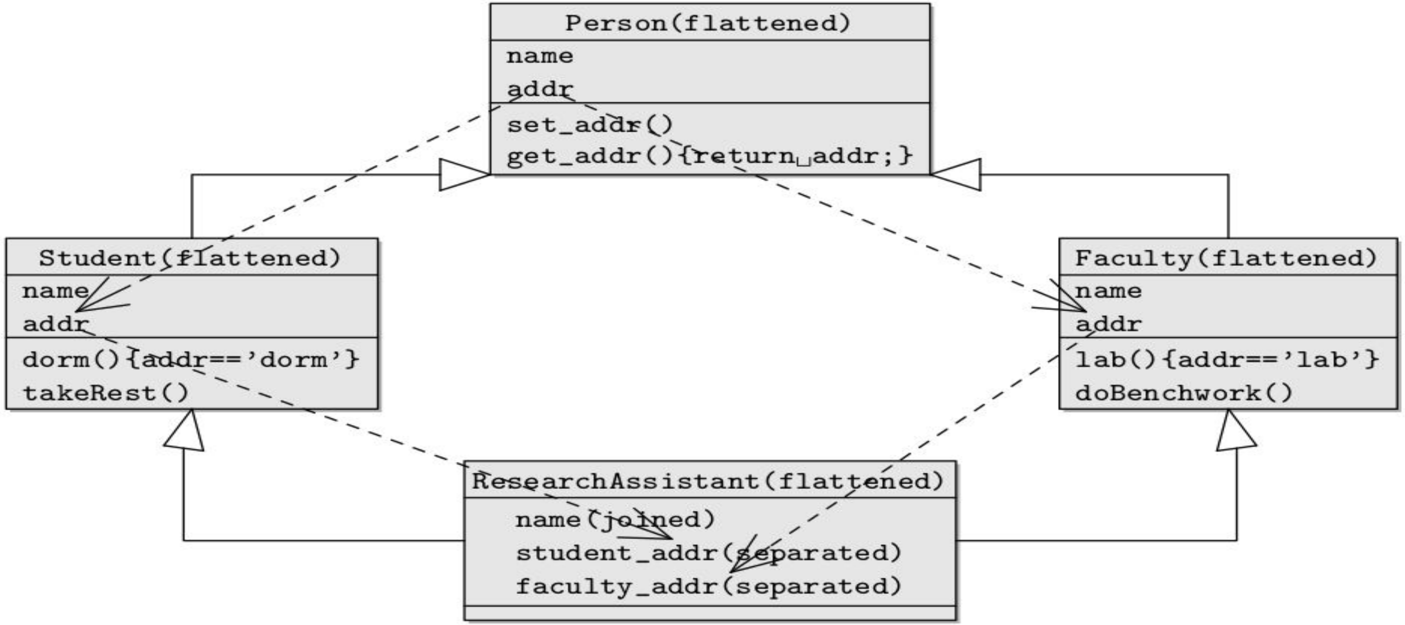


Fig. 1. the diamond problem: the ideal semantics of *fields* name & addr, which is not achievable in C++'s plain MI mechanism: with name joined into one field, and addr separated into two fields

Disclosure: This work is patent pending.

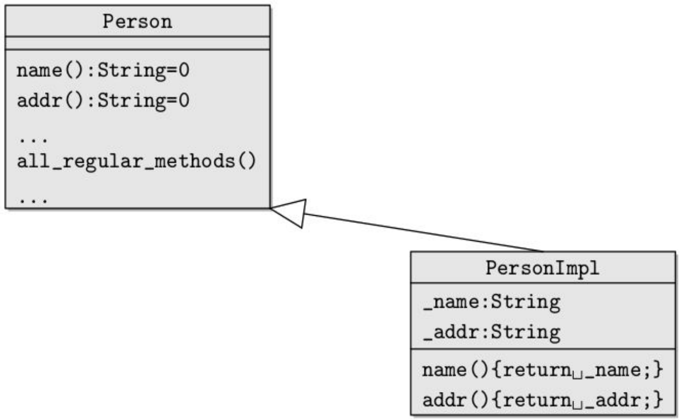
Current (less-ideal) engineering practice: use composition instead of MI

```
class ResearchAssistant : public StudentI, public FacultyI {
  Student _theStudentSubObject;  // composition
  Faculty _theFacultySubObject;  // composition

  // Problem 1: manual forwarding for *every* methods, i.e. code duplication
  void doBenchWork()  { _theFacultySubObject.doBenchWork(); }
  void takeRest()     { _theStudentSubObject.takeRest();    }

  String lab()  { return _theFacultySubObject._addr; }
  String dorm() { return _theStudentSubObject._addr; }

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

  String name() {
    set_name_mtx.lock();
    String r = _theStudentSubObject._name;
    set_name_mtx.unlock();
    return r;
  }
  String name(String name) {
    set_name_mtx.lock();
    _theStudentSubObject._name = name;  // dup fields
    _theFacultySubObject._name = name;
    set_name_mtx.unlock();
  }
};
```

DDIFI: the **inherited fields** are causing so much trouble, let's just ***get rid of them***!

Then how do we write regular methods?

- Well, just use: **abstract** property method (accessor) methods, i.e without actual field definition.
- **Decouple** data-interface (class Person with abstract property methods) from data-implementation (class PersonImpl where the fields and property methods are actually defined)
    - *Note: the data-interface class contains the regular methods **implementation**, which are meant to be **inherited (code reused)**!*
- Delay the data (field) definition only in the implementation class.



Disclosure: This work is patent pending.

Compare programming paradigms: procedural, OOP, DDIFI

<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>

Disclosure: This work is patent pending.

名不正，则言不顺；言不顺，则事不成。
(You must first name it properly, in order to talk about it intelligently.)

– Confucius

Define a new concept: **semantic branching site**
The two sub-class `Faculty` and `Student` actually has assigned *two different semantics* to their inherited `Person.addr`:

- `Faculty` use `addr` with “lab” semantics
- `Student` use `addr` with “dorm” semantics

We call `` `Person` `` is the **semantic branching site** of `` `addr` ``.

Then

- Field **join**: will be achieved by *overriding* virtual function of the *same* name
- Field **separation:** will be achieved by *defining and overriding new* semantic assigning property.

“Talk is cheap, show me the code.”

– Linus Torvalds

Now, let’s walk thru the code: https://github.com/joortcom/DDIFI

Disclosure: This work is patent pending.

```
// 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; }
};
```

Disclosure: This work is patent pending.

Immediately below the **semantic branching site**: Introduce **new semantic assigning property**:

```
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
};
```

Disclosure: This work is patent pending.

Immediately below the semantic branching site, Introduce new semantic assigning property:

```
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
};
```

Disclosure: This work is patent pending.

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

class ResearchAssistantImpl : public ResearchAssistant {  // only inherit from ResearchAssistant
 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 = “ResAssis”;
    _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; }
};
```

Disclosure: This work is patent pending.

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

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;
}

$ ./ddifi
ResAssis doBenchwork in the lab  # only one name: joined
ResAssis takeRest in the dorm    # but  two addr: separated
ResAssis doBenchwork in the lab  # total: 3 fields!
ResAssis takeRest in the dorm
```

Disclosure: This work is patent pending.

Alternative implementation of `ResearchAssistant`, use computation instead of raw field

```
// only inherit from ResearchAssistant interface, but not from any other xxxImpl class
class BioResearchAssistantImpl : public ResearchAssistant {
 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
  }
};
```

Disclosure: This work is patent pending.

Formalize it as 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 multiplely from any other data-interfaces.
2. class FooImpl inherit from Foo, as data implementation, which contains fields (if any) and implement property methods.

Rule 2 (data-interface class). In the data-interface class Foo:

1. define or override the **(abstract) properties** (from parent classes if any), 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.

Disclosure: This work is patent pending.

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).

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). E.g. as we showed in `ResearchAssistantImpl`.

Disclosure: This work is patent pending.

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.

E.g. this is how we handled `Person.addr`

Disclosure: This work is patent pending.

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.

The success of DDIFI depends on: method implementation without concrete fields definition.

… … does it ring a bell? :-)

Disclosure: This work is patent pending.

Java (v8.0, 2014) & C# (v8.0, 2019) default interface methods

Demo: DDIFI can be used in Java & C# to achieve **clean MI!**

code walk thru: https://github.com/joortcom/DDIFI/tree/main/java_csharp_python

So now with DDIFI, Oracle & Microsoft can **rebrand** their Java & C# as **clean** multiple inheritance languages ! 😂 (and D too, we will show).

In retrospect, C++ (Cfront v2.0) since 1989 has all the language mechanisms that DDIFI uses to achieve **clean** MI! But for decades, people avoided MI, haunted by the diamond problem complexity, until now we solved it.
Challenge: test w/ Cfront v2.0 https://github.com/joortcom/DDIFI/tree/main/cfront (and send me a PR).

DDIFI in C#, Python, Eiffel, other languages etc.: are left as an exercise.
Demo: We can do it in **D** too, YES! current D can do clean MI with DDIFI!
https://github.com/joortcom/DDIFI/blob/main/d/MI.d

- only a bit hackish: need to use template mixin + static if
- will be nicer, if D also supports Java's default interface methods.

Disclosure: This work is patent pending.

Pros & Cons

Pros:

- **Clean**: completely solved the diamond problem cleanly.
- **General**: works in C++ / Python / Java / C# / Eiffel / D! etc…

Cons:

- Each class now split into two classes: one as data-interface (also contains regular methods implementation), and the other as data-implementation.
  - Rebuttal: “program to interface” is a good practice in almost any serious software project already, which is well-understood by the developers.
- Must access fields using property method in public & protected methods, i.e. incur lots of virtual function calls.
  - Rebuttal: virtual methods is the corner-stone of OOP (since its start in 1960s’), it is well optimized by modern compilers.
  - Also one can use local temp vars to reduce the number of virtual property method calls needed.

Disclosure: This work is patent pending.

General guidelines

For planned MI, absolutely known to be field name-clash *free*, then use the language's native MI mechanism.

Otherwise, esp. for ***unplanned*** MI, (un-)anticipated field-name clash, use DDIFI:

1. First define fields as virtual property methods.
2. Then write regular-methods, by using the virtual property.
3. Implement the class property by either define data fields or via computation in the implementation class.

Disclosure: This work is patent pending.

An analogy

- Fields are like legs of a table.
- On top of these legs, we can build application functionalities (methods via computation), e.g place potted plants on top.
- But in certain scenarios (multiple inheritance), the solid legs caused too much trouble for us
- … then …

Disclosure: This work is patent pending.

This is what we can do:

Virtual legs (fields) are more flexible!

Q & A

Disclosure: This work is patent pending.