COMP 3550

6.1 — PROBLEMS WITH INHERITANCE

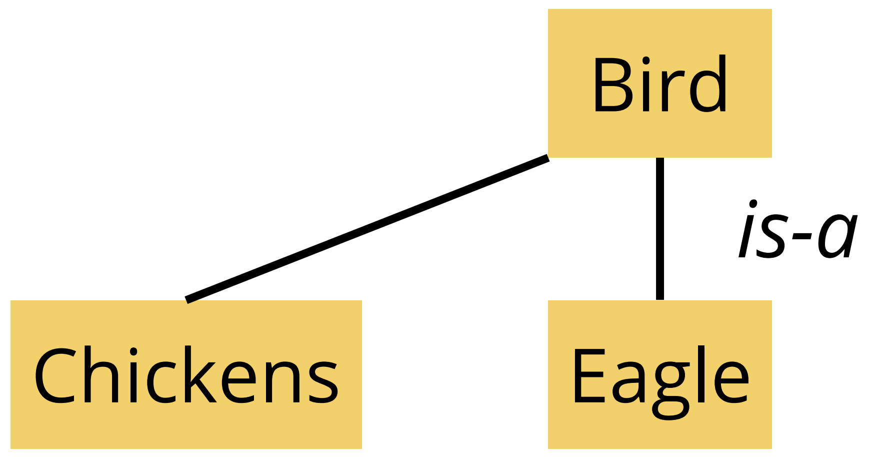
Week 6: Alternatives to Inheritance & Dependency Injection

INHERITANCE IN THEORY VS. PRACTICE

INHERITANCE: THE PROMISE

- Reuse existing code by extending a base class
- We learned this in second year
- One base class & many specialized subclasses

Inheritance allows for reuse… but at what cost?

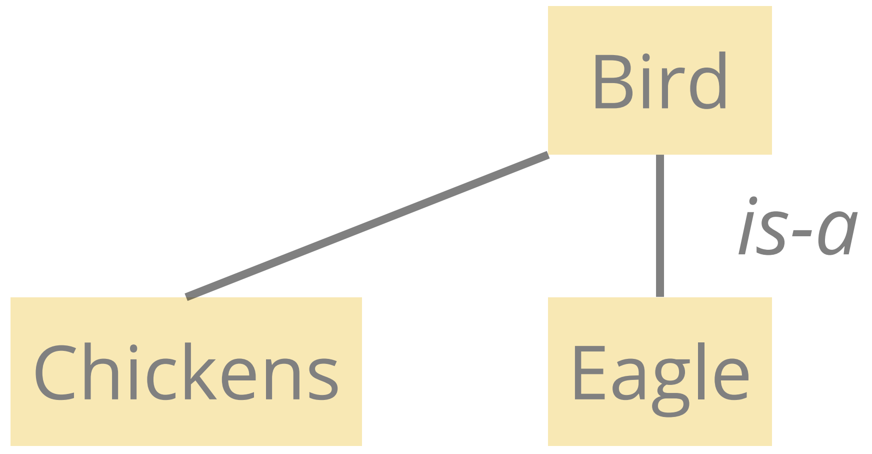


Himbeault 2025 ©

INHERITANCE IN THEORY VS. PRACTICE

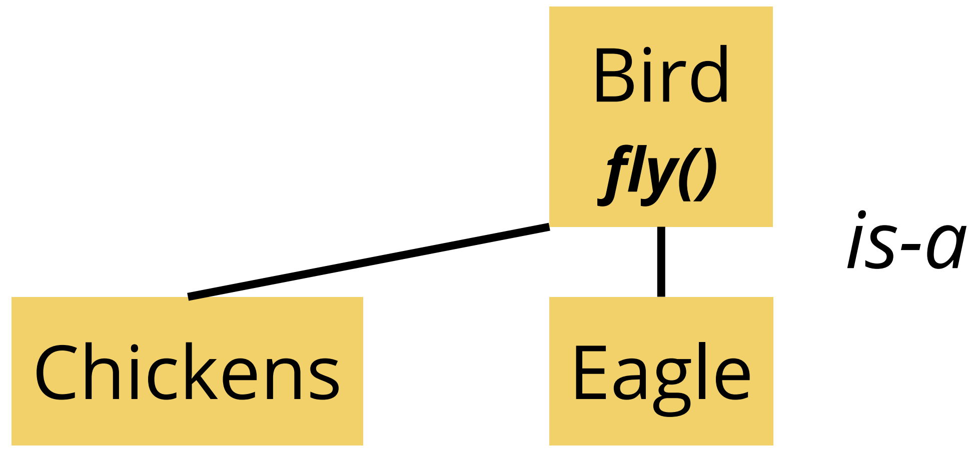
INHERITANCE: THE PROMISE

- Reuse existing code by extending a base class
- We learned this in second year
- One base class & many specialized subclasses



INHERITANCE: THE REALITY

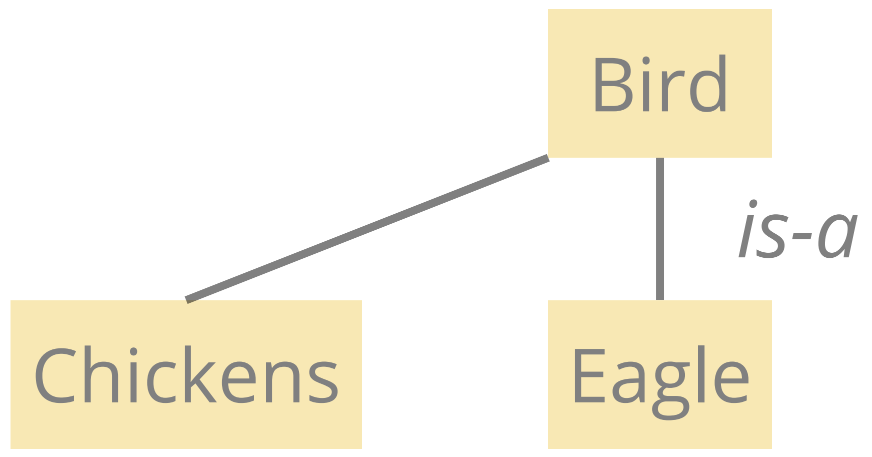
- Tight coupling
  - changes to the base ripple everywhere
- Fragile hierarchies
  - adding a new subclass breaks assumptions
- Inflexible design
  - locked into a certain model early



INHERITANCE IN THEORY VS. PRACTICE

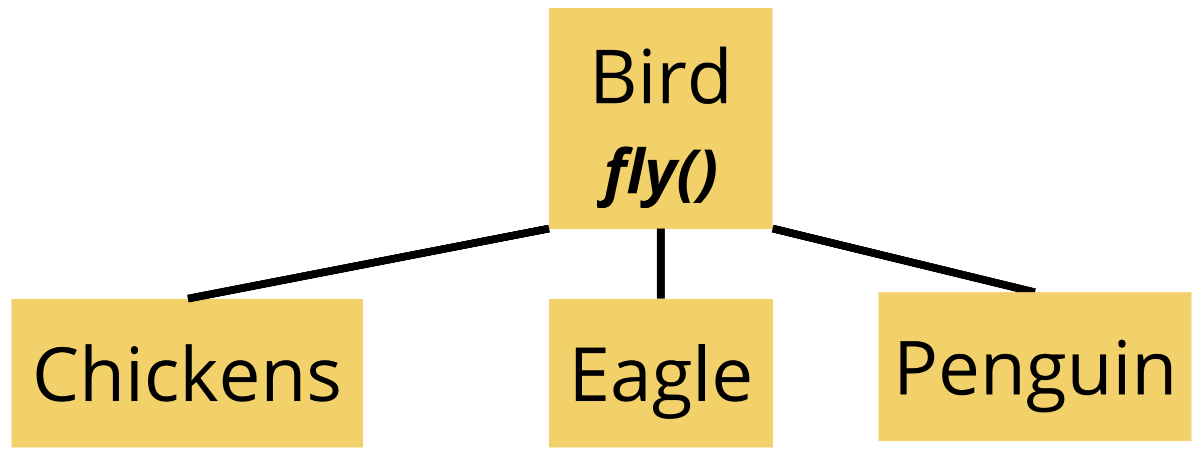
INHERITANCE: THE PROMISE

- Reuse existing code by extending a base class
- We learned this in second year
- One base class & many specialized subclasses



INHERITANCE: THE REALITY

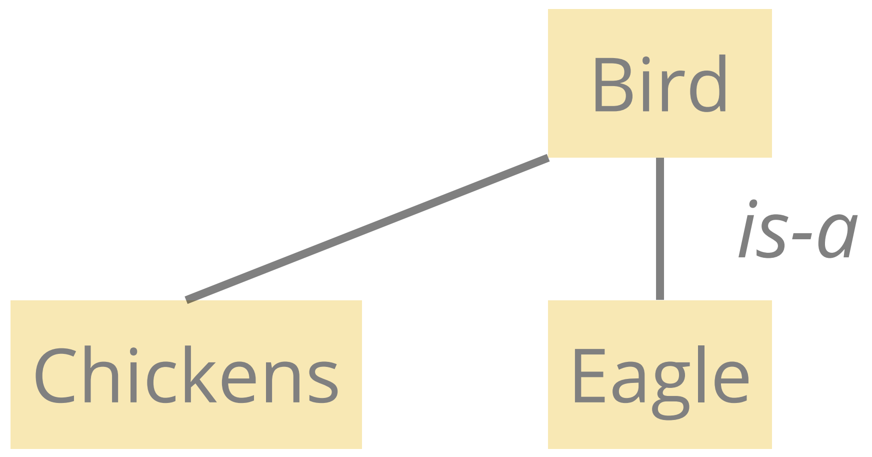
- Tight coupling
  - changes to the base ripple everywhere
- Fragile hierarchies
  - adding a new subclass breaks assumptions
- Inflexible design
  - locked into a certain model early



INHERITANCE IN THEORY VS. PRACTICE

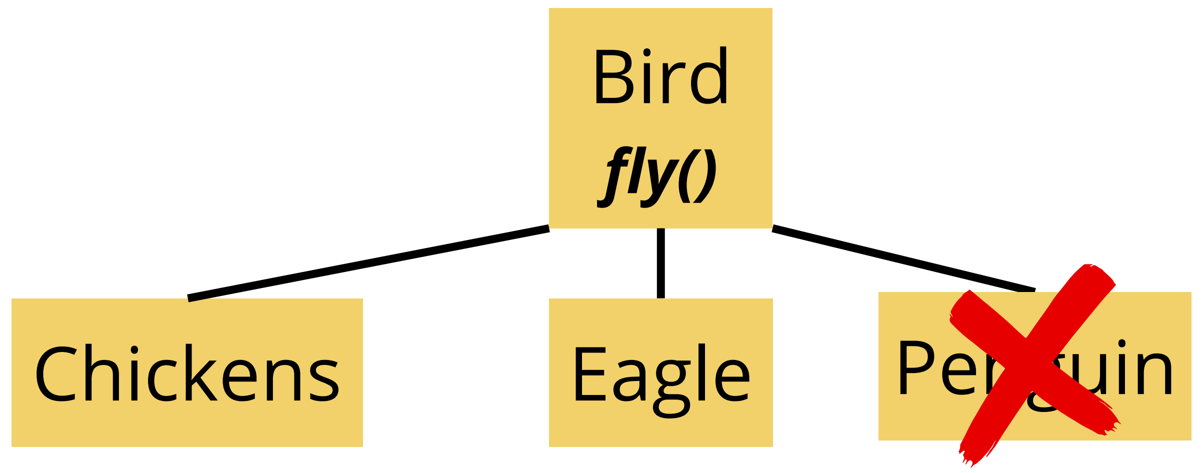
INHERITANCE: THE PROMISE

- Reuse existing code by extending a base class
- We learned this in second year
- One base class & many specialized subclasses



INHERITANCE: THE REALITY

- Tight coupling
  - changes to the base ripple everywhere
- Fragile hierarchies
  - adding a new subclass breaks assumptions
- Inflexible design
  - locked into a certain model early



REAL-WORLD FRAGILITY EXAMPLES

- **Subclass overrides breaking parent contracts**
  - A reminder of **LSP**: “If Square is a kind of Rectangle, but changes setWidth() and setHeight() so they no longer behave as expected, the parent’s promises are broken.”
- **Inheriting unused methods**
  - Example: A Printer class with scan() and fax() methods — inherited by ThermalReceiptPrinter that doesn’t support either.
- **Small change in parent = big surprise in child**
  - Example: Base class changes default sorting from ascending to descending and suddenly child’s results are wrong.

- We end up with **The Fragile Base Class Problem**

THE FRAGILE BASE CLASS PROBLEM

```java
class ReportGenerator {
    void generate() {
        fetchData();
        format();
        print();
    }

    protected void format() {
        System.out.println("Formatting as PDF...");
    }
}

class HTMLReportGenerator extends ReportGenerator {
    @Override
    protected void format() {
        System.out.println("Formatting as HTML...");
    }
}
```

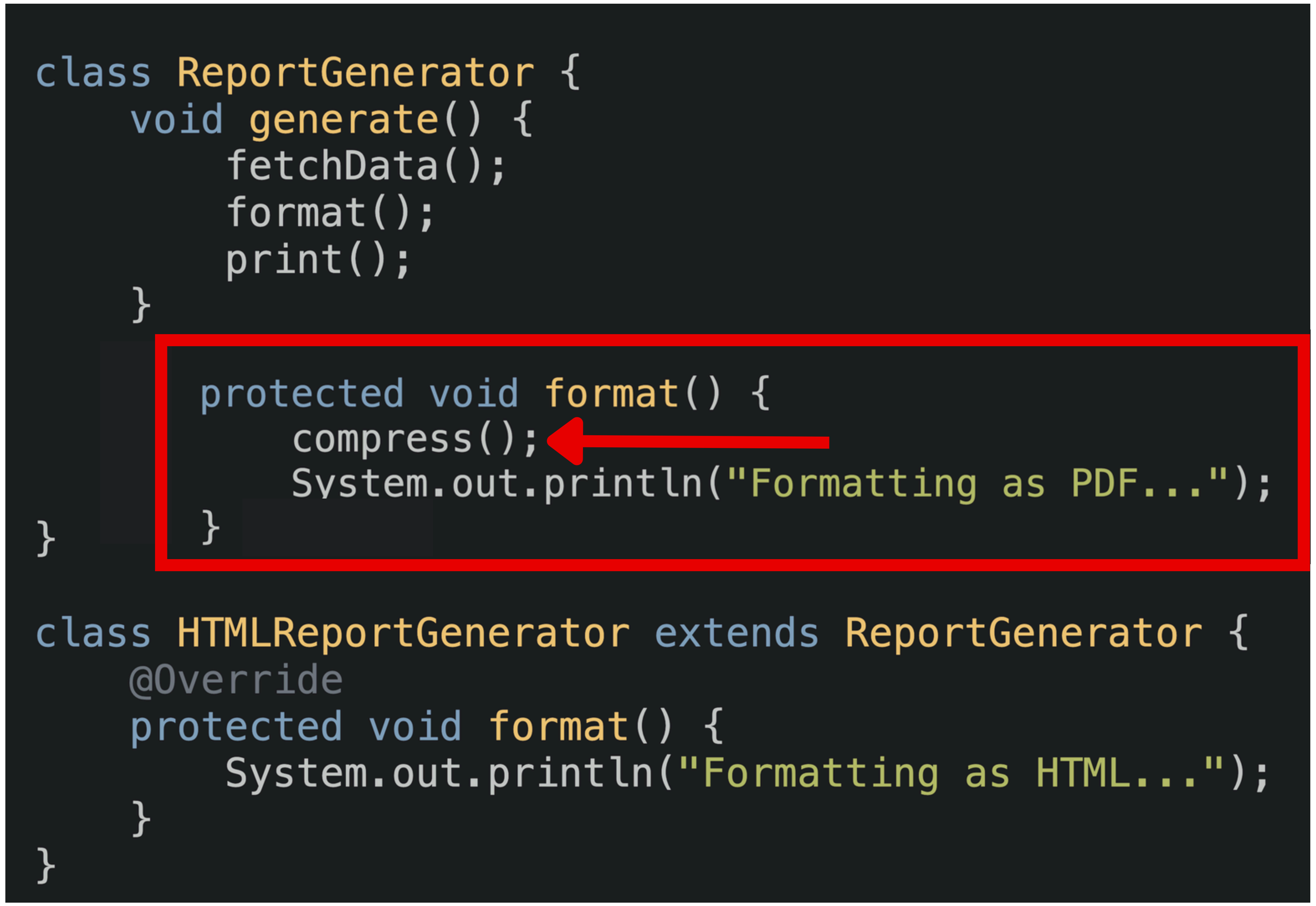
THE FRAGILE BASE CLASS PROBLEM

```
class ReportGenerator {
    void generate() {
        fetchData();
        format();
        print();
    }

    protected void fo
        System.out.pr
    }
}

class HTMLReportGener
    @Override
    protected void fo
        System.out.pr
    }
}
```

THE FRAGILE BASE CLASS PROBLEM



```java
class ReportGenerator {
    void generate() {
        fetchData();
        format();
        print();
    }

       protected void format() {
           compress();
           System.out.println("Formatting as PDF...");
       }
}

class HTMLReportGenerator extends ReportGenerator {
    @Override
    protected void format() {
        System.out.println("Formatting as HTML...");
    }
}
```

THE DIAMOND OF DOOM & TIGHT COUPLING

When inheritance paths cross, and everything gets tangled.

The Diamond of Doom (concept)

- **Problem:** Class inherits from two classes that share a common ancestor
- **Leads to ambiguity:** Which version of the shared ancestor’s method should be used?
- Java avoids this for classes, but it’s common in other languages (C++, Python) and still possible (in Java) via interfaces with default methods.

THE DIAMOND OF DOOM & TIGHT COUPLING

When inheritance paths cross, and everything gets tangled.

The Diamond of Doom (concept)

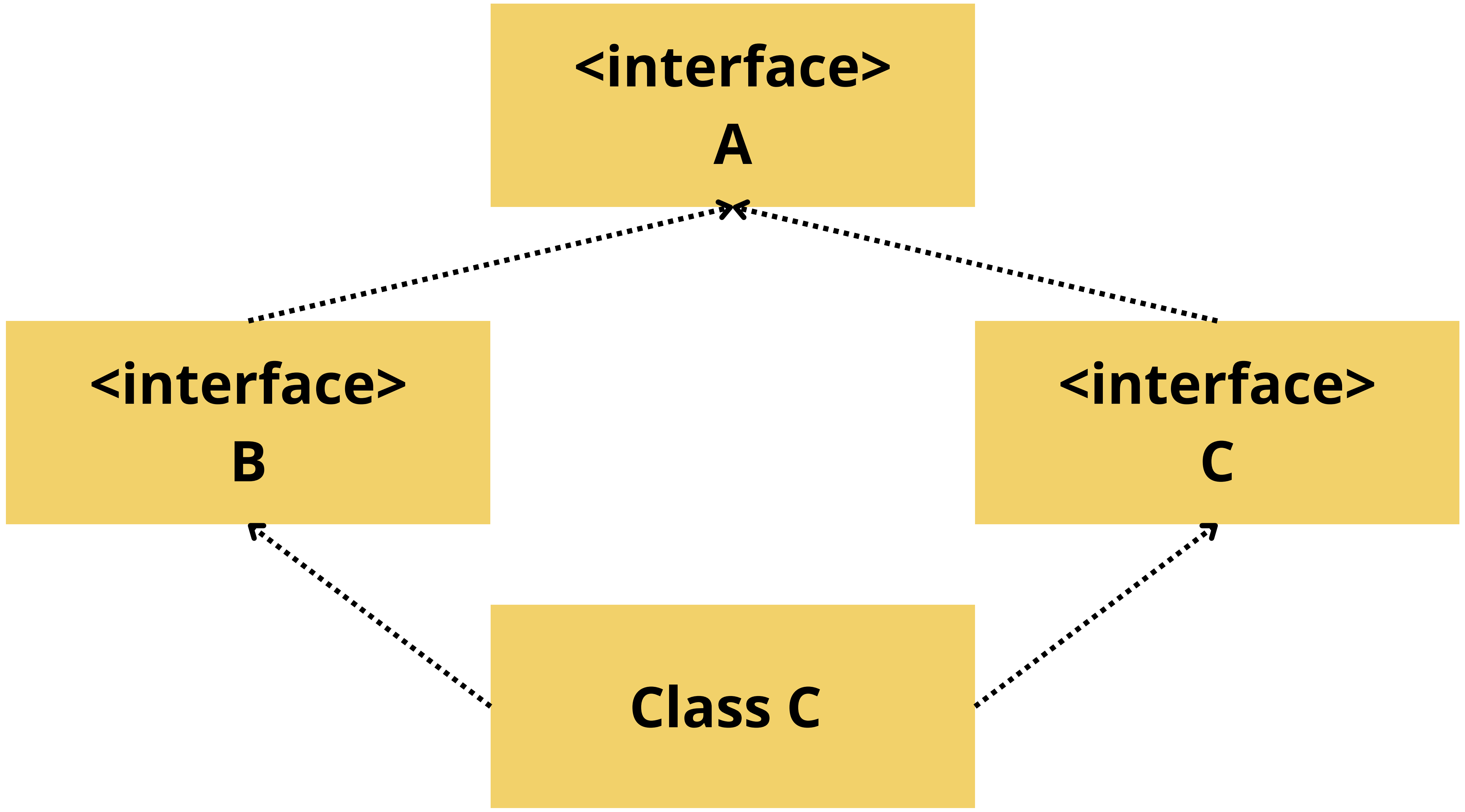
- **Problem:** Class inherits from two classes that share a common ancestor
- **Leads to ambiguity:** Which version of the shared ancestor's method should be used?
- Java avoids this for classes, but it's common in other languages (C++, Python) and still possible (in Java) via interfaces with default methods.

Tight Coupling tie-in

- Even without multiple inheritance, deep or cross-linked hierarchies:
  - Changes in one class ripple to many
  - Harder to test in isolation
  - Subclasses locked into parent's choices

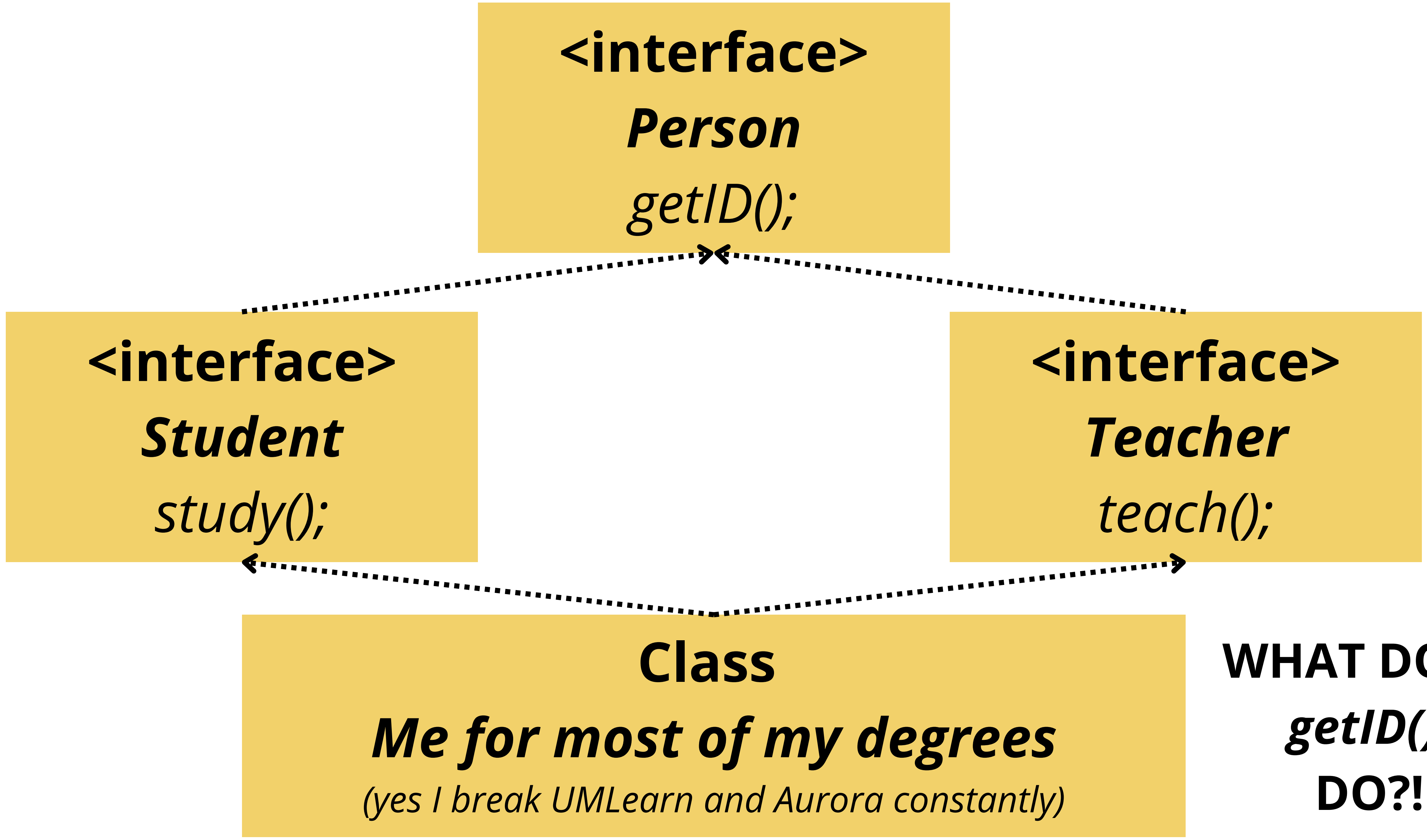
THE DIAMOND OF DOOM & TIGHT COUPLING

When inheritance paths cross, and everything gets tangled.



THE DIAMOND OF DOOM & TIGHT COUPLING

When inheritance paths cross, and everything gets tangled.

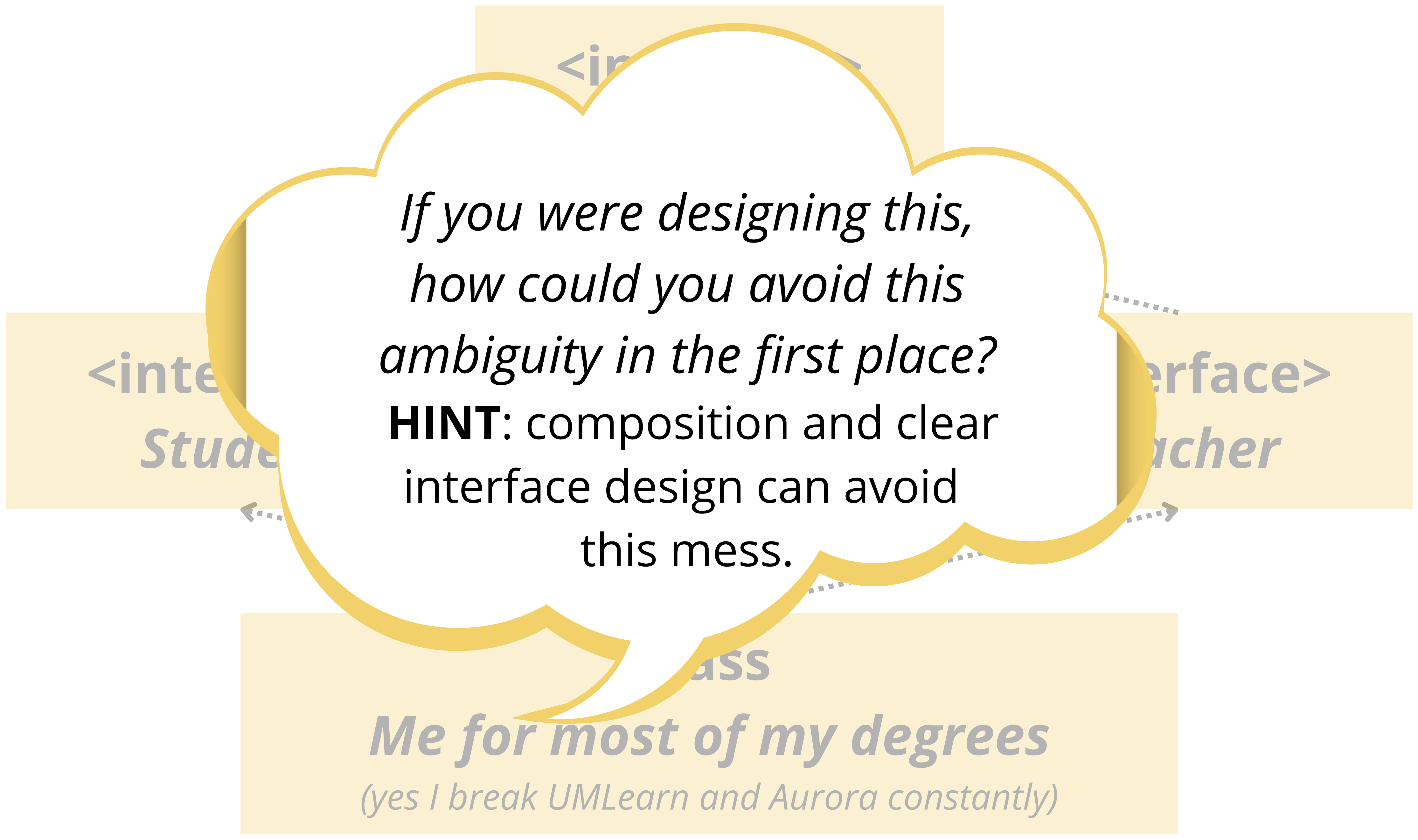


WHAT DOES *getID()* DO?!

Himbeault 2025 ©

THE DIAMOND OF DOOM & TIGHT COUPLING

When inheritance paths cross, and everything gets tangled.


<in
<inte
Stud
erface>
acher
ass
Me for most of my degrees
(yes I break UMLearn and Aurora constantly)

If you were designing this, how could you avoid this ambiguity in the first place?

HINT: composition and clear interface design can avoid this mess.

IS-A...NIGHTMARE?

Think about genetics and your parents for a second....

Which sentence makes more sense?

Child A ***is a*** *genetic copy of their biological parent?*

or

Child A ***has some of*** *the genetic information from their biological parent?*

HOW DO WE SOLVE THIS?

From “IS-A” to “HAS-A”

Recap of the analogy:

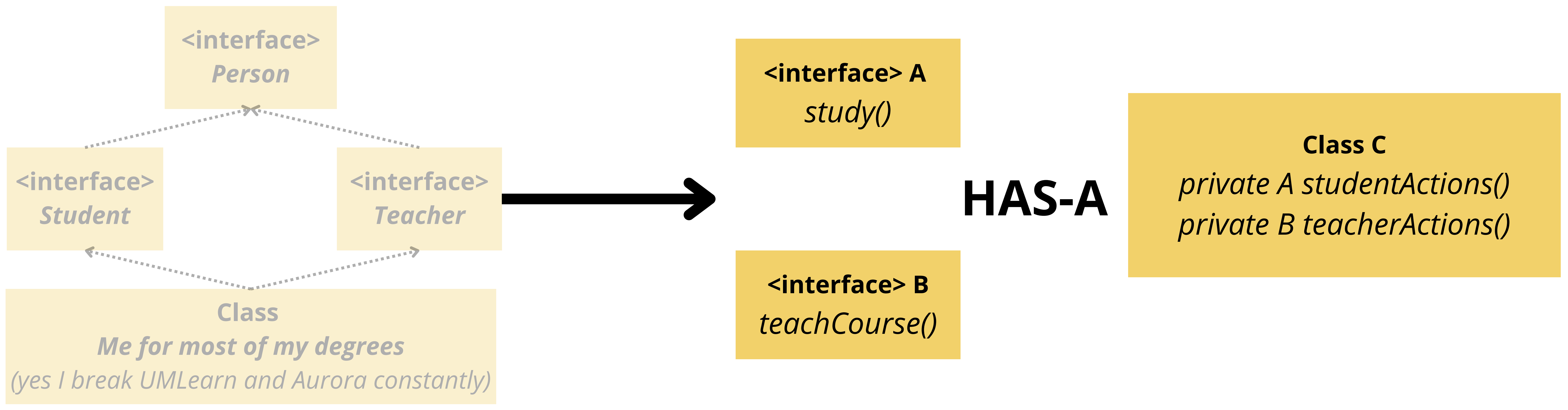
- **IS-A (inheritance)**: Child A is a type of Parent, shares core blueprint.
- **HAS-A (composition):** An object contains another as a part of its makeup, not the same type.

A Child HAS eyes, a brain, skin, etc.

- This means I can HAVE my Biological parents eye colour, or skin color, etc.
- I **do not have THEIR** eyes. That would be different...and weird.

HOW DO WE SOLVE THE LAUREN PROBLEM?

Depends who you ask I would imagine but in code:
Multiple Roles, Not Multiple Inheritance
Interfaces are for BEHAVIOURS, not properties



<interface> Person
<interface> Student
<interface> Teacher
Class
Me for most of my degrees
(yes I break UMLearn and Aurora constantly)
<interface> A
study()
<interface> B
teachCourse()
HAS-A
Class C
private A studentActions()
private B teacherActions()

RULE OF THUMB

Quick aside: where does this term even come from?

- *17th and 18th centuries, tradespeople like carpenters, brewers, and millers used the thumb as a quick and easy way to estimate measurements.*
- *i.e. a general approximation we can safely use most of the time*

Interestingly:
A persistent, but ***unfounded and false****, theory connects the phrase to domestic violence.*

Use inheritance (meaning superclasses, interfaces, etc.) for "is-a" and shared behavior, otherwise, prefer composition

PROJECT PAUSE & REFLECT

Find one class using inheritance.

*(**again**, this can still mean interface implementation) in your project.)*

Can you replace it with composition?