Software Development (CS2500)

Lecture 26: Inheritance (Continued)

M. R. C. van Dongen

November 22, 2013

Outline

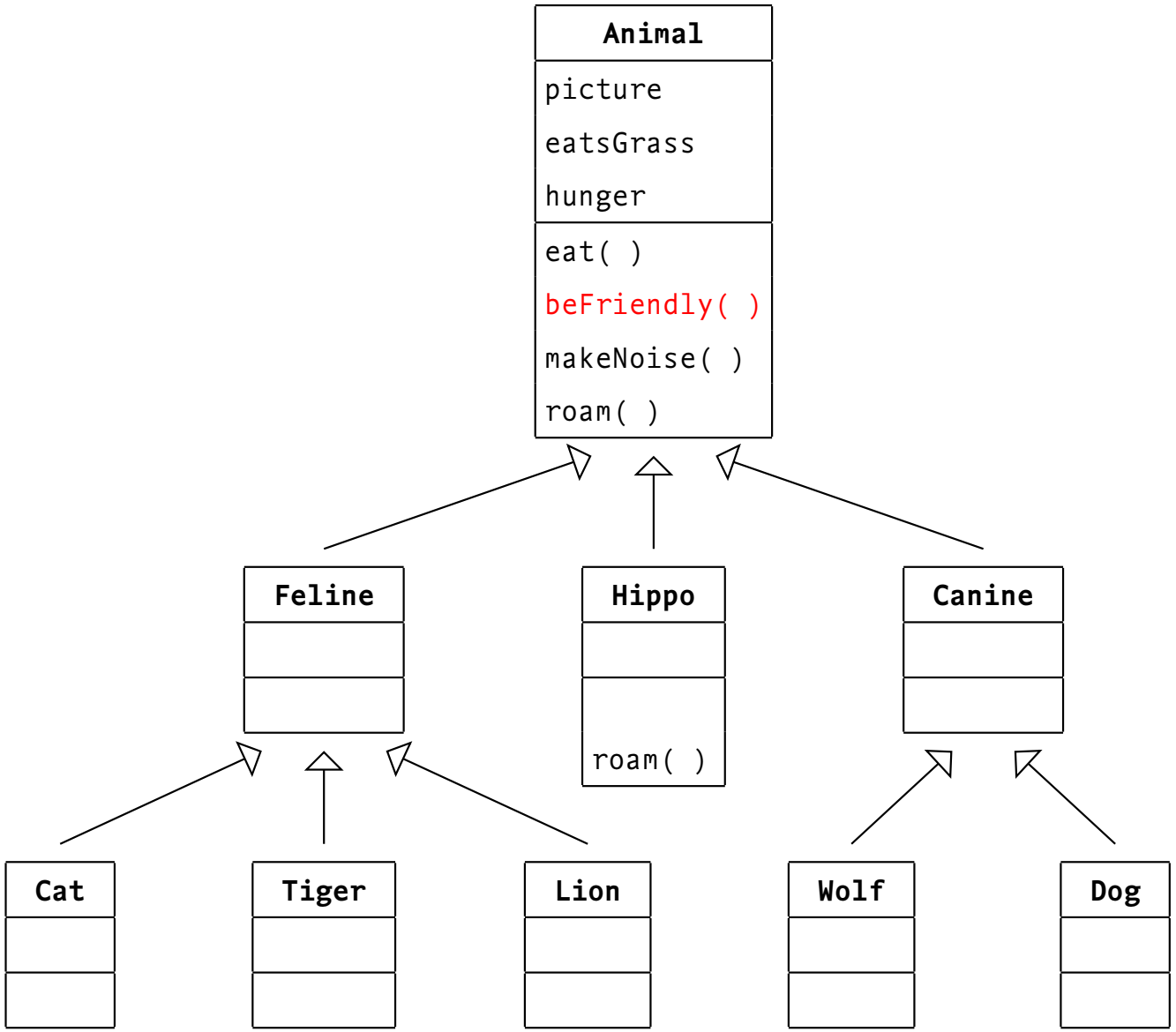
- We study *multiple inheritance:*
  - We start with a case study;
  - We see advantages and disadvanteges of different designs;
  - We end up with a disasterous design complication;
  - We learn how overcome the complication.
- We study the *strategy design pattern:*
  - Defines a class of related algorithms;
  - Encapsulates them;
  - Makes them interchangable.
- We learn three disign principles:
  - Encapsulate what varies;
  - Program to an interface;
  - Favour composition over inheritance.

Multiple Inheritance

- Let's introduce `Pets` to our `Animal` class hierarchy.
- The `Pets` can `beFriendly( )`.
- Other animals don't have `beFriendly( )` behaviour.
- Our design should allow for polymorphic pet variables.

Adding Pets to our Fota Application

Option I: Adding the `Pet` Method to the `Animal` Class



Adding Pets to our Fota Application

Option I: Adding the `Pet` Method to the `Animal` Class

Pros: The are two main advantages:

1. All `Pets` will inherit `Pet` behaviour, and
2. `Animal` can act as a polymorphic type for `Pets`.

Cons: There are also disadvantages:

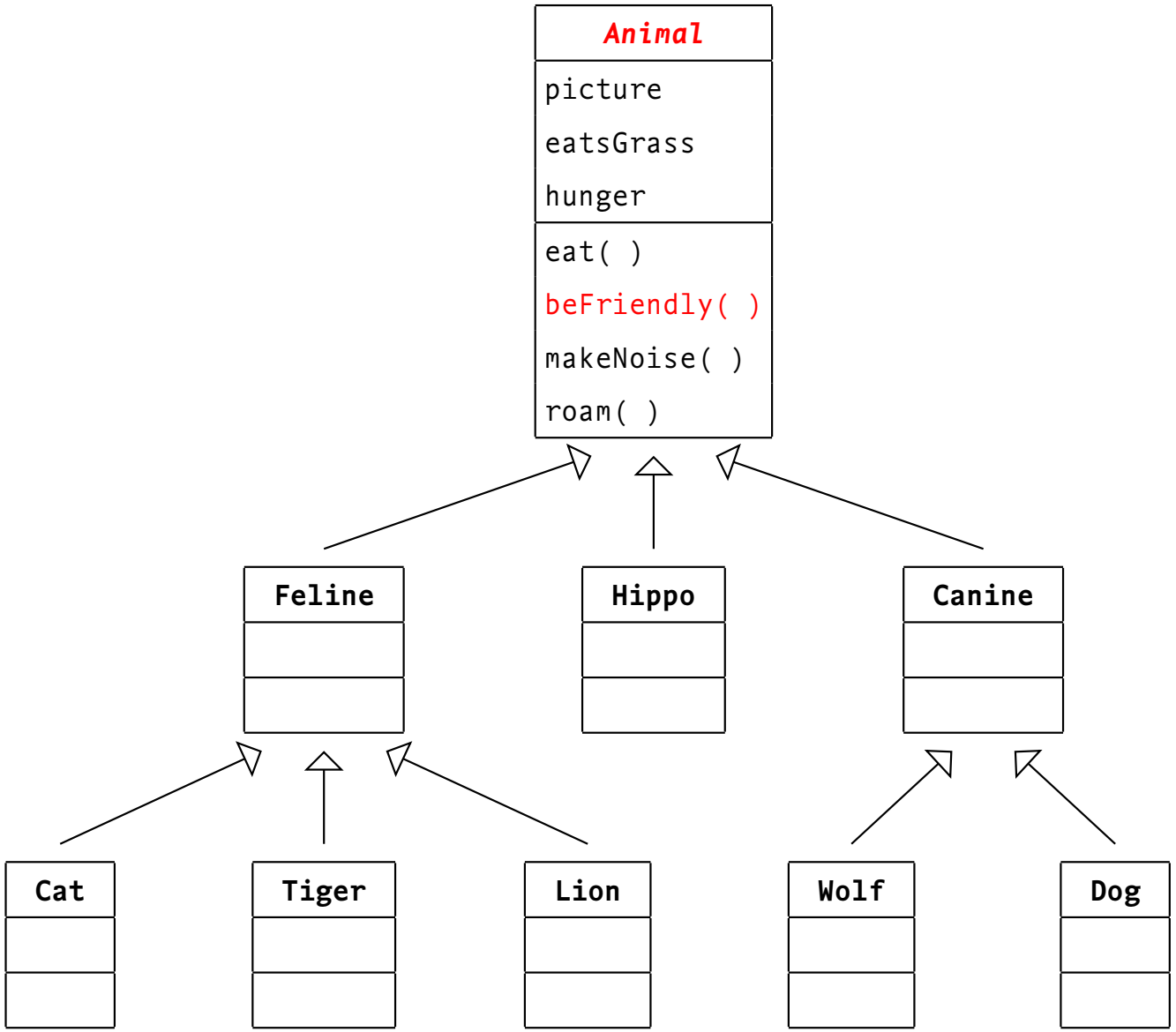
1. We don't have a proper `Pet` type.
2. Non-`Pets` will also get `beFriendly( )` behaviour.
3. Still must override `beFriendly( )` for `Dog` & `Cat`.

Conclusion: Clearly the disadvantages outweigh the advantages.

Cause: The Is-A test fails for non-`Pets`.

Adding Pets to our Fota Application

Option II: As Option I but Make `Animal` Class Abstract



Adding Pets to our Fota Application

Option II: As Option I but Make `Animal` Class Abstract

Pros: The advantages are better than before.

1. We can make all animals behave appropriately.
2. `Animal` can act as a polymorphic type for `Pets`.

Cons:

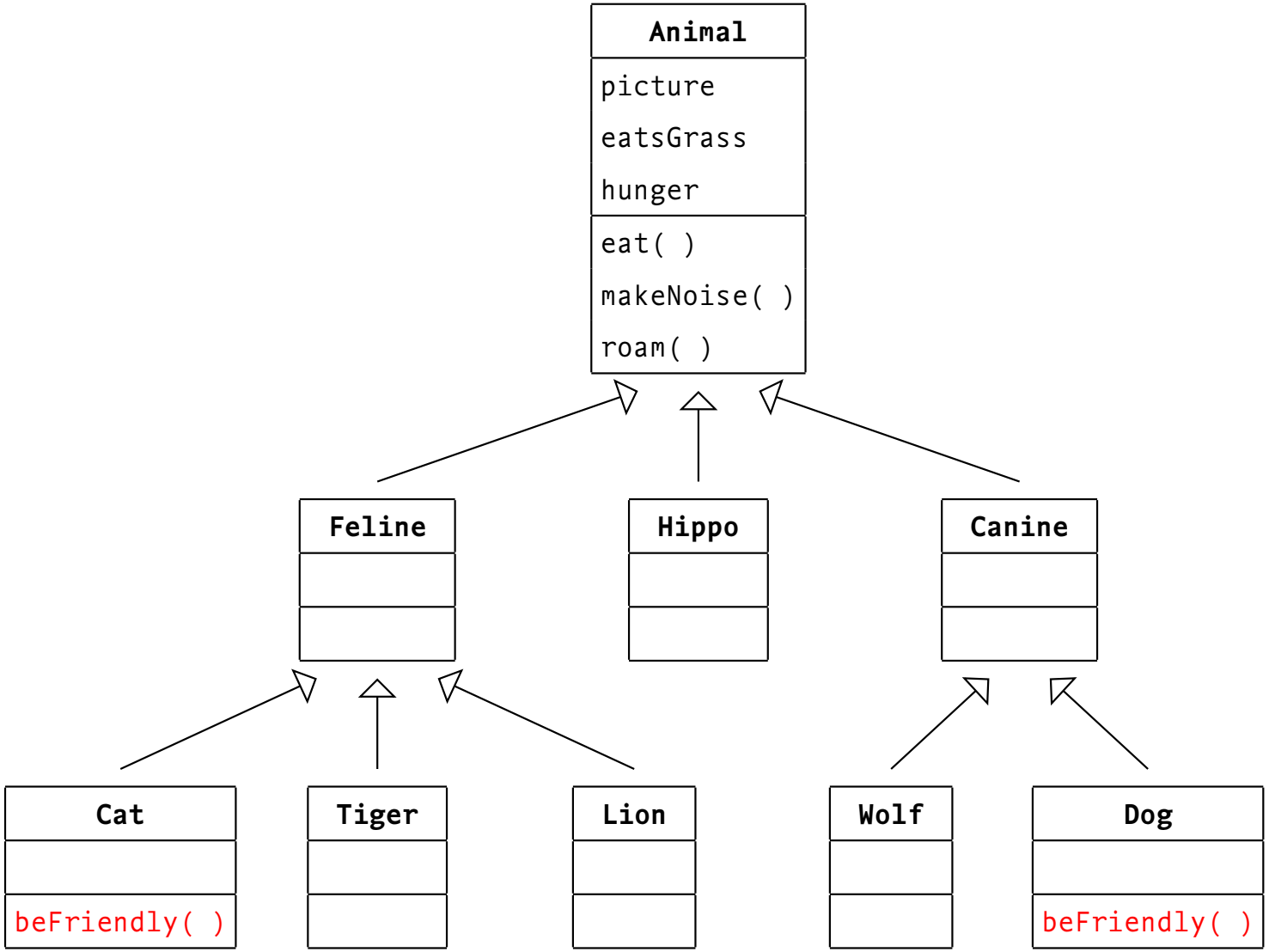
We still don't have a proper `Pet` type. Must override `beFriendly( )` in all concrete classes.

Conclusion: This design is worse than Option I.

Cause: The Is-A test fails for non-`Pets`.

Adding Pets to our Fota Application

Option III: Put the `Pet` Method where It Belongs



Adding Pets to our Fota Application

Option III: Put the `Pet` Method where It Belongs

Pros: The following are some advantages.

- Definition of `beFriendly( )` is where it belongs.
- Implementing `beFriendly( )` requires little effort.
- All animals behave appropriately.

Cons: The following are some disadvantages.

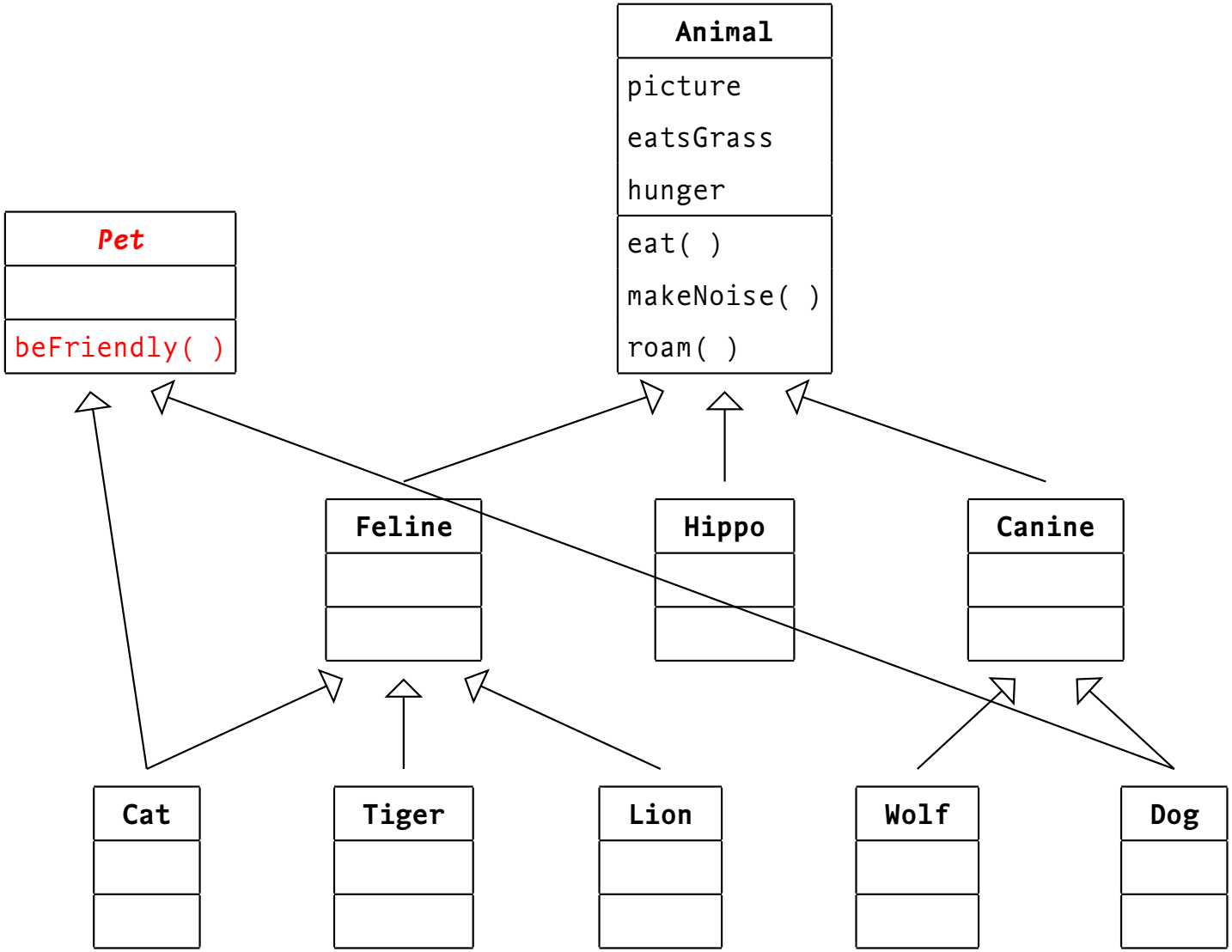
- We still don't have a proper `Pet` type.
- The `befriendly( )` method isn't abstract.
  - We can't guarantee a consistent `beFriendly( )`.
- We lose a proper polymorphic type for `Pets`.

Conclusion: This design makes `Pets` difficult to work with.

Cause: Polymorphism is a requirement for most applications.

Adding Pets to our Fota Application

Option IV: Two Superclasses for Pets



Adding Pets to our Fota Application

Option IV: Two Superclasses for `Pets`

Pros: The following are the advantages.

- The `beFriendly( )` method is where it belongs.
- Implementing `beFriendly( )` requires little effort.
- Guarantees consistent `beFriendly( )` definitions.
- `Pet` can act as a polymorphic type for pets.

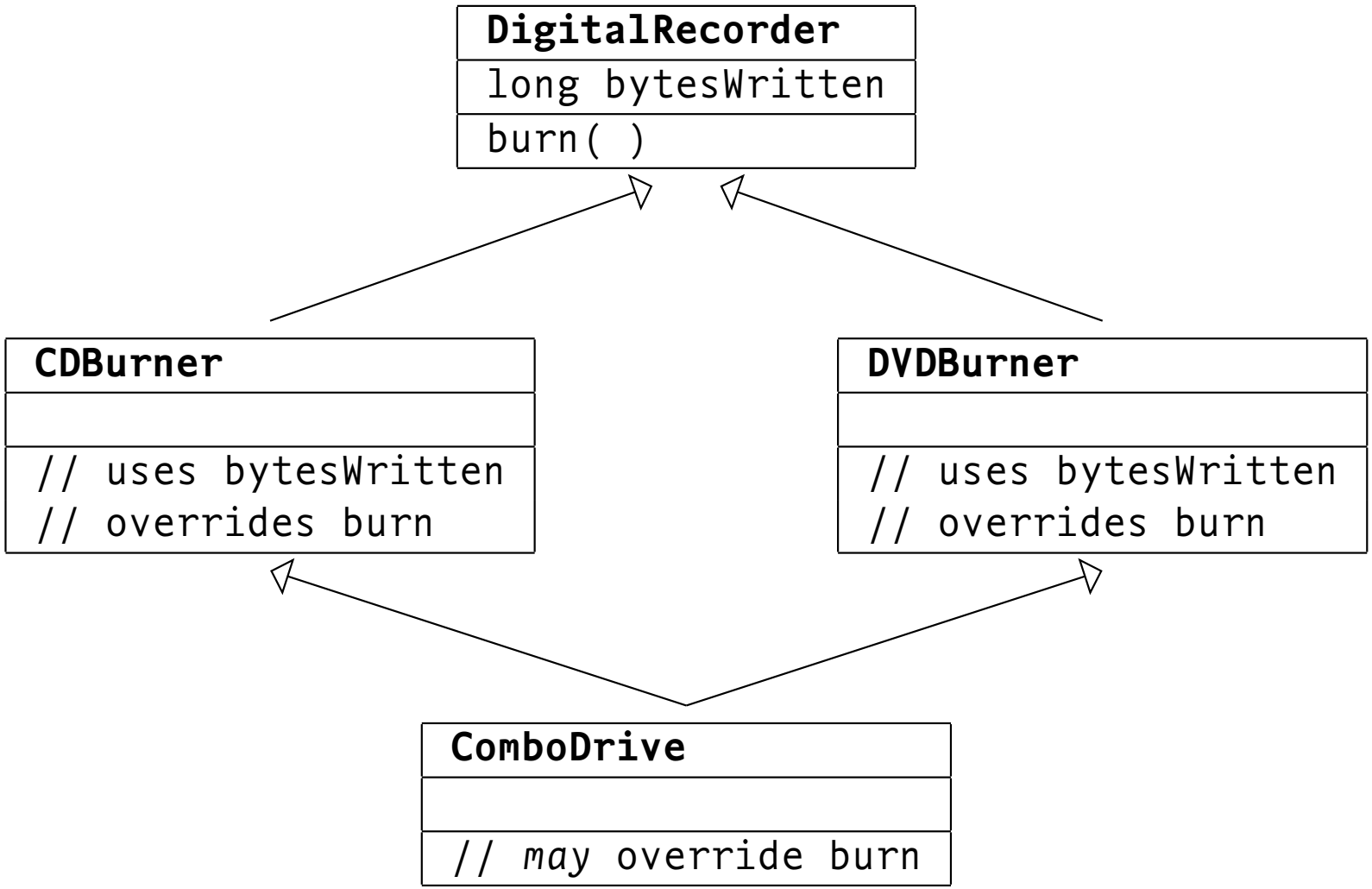
Cons: `Java` doesn't allow *multiple inheritance*.

Conclusion: This design is ideal but impossible.

Cause: A decision by the `Java` language designers.

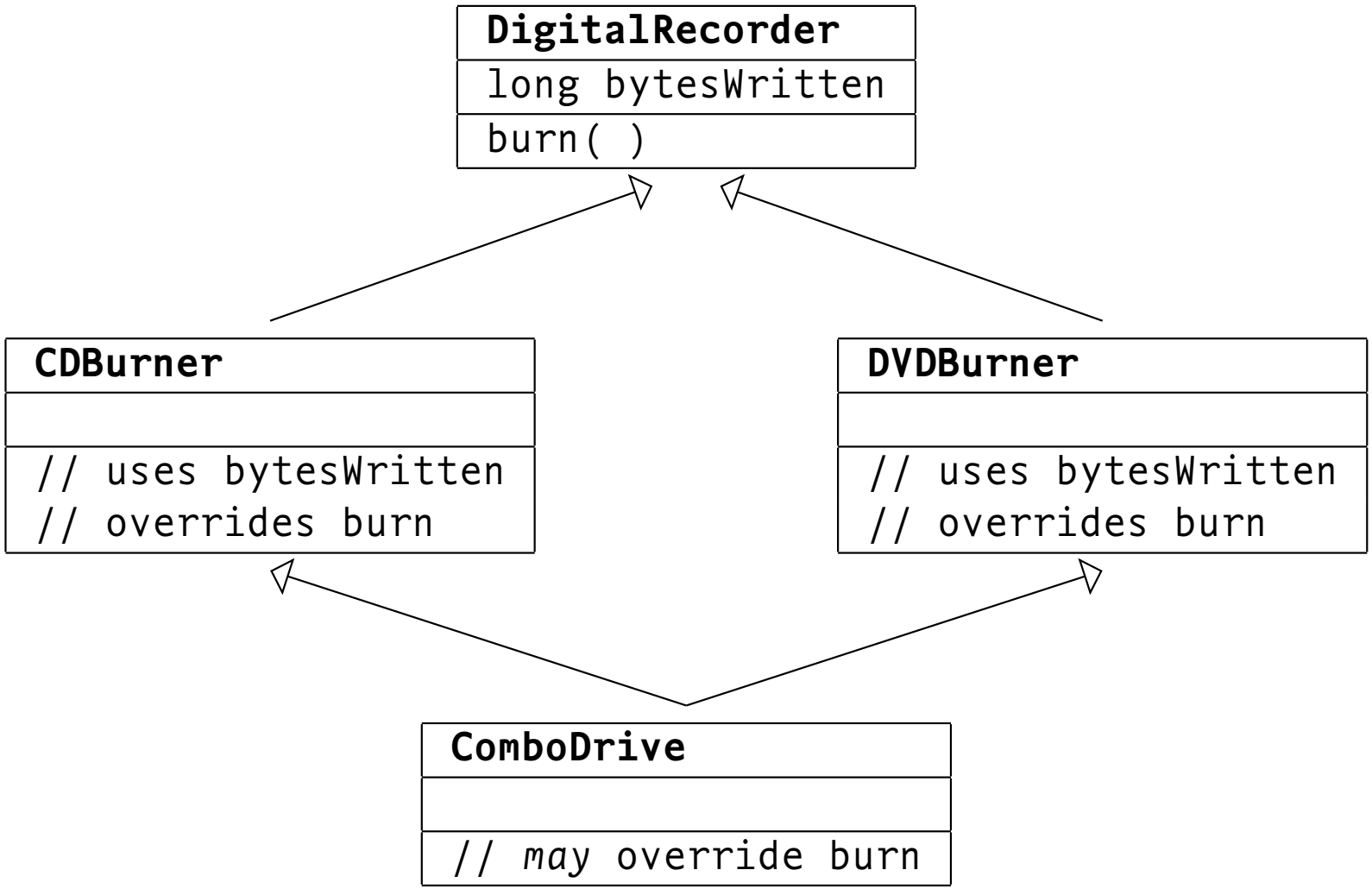
Deadly Diamond of Death

Different Assumptions about Valid Values for `bytesWritten`



Deadly Diamond of Death

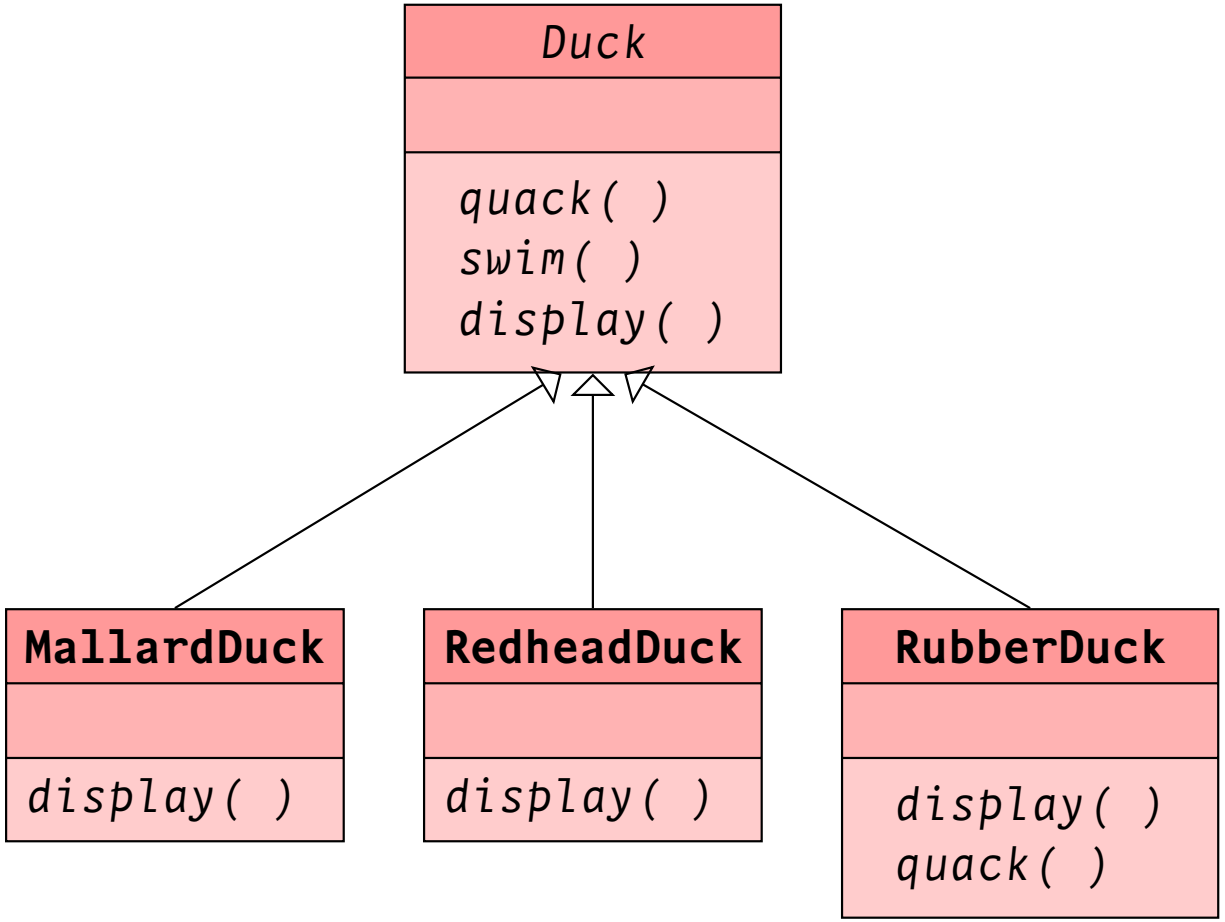
Which `burn( )` Should be Overriden?



Implementing Duck Games

- Joe works at SimuDuck™.
- SimuDuck™ specialises in  pond simulation games.
  - These games involves lots of quacking and swimming  s.
- Joe is in charge of SimuDuck™'s most popular game.
- The game is written in `Java` and is based on inheritance.

The Design



Enters Mr Change

Enters Mr Change

Enters Mr Change

Enters Mr Change

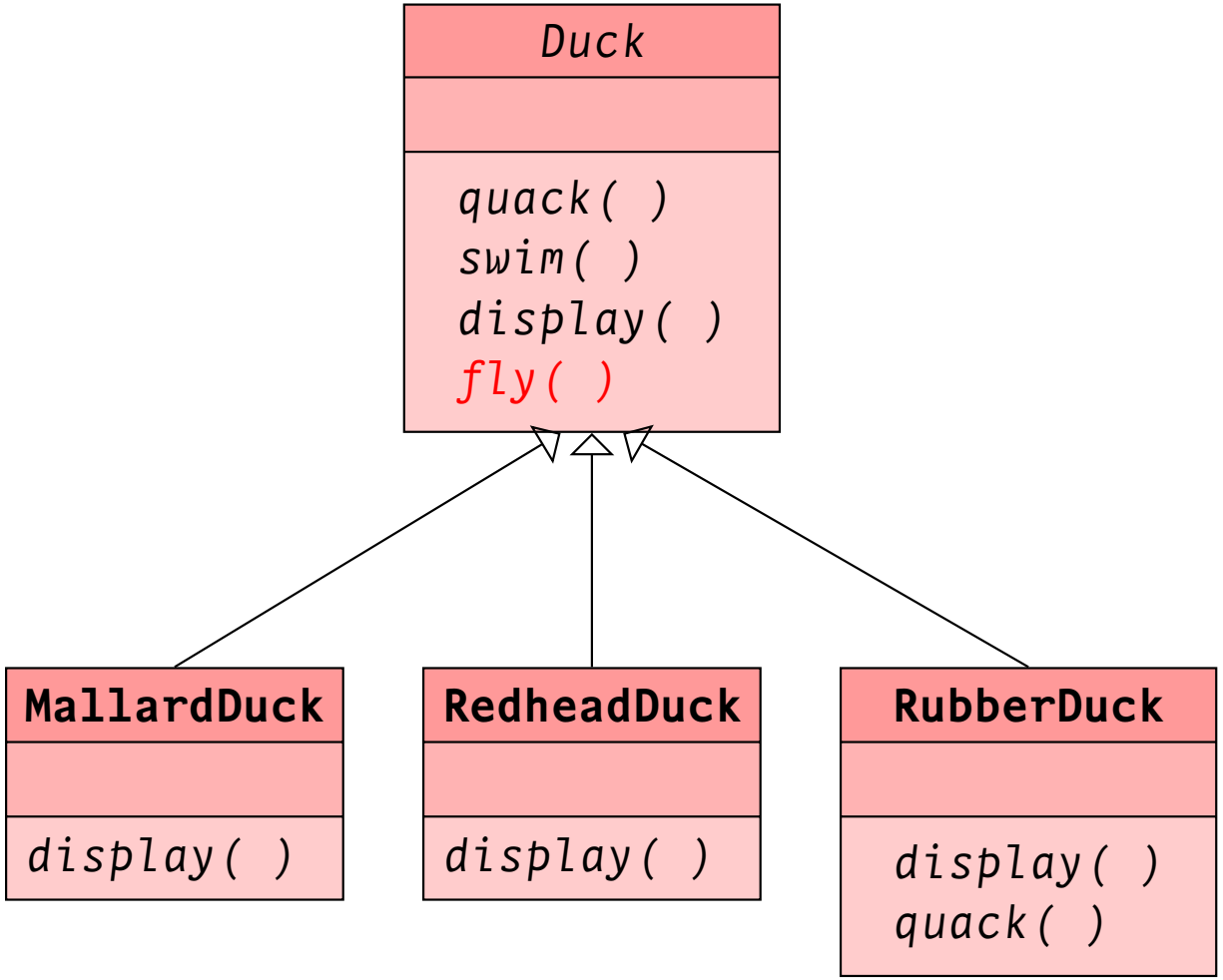
Enters Mr Change

Enters Mr Change

It requires just a bit of programming.

Enters Mr Change

The Design



The Verdict?

The Verdict?

The Verdict?

What Had Gone Wrong?

- At first Joe didn't understand what had gone wrong.

What Had Gone Wrong?

- At first Joe didn't understand what had gone wrong.
- It was inheritance that was causing the problem.
  - The `Duck` class defined the default `fly( )` behaviour.
  - This was inherited by *all* `Duck` subclasses.
  - None of the subclasses overrode the behaviour.
  - Therefore all  s had the default `fly( )` behaviour.
  - Including `RubberDucks`.

What Should Joe Do?

What Should Joe Do?

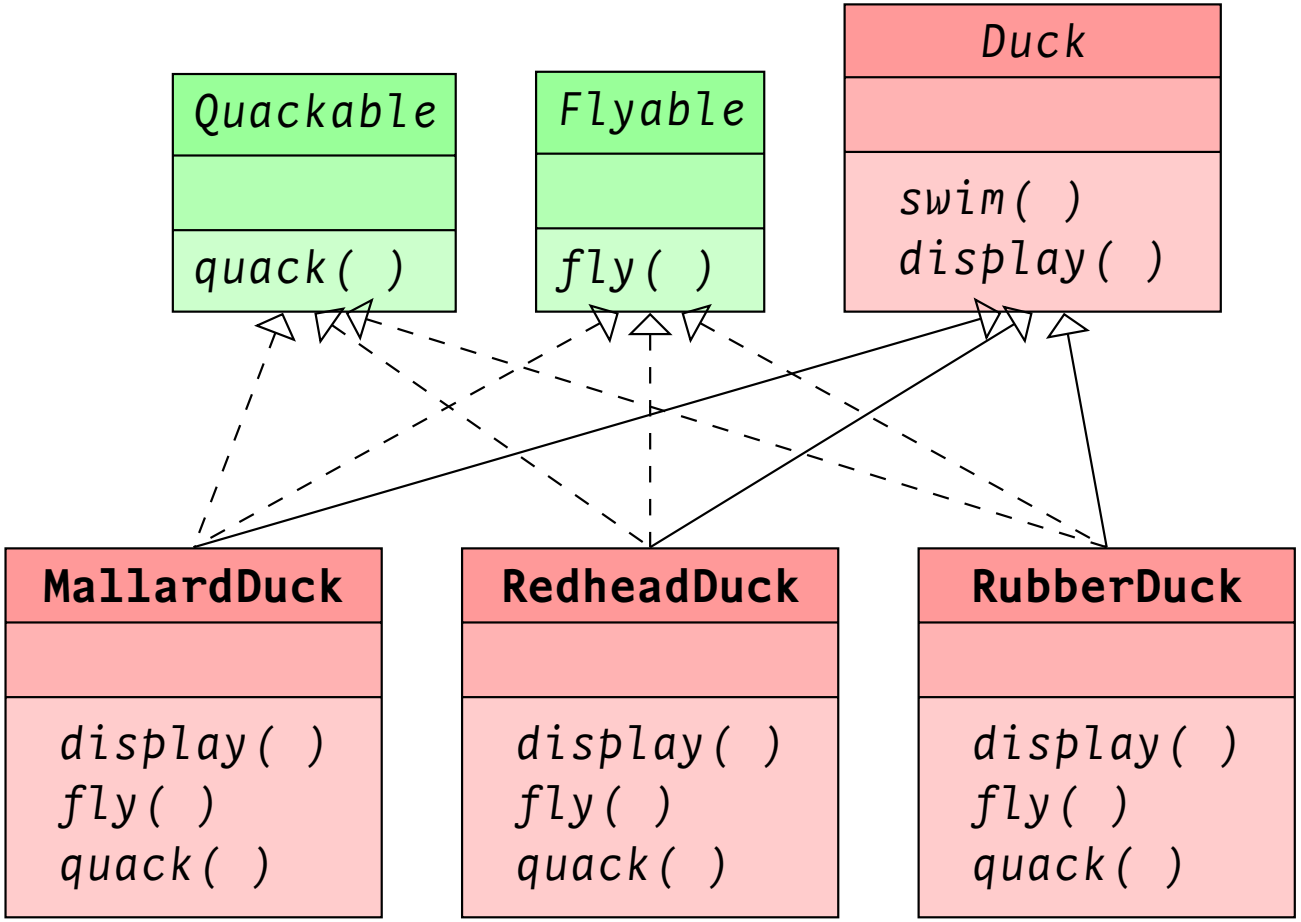
Should he override `fly( )` in the `RubberDuck` Class?

What Should Joe Do?

Should he override `fly( )` in the `RubberDuck` Class?

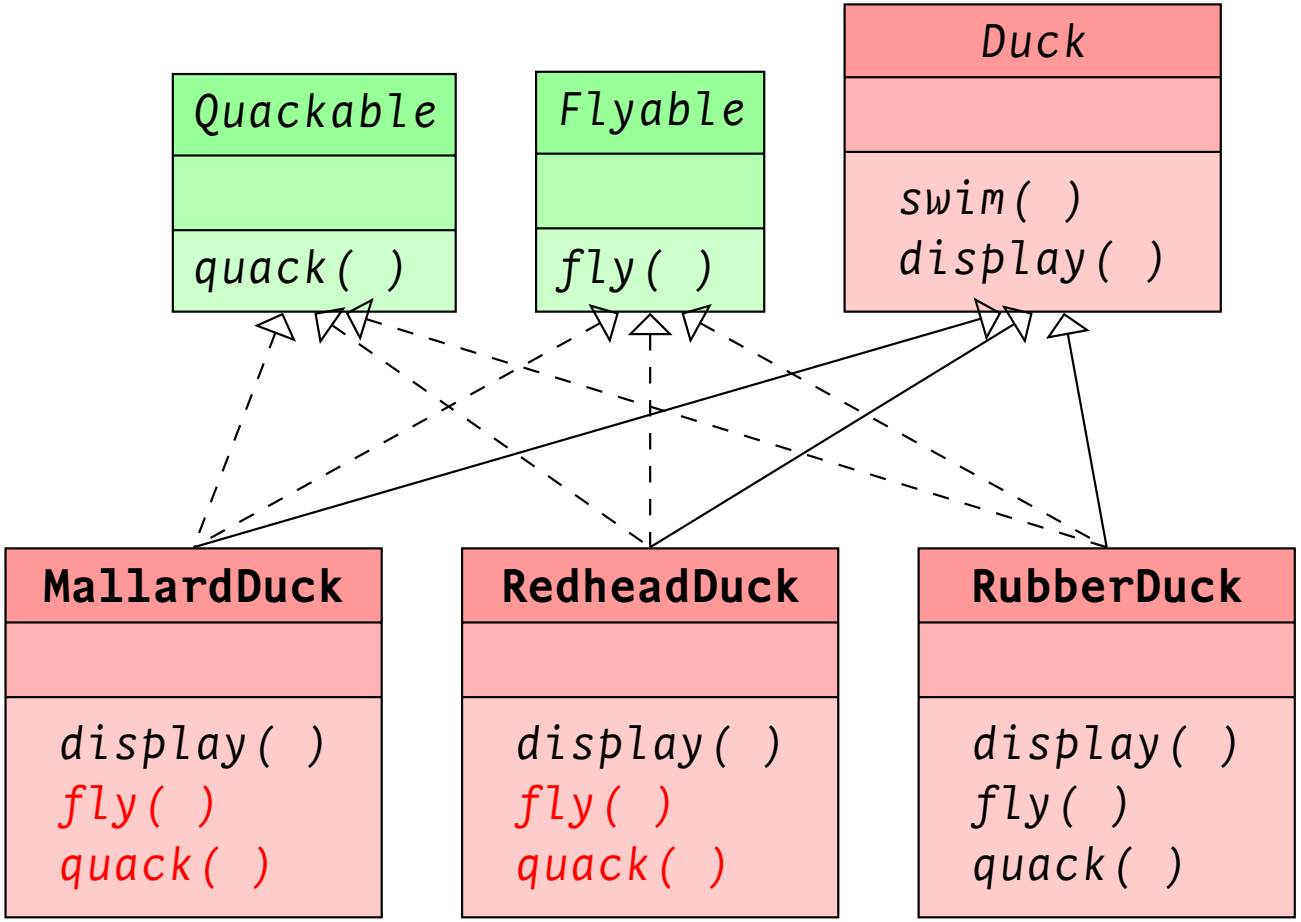
- If he did that he might have to duplicate code later.
  - For example, what if a `WoodenDecoyDuck` was added later?
  - `RubberDuck` and `WoodenDecoyDuck` were almost the same,
    - Yet shared no code....
- Of course he could introduce a common superclass.
  - But that would mean much work.
  - Also there was no guarantee that work would stop there.

Should he Use an Interface?



Should he Use an Interface?

Did Somebody say Code Duplication?



What Joe Really Wants

- Joe really wants software that doesn't change.
- He does realise that change is the only constant.
- Code changes should have little impact on existing code.
- That would save much time rewriting existing code.

First Design Principle

- We've seen that inheritance hasn't worked for Joe.
  - When the (`Duck`) superclass changes this affects all subclasses.
- Interfaces cannot change but they have no implementation:
  - No code reuse.

First Design Principle

Encapsulate what Varies

- We've seen that inheritance hasn't worked for Joe.
  - When the (`Duck`) superclass changes this affects all subclasses.
- Interfaces cannot change but they have no implementation:
  - No code reuse.
- The following design principle may help Joe:

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

First Design Principle

Encapsulate what Varies

- We've seen that inheritance hasn't worked for Joe.
  - When the (`Duck`) superclass changes this affects all subclasses.
- Interfaces cannot change but they have no implementation:
  - No code reuse.
- The following design principle may help Joe:

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

- We implement each aspect class as a behaviour:
  - Implement separate classes for different behaviour.
  - Lets us choose specific behaviour by selecting a specific class.
  - Reusing the implementation comes for free.
  - Separates the implementation: increases flexability.

Encapsulate what Varies

- Most classes *implemented* `Flyable` and `Quackable`.
- This is what caused the code duplication.
- We're going to encapsulate what varies:
  - We separate what varies: `fly( )` and `quack( )` behaviour.
  - We define a `Flyable` interface.
    - Encapsulate each different `fly( )` behaviour as separate class.
  - We also define a `Quackable` interface.
    - Encapsulate each different `quack( )` behaviour as separate class.
  - We reuse the behaviour in the actual `Duck` subclasses.
    - This is done using *delegation*.
    - (It involves a design pattern.)

Second Design Principle

- We need to design classes that implement  behaviour.
- Behaviour is *assigned* to specific `Duck` instance attributes.
  - Assiging behaviour can even be done at runtime.

Second Design Principle

Program to an Interface

- We need to design classes that implement  behaviour.
- Behaviour is *assigned* to specific `Duck` instance attributes.
  - Assiging behaviour can even be done at runtime.
- The following design principle is exactly what we need:

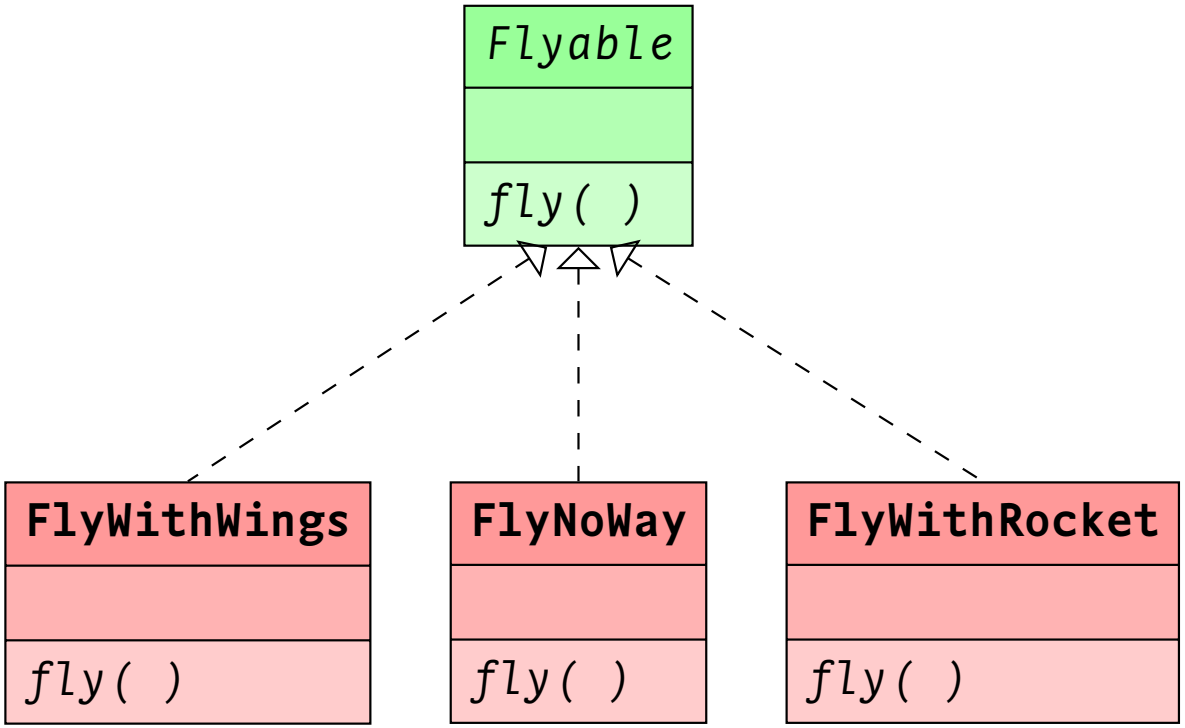
Design Principle

Program to an interface, not to an implementation.

Programming to an Interface

- We use an interface (a supertype) for each behaviour.
  - `Flyable`, `Quackable`, ...
  - Specific classes implement specific behaviours.
  - We use instances of these classes to use the behaviour.
- Before we depended on an *implementation:*
  - Default or overridden class behaviour.
- Now we depend on an interface: an object with a type.
- Clients are now *unaware* of actual type and class of object.
  - This greatly reduces subsystem dependencies.

Implementing the `fly( )` Behaviour



Integrating the Duck Behaviour

Each Duck Delegates the fly( ) and quack( ) Behaviour

Duck
private Flyable flyer private Quackable quacker
public final fly( ) { flyer.fly( ); } public final quack( ) { quacker.quack( ); } public swim( ) public display( )

Implementing the MallardDuck

Java

```
public class MallardDuck extends Duck {
    public MallardDuck( ) {
        super( new SqueekQuack( ), new FlyWithWings( ) );
    }

    @Override
    public void display( ) {
        System.out.println( "MallardDuck here...." );
    }
}
```

Implementing the MutableDuck

Java

```
public class MutableDuck extends Duck {
    public MutableDuck( ) {
        super( new SqueekQuack( ), new FlyWithWings( ) );
    }

    public void setQuackBehaviour( Quackable quacker ) {
        // Assumes quacker is public/not final now.
        this.quacker = quacker;
    }

    public void setFlyBehaviour( Flyable flyer ) {
        // Assumes flyer is public/not final now.
        this.flyer = flyer;
    }

    @Override
    public void display( ) {
        System.out.println( "MutableDuck here...." );
    }
}
```

Inheritance versus Object Composition

Inheritance: Lets us create subclasses: *white-box reuse*.

- Subclass inherits superclass behaviour.
- Subclasses can override superclass behaviour.
- You get code reuse for free.
- You cannot change behaviour at runtime.
- Violates encapsulation.
  - Subclass may rely on superclass implementation.
  - Subclass may break when superclass is changed.

Composition: Lets you *compose* classes: *black-box reuse*.

- A client class may use an object.
- You get code reuse but it takes more effort.
- Lets you change behaviour at runtime.
- Respects encapsulation.
  - Helps encapsulated classes focus on a single task.

Has-A can be better than Is-A

- In our new design we rely on Has-A (more then on Is-A):
  - Each `Duck` has-a `flyer`, and
  - Each `Duck` has-a `quacker`.
- "Has-A" lets us implement behaviour by *composing* classes.
- The result is a more flexible design:
  - It lets us encapsulate behaviour.
  - We can change behaviour at runtime.

Third Design Principle

- In our new design we rely on Has-A (more then on Is-A):
  - Each `Duck` has-a `flyer`, and
  - Each `Duck` has-a `quacker`.
- "Has-A" lets us implement behaviour by *composing* classes.
- The result is a more flexible design:
  - It lets us encapsulate behaviour.
  - We can change behaviour at runtime.

Design Principle

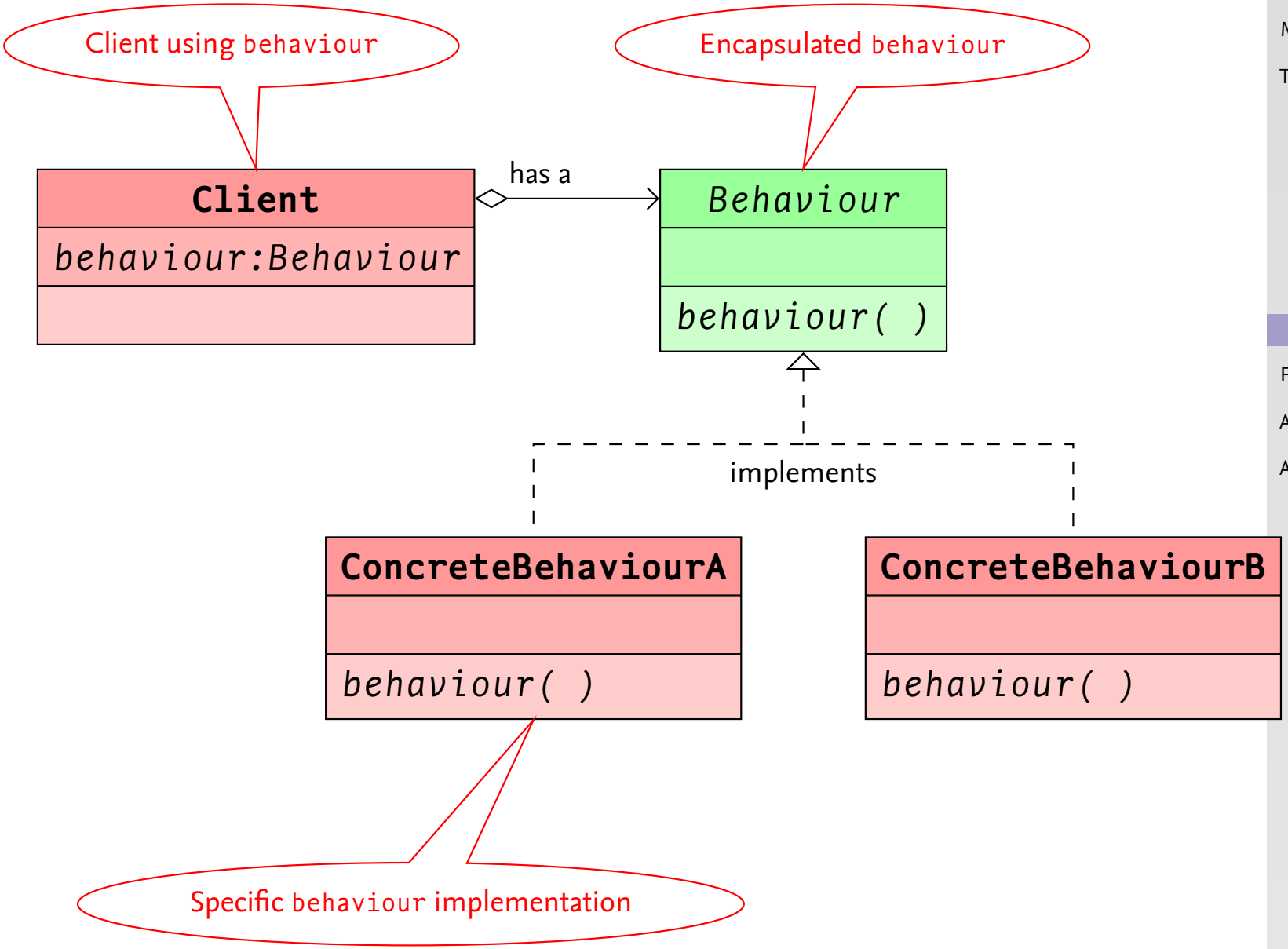
Favour Composition over Inheritance

The Strategy Pattern

Design Pattern

- *The Strategy Pattern:*
  - *Defines a class of algorithms;*
  - *Encapsulates each algorithm; and*
  - *Makes them interchangeable.*
- *Lets the algorithms vary independently from clients using it [Gamma et al. 2008].*

Finally: Strategy Pattern



For Monday

- Study the presentation.
- Implement the SimuDuck™ application (optional).
- Study [Horstmann 2013, Sections 9.1–9.4].

Acknowledgements

- The first part of the lecture is based on [Sierra, and Bates 2004].
- The second part is based on [Freeman, and Freeman 2005].

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.