CMPT 373
Software Development Methods

Using Inheritance (and Not Abusing It)

Nick Sumner
wsumner@sfu.ca

with some material from Bertrand Meyer

What is inheritance?

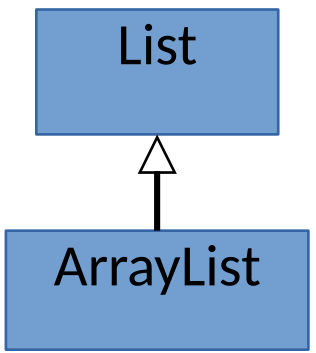
- You should *already* be comfortable with inheritance

What is inheritance?

- You should *already* be comfortable with inheritance
- Review of ***inheritance***:

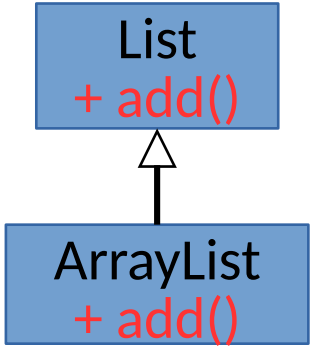
What is inheritance?

- You should *already* be comfortable with inheritance
- Review of ***inheritance***:
  - Create a *new* type based on an *existing* type



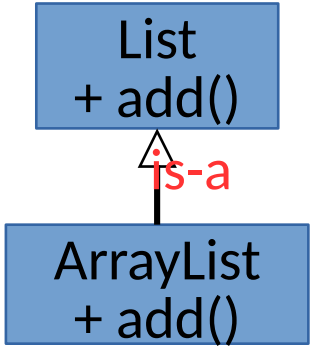
What is inheritance?

- You should *already* be comfortable with inheritance
- Review of ***inheritance***:
  - Create a *new* type based on an *existing* type
  - Shares properties and behaviors with the new type



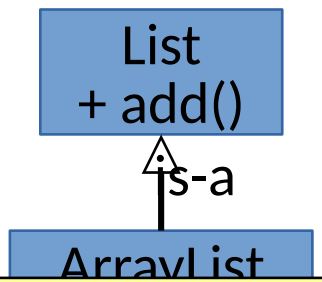
What is inheritance?

- You should *already* be comfortable with inheritance
- Review of ***inheritance***:
  - Create a *new* type based on an *existing* type
  - Shares properties and behaviors with the new type
  - *Can* establish a subtyping relationship



What is inheritance?

- You should *already* be comfortable with inheritance
- Review of ***inheritance***:
  - Create a *new* type based on an *existing* type
  - Shares properties and behaviors with the new type
  - *Can* establish a subtyping relationship



Note: This conflates ***inheritance*** & ***subtyping***,
but it is probably what you know (from Java/C++).

What does good inheritance look like? (Review)_____

- Initial guidelines:

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived

Derived is ***substitutable*** for Base

Base
A foo(B b)

Derived
C foo(D d)

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general

Base
A foo(B b)

Derived
C foo(D d)

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general

B <: D

Base A foo(B b)	Derived C foo(D d)

Arguments are ***contravariant***

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained

Base A foo(B b)	Derived C foo(D d)

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained

C <: A

Base	Derived
A foo(B b)	C foo(D d)

Return types are ***covariant***

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - Preconditions are not stronger

```
assert(x > 0)
```

Base
A foo(B b)

```
assert(x != 0)
```

Derived
C foo(D d)

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - Preconditions are not stronger
    - Postconditions are not weaker

Base A foo(B b)	Derived C foo(D d)
`assert(result != 0)`	`assert(result > 0)`

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - Preconditions are not stronger
    - Postconditions are not weaker
    - Invariants must still hold

Base
A foo(B b)

Derived
C foo(D d)

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - Preconditions are not stronger
    - Postconditions are not weaker
    - Invariants must still hold

Base
A foo(B b)

Derived
C foo(D d)

How does the Liskov Substitution Principle relate to coupling from using inheritance?

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
  - Be wary of implementation inheritance

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
  - Be wary of implementation inheritance
    - Hierarchies delocalize code, yielding a *yo-yo effect*

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
  - Be wary of implementation inheritance
    - Hierarchies delocalize code, yielding a *yo-yo effect*
    - Ambiguous overrides break encapsulation

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
  - Be wary of implementation inheritance
    - Hierarchies delocalize code, yielding a *yo-yo effect*
    - Ambiguous overrides break encapsulation

```
class Parent {
  virtual void foo() { bar(); }
  virtual void bar() {}
};
```

[Bloch, “Effective Java”]

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
  - Be wary of implementation inheritance
    - Hierarchies delocalize code, yielding a *yo-yo effect*
    - Ambiguous overrides break encapsulation

```
class Parent {
  virtual void foo() { bar(); }
  virtual void bar() {}
};
```

```
class Child : public Parent {
public:
  virtual void bar() { foo(); }
};
```

[Bloch, “Effective Java”]

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
  - Be wary of implementation inheritance
    - Hierarchies delocalize code, yielding a *yo-yo effect*
    - Ambiguous overrides break encapsulation

```
class Parent {
  virtual void foo() { bar(); }
  virtual void bar() {}
};
```

```
class Child : public Parent {
public:
  virtual void bar() { foo(); }
};
```

[Bloch, “Effective Java”]

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
  - Be wary of implementation inheritance
    - Hierarchies delocalize code, yielding a *yo-yo effect*
    - Ambiguous overrides break encapsulation

```
class Parent { public:
};class Child : public Parent {
  public:
    virtual void bar() { foo(); }
  };
```

Non Virtual Interfaces (NVI) help clarify & are common in C++.

[Bloch, "Effective Java"]

```
class Parent {
public:
  void foo() { barImpl(); }
  void bar() { barImpl(); }
private:
  virtual void barImpl() = 0;
};
```

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
  - Be wary of implementation inheritance
    - Hierarchies delocalize code, yielding a *yo-yo effect*
    - Ambiguous overrides break encapsulation

```
class Parent {
public:
  void foo() { barImpl(); }
  void bar() { barImpl(); }
private:
  virtual void barImpl() = 0;
};
```

Non Virtual Interfaces (NVI) help clarify & are common in C++.

```
};class Child : public Parent {
public:
                                    }
};
```

Other patterns help even more...

[Bloch, "Effective Java"]

What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
  - Be wary of implementation inheritance

Here endeth the review

So let’s try it out…

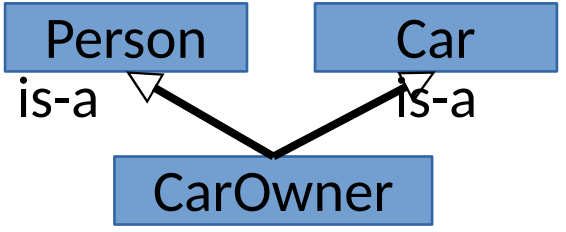
Note: We will go from absurd to practical

So let’s try it out…

- Suppose we want to model a person who owns a car…

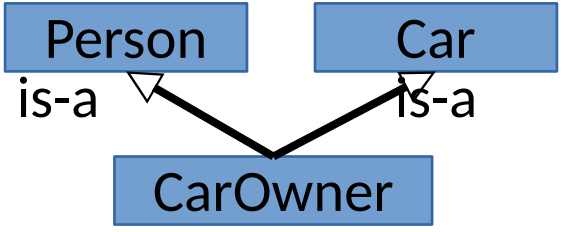
So let’s try it out…

- Suppose we want to model a person who owns a car…



So let's try it out...

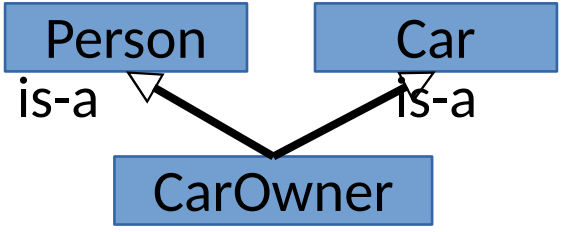
- Suppose we want to model a person who owns a car...




```
class CarOwner : public Person, Car
  { };
```

So let's try it out…

- Suppose we want to model a person who owns a car…

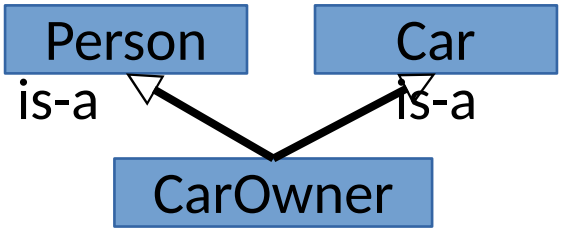



```
class CarOwner : public Person, Car
  { };
```

Is this good or bad?
Why?

So let's try it out…

- Suppose we want to model a person who owns a car…



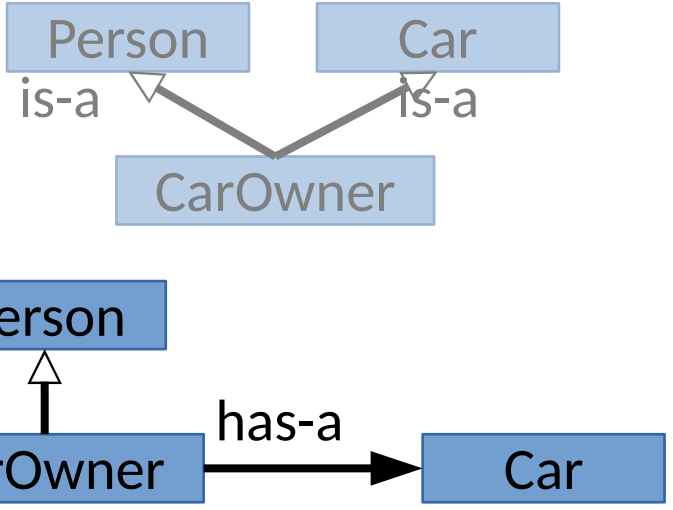

```
class CarOwner : public Person, Car
  { };
```

Is this good or bad?
Why?

How could you make it better?

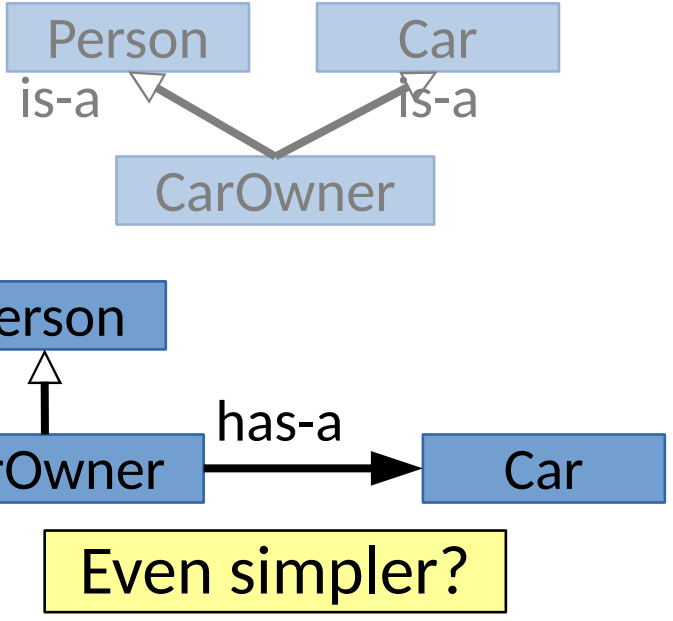
So let's try it out...

- Suppose we want to model a person who owns a car...



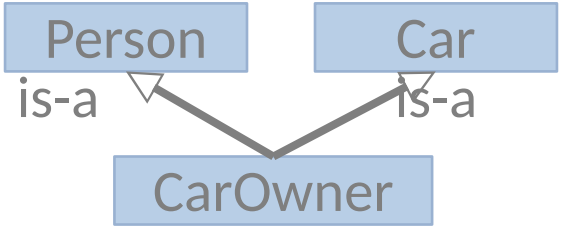
So let's try it out...

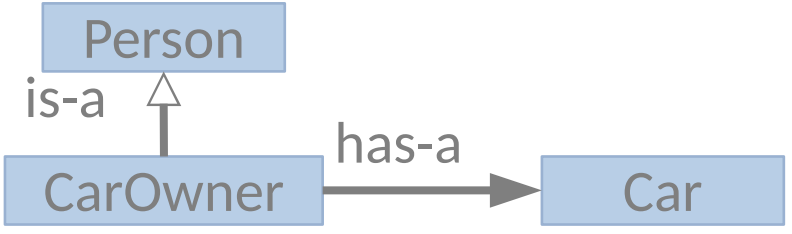
- Suppose we want to model a person who owns a car...

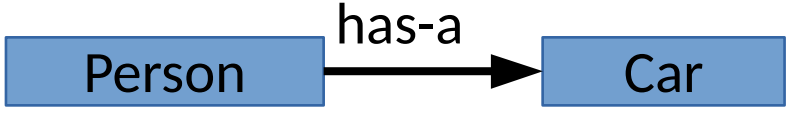


So let's try it out...

- Suppose we want to model a person who owns a car...

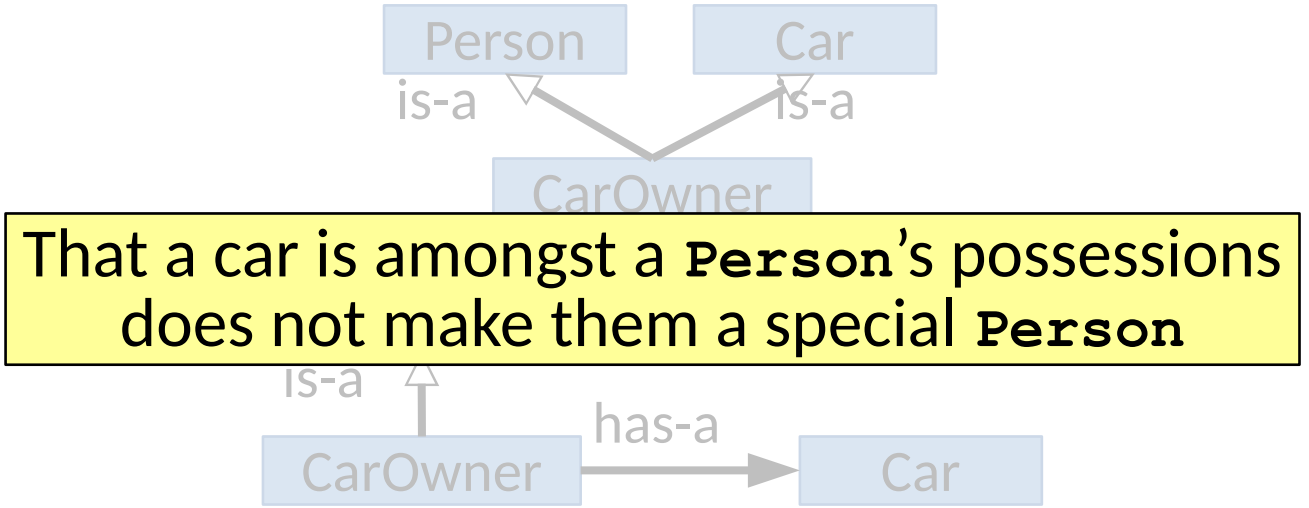


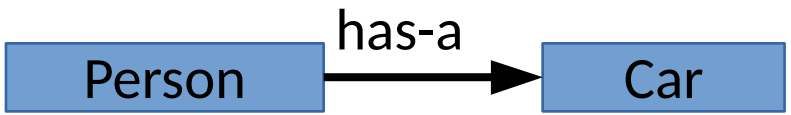




So let's try it out...

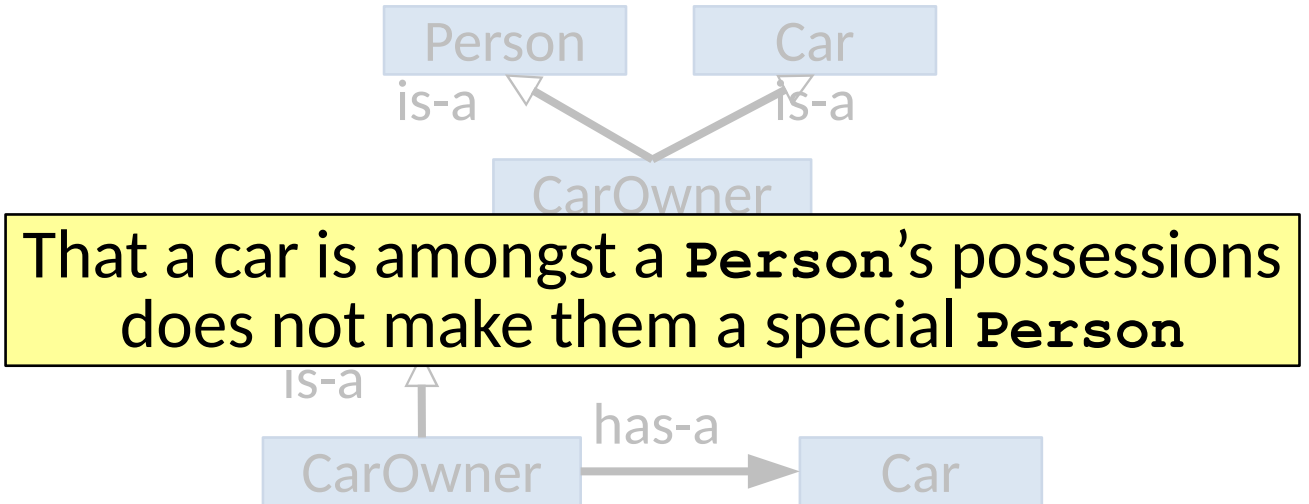
- Suppose we want to model a person who owns a car...





So let's try it out...

- Suppose we want to model a person who owns a car...



This absurd example captures common, subtle mistakes

So why is inheritance hard?

- Do the *LSP* and *has-a* relationships *unambiguously* tell us how to apply inheritance?

So why is inheritance hard?

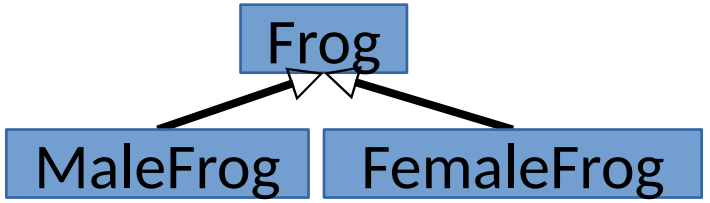
- Do the *LSP* and *has-a* relationships *unambiguously* tell us how to apply inheritance?

`Frog`s can be male or female

So why is inheritance hard?

- Do the *LSP* and *has-a* relationships *unambiguously* tell us how to apply inheritance?

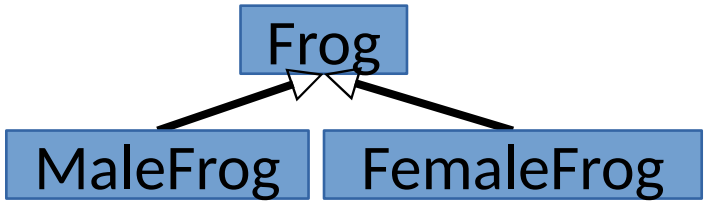
`Frog`s can be male or female



So why is inheritance hard?

- Do the *LSP* and *has-a* relationships *unambiguously* tell us how to apply inheritance?

`Frog`s can be male or female



So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!

So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

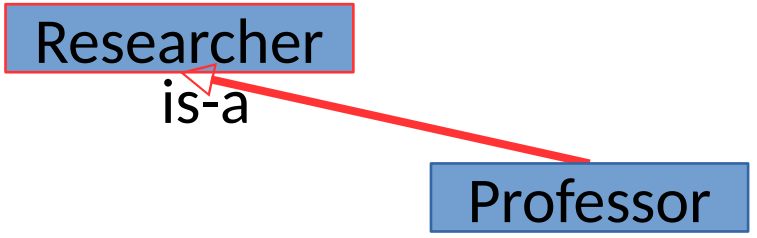
So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

Professor

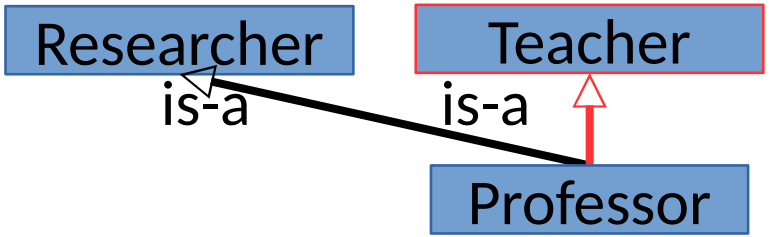
So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components



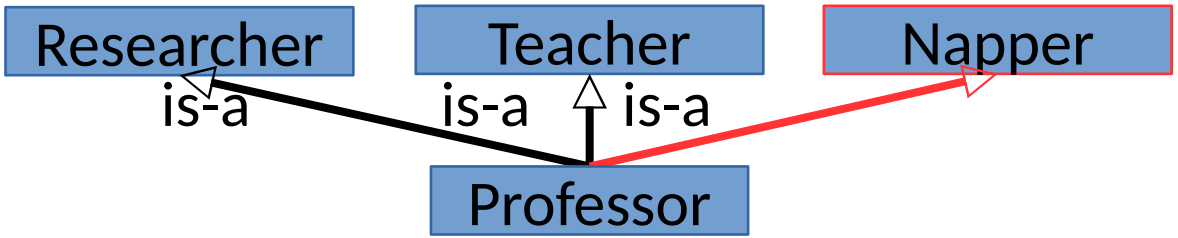
So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components



So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

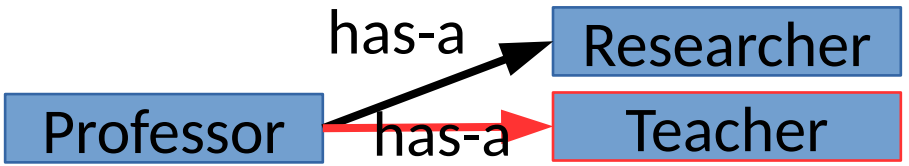


So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

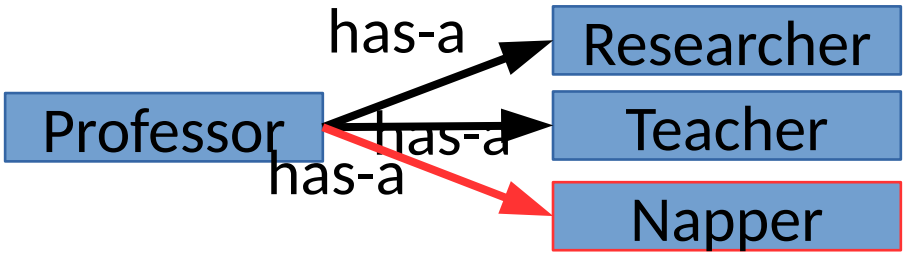
So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components



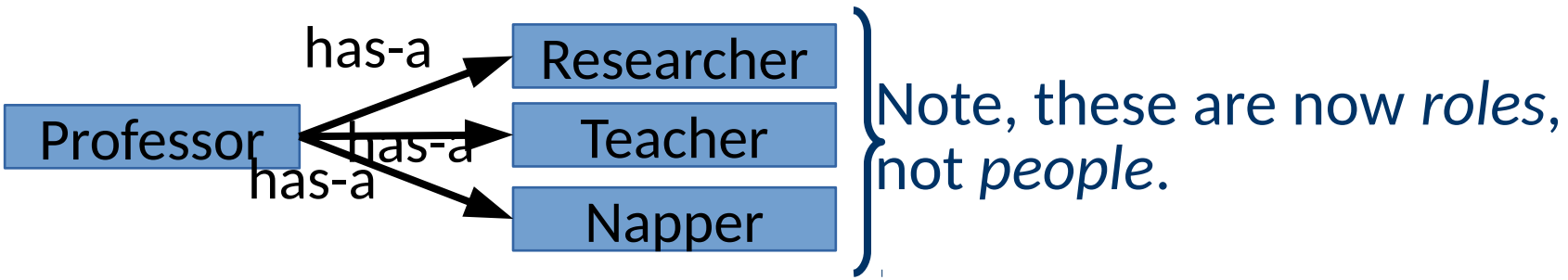
So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components



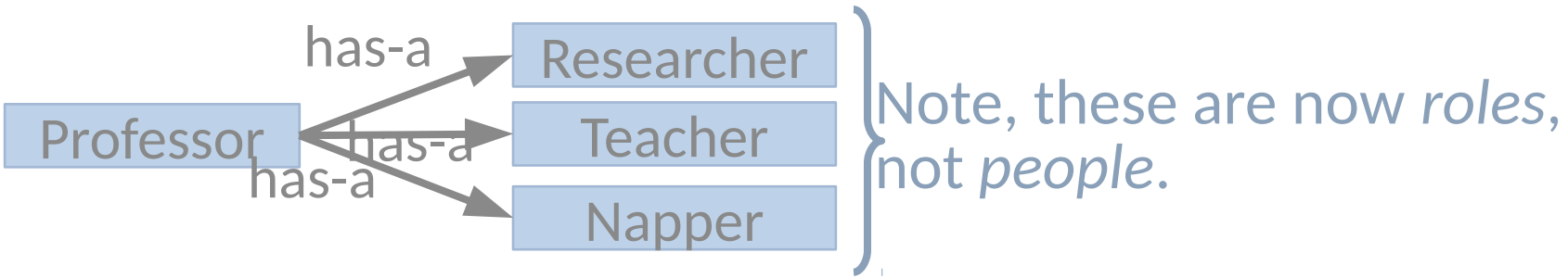
So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components



So why is inheritance hard?

- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components



- **Whenever *is-a* applies, you must still make a decision**

Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Inheritance often *precludes* it

Choosing is-a or has-a

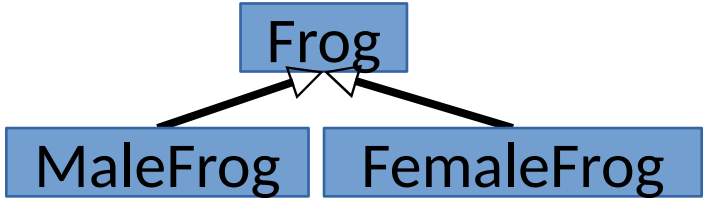
- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often *simplifies* it

Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic

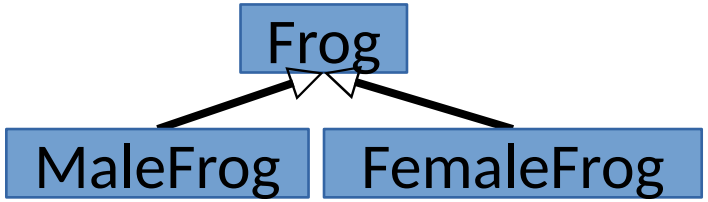
Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic



Choosing is-a or has-a

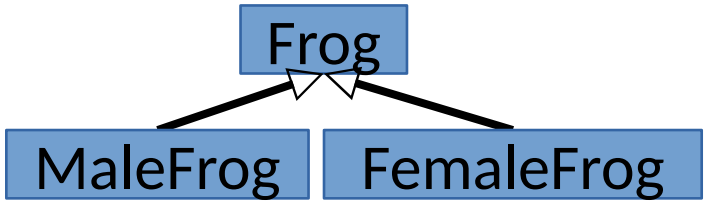
- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic



`Frog`s and other animals
can spontaneously change sex!

Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic



`Frog`s and other animals
can spontaneously change sex!

Knowing in advance is hard.
Composition is flexible & adapts to requirements.

Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic

- Guide 2: Might the type be used polymorphically?
  - Composition does not intrinsically aid it

Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic

- Guide 2: Might the type be used polymorphically?
  - Composition does not intrinsically aid it
  - Inheritance enables it

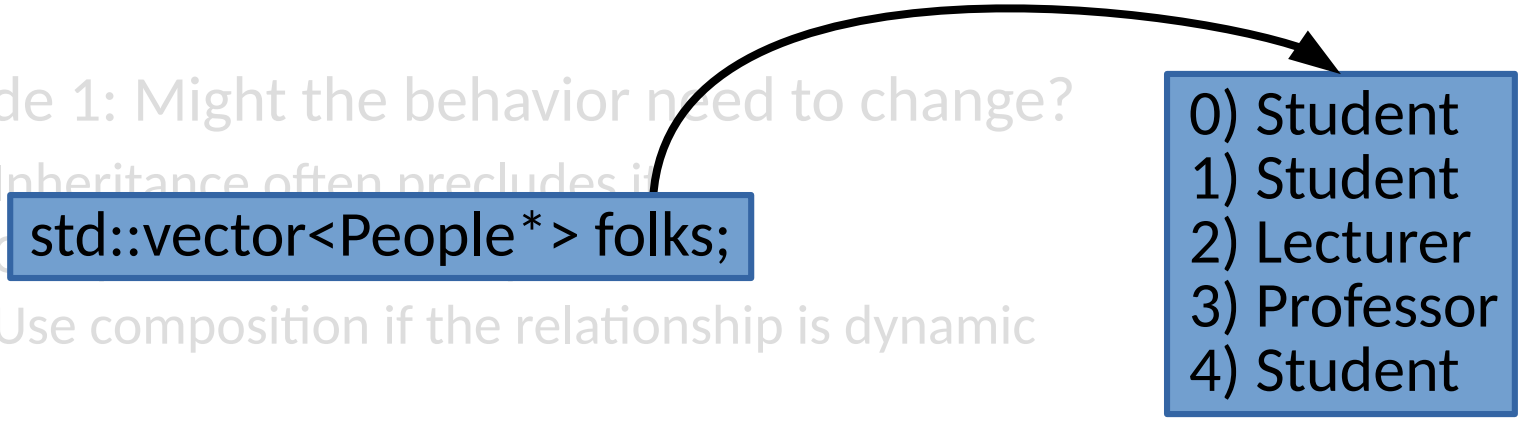
Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic

- Guide 2: Might the type be used polymorphically?
  - Composition does not intrinsically aid it
  - Inheritance enables it
  - Consider inheritance when a reference to a general type may point to a more specific one.

Choosing is-a or has-a

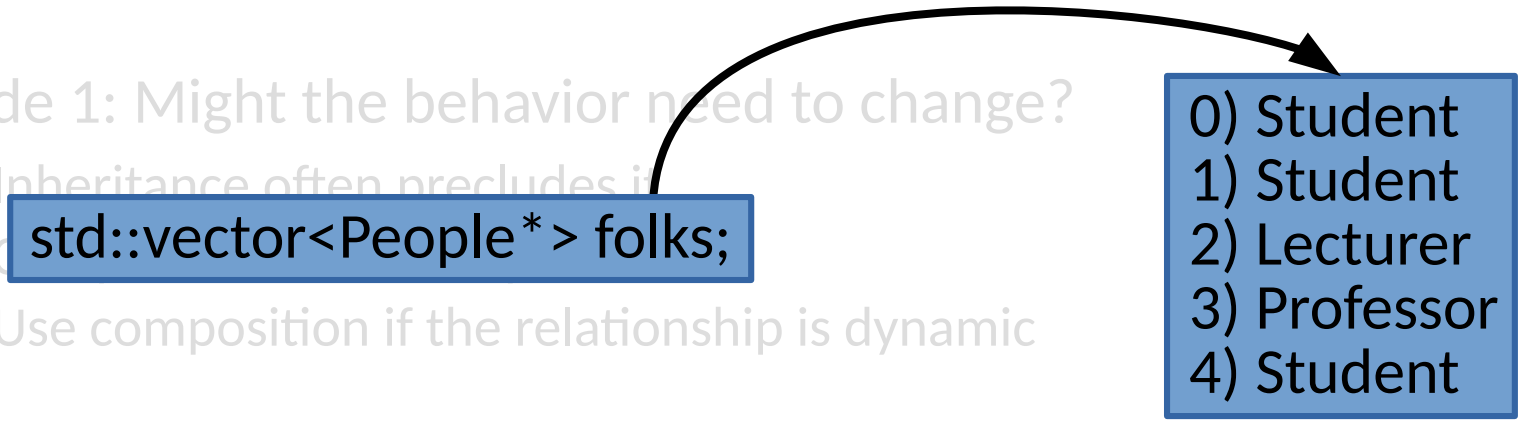
- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition enables it
  - Use composition if the relationship is dynamic



- Guide 2: Might the type be used polymorphically?
  - Composition does not intrinsically aid it
  - Inheritance enables it
  - Consider inheritance when a reference to a general type may point to a more specific one.

Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition enables it
  - Use composition if the relationship is dynamic



- Guide 2: Might the type be used polymorphically?
  - Composition does not intrinsically aid it
  - Inheritance enables it
  - Consider inheritance when a reference to a general type may point to a more specific one

We will revisit this in the context of ***algebraic data types***.

So let's try it out…

- I need
  - Many different types of animals.

This should sound familiar…

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

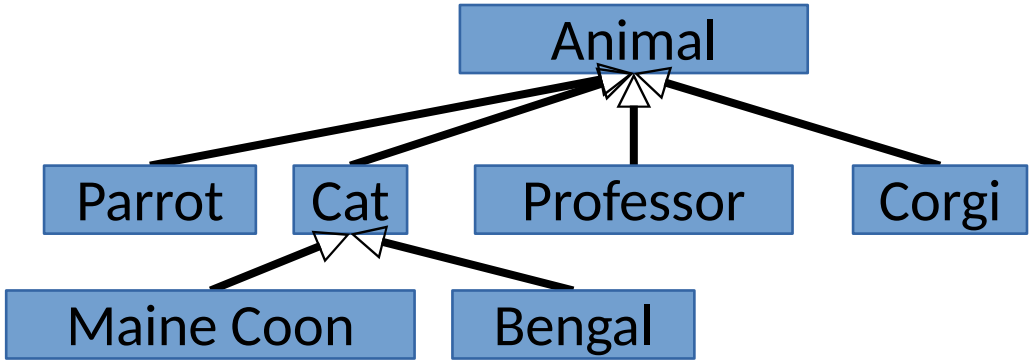
So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

What does my design look like based on the rules?

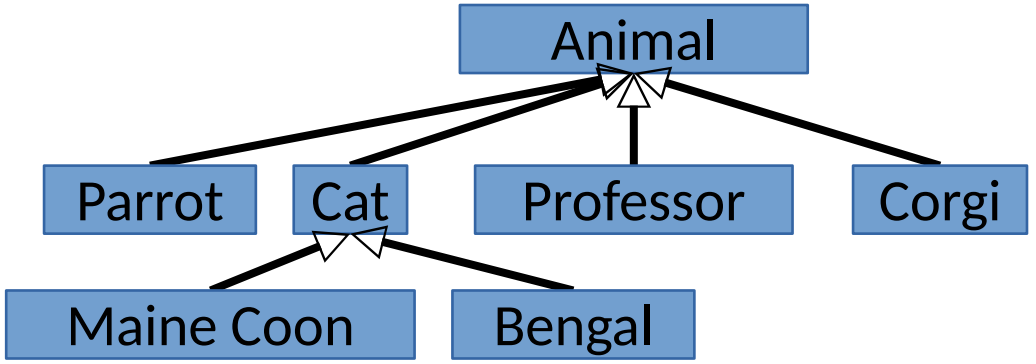
So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.



So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

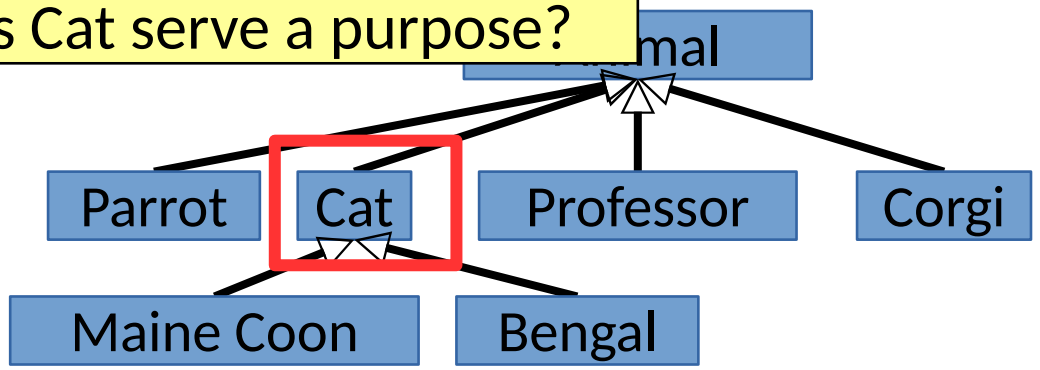


Is this good?

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

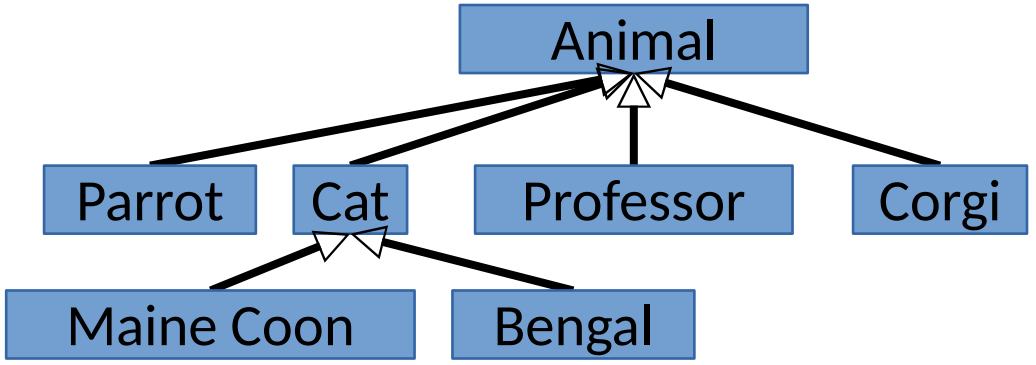
Does Cat serve a purpose?



Is this good?

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

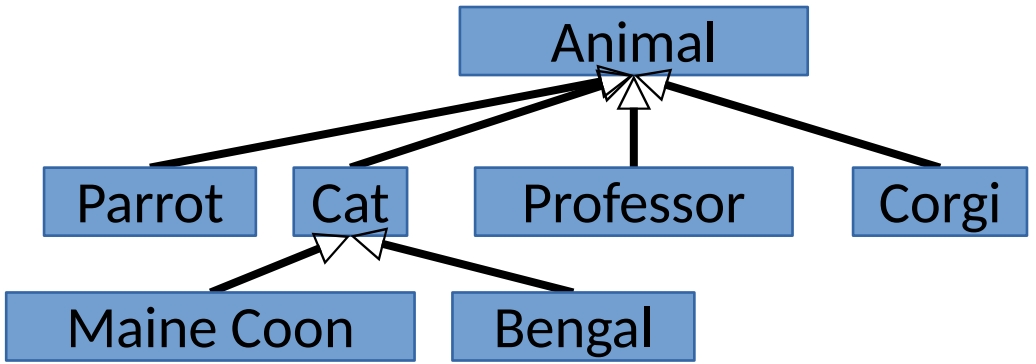


Is this good?

Does it achieve reuse?

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.



Is this good?

Does it achieve reuse?

What if I want a new Animal at run time?

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

If someone on my team did this multiple times,
I would fire them.

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

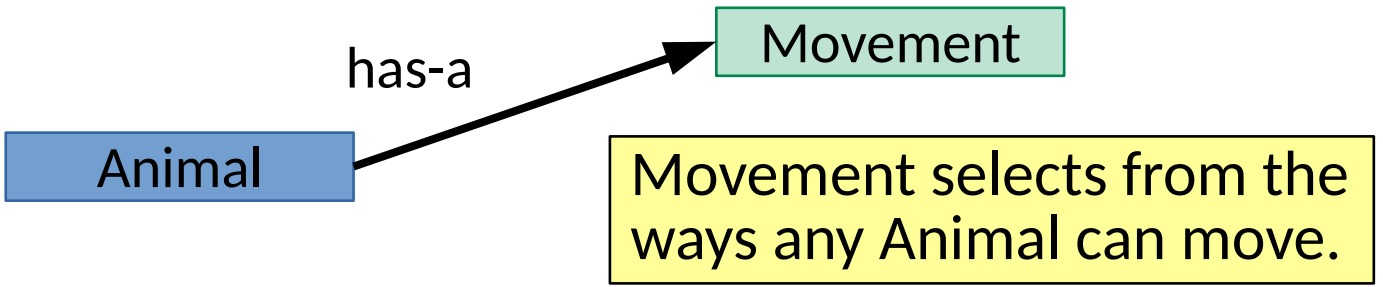
Recall: identify & isolate change

So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

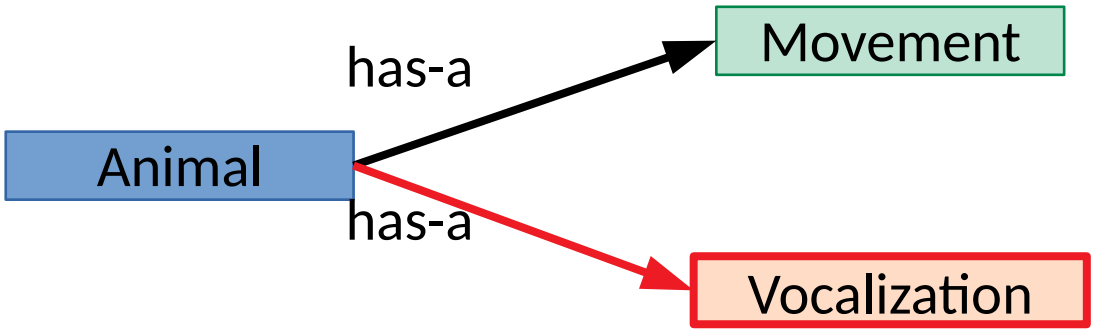


So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

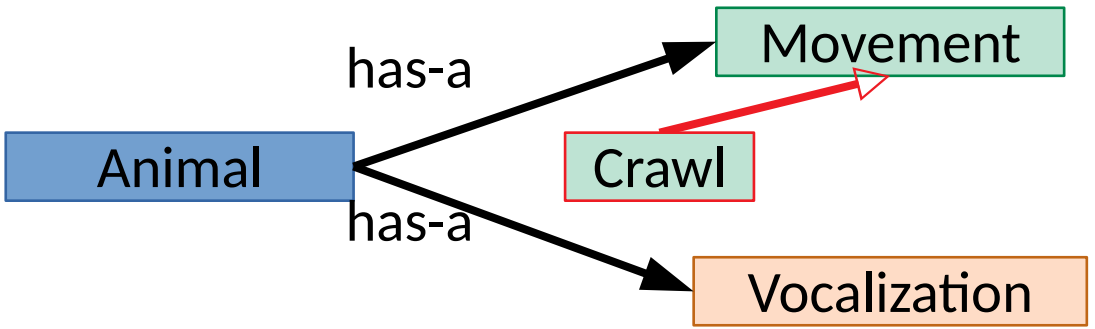


So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

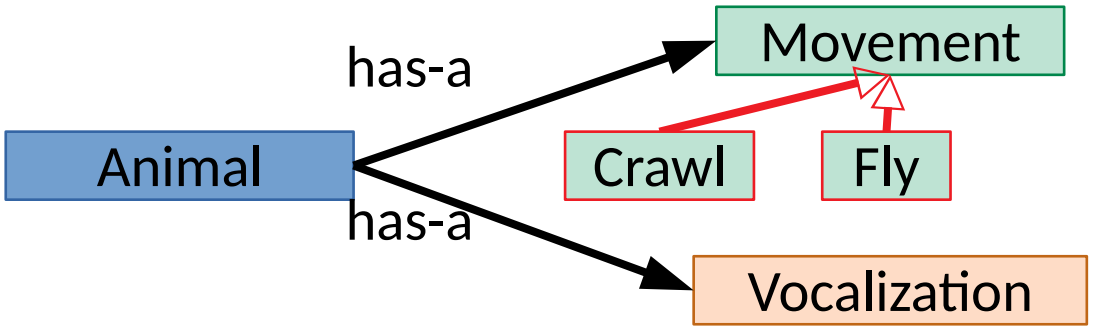


So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

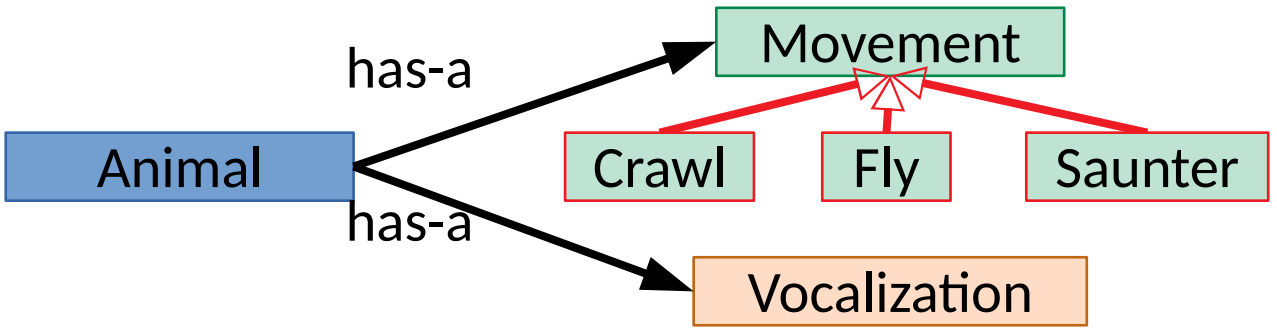


So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

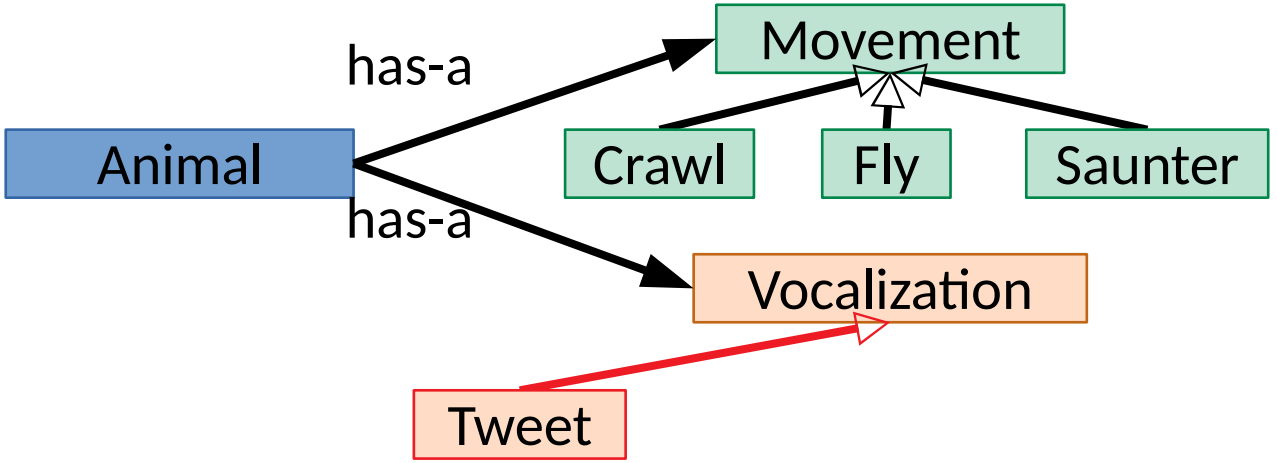


So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

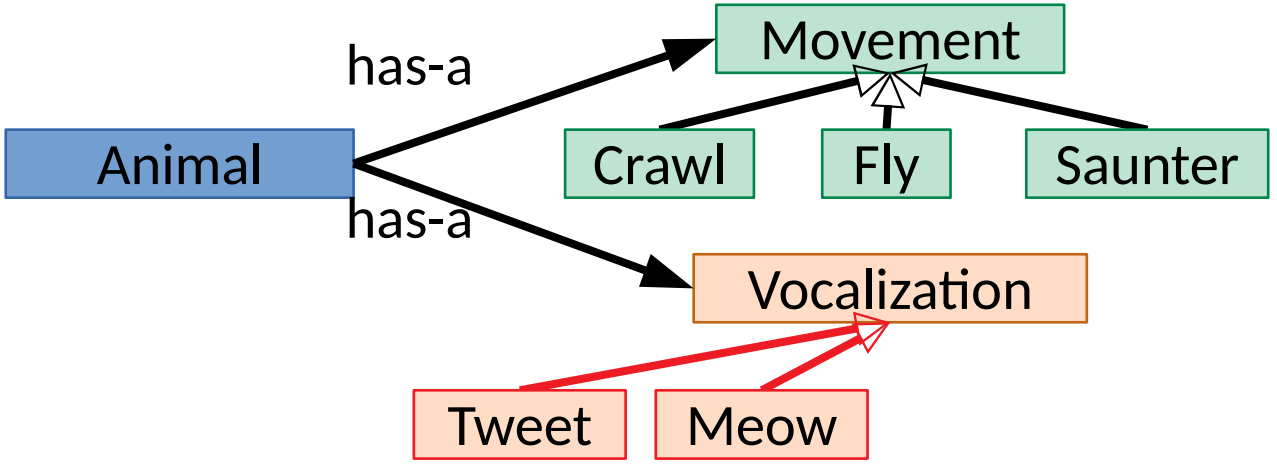


So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

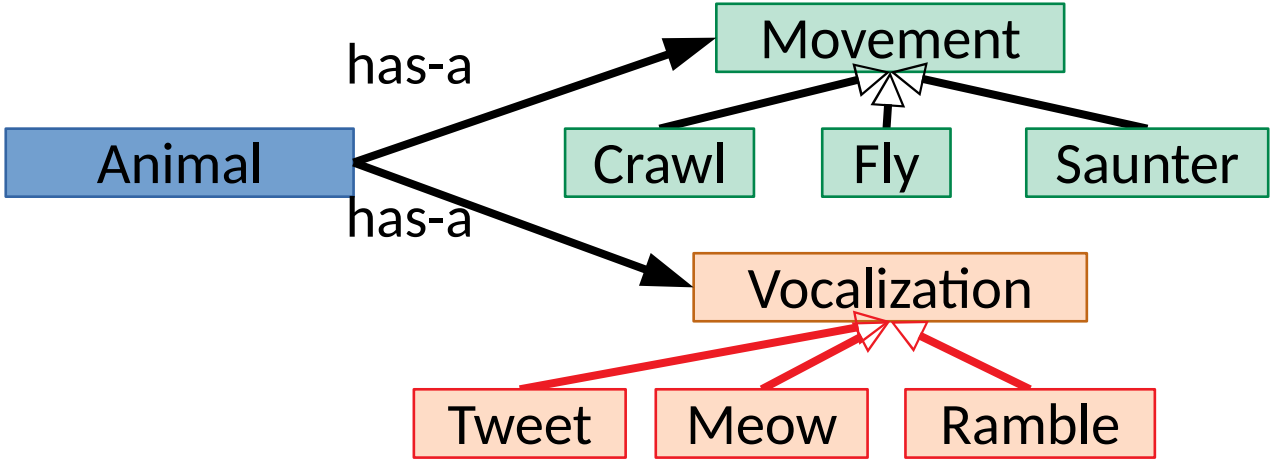


So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

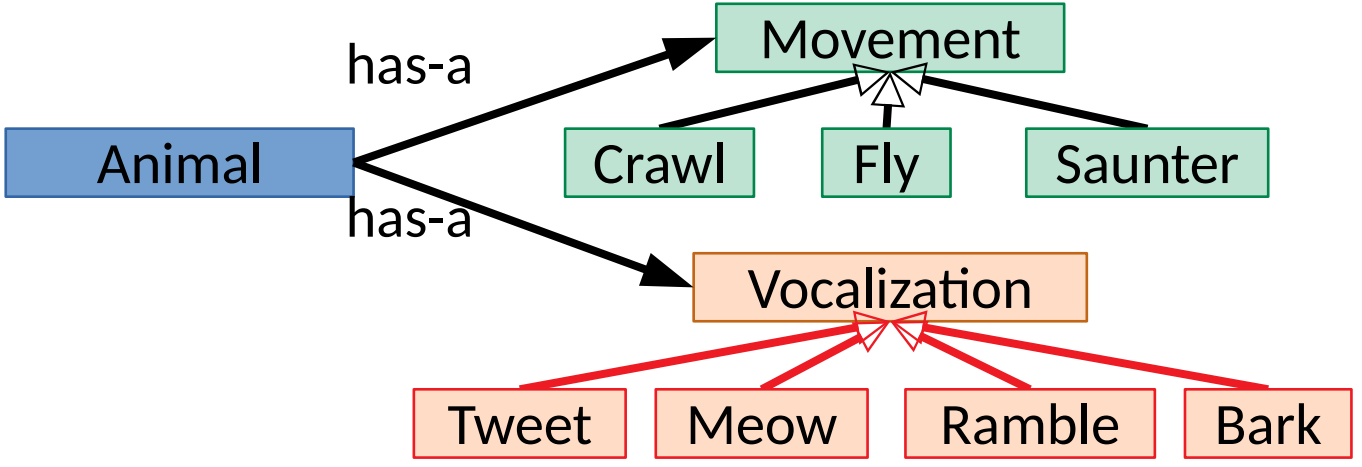


So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change



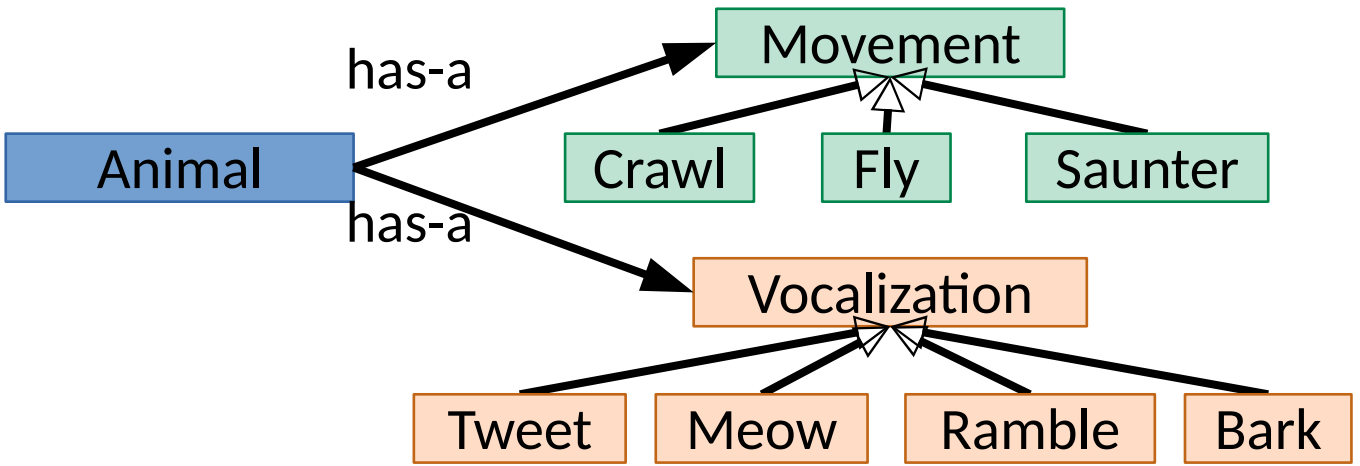
So let's try it out...

- I need
  - Many different types of animals.
  - Each should be able to **`move()`** and **`speak()`**.
  - An **`Animal&`** should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

```
class Animal {
  Movement& m;
  void move() {
    m.move();
  }
};
```



So let’s try it out…

- So let’s try it out…**(!)**

Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal

Shallow, fine grained inheritance

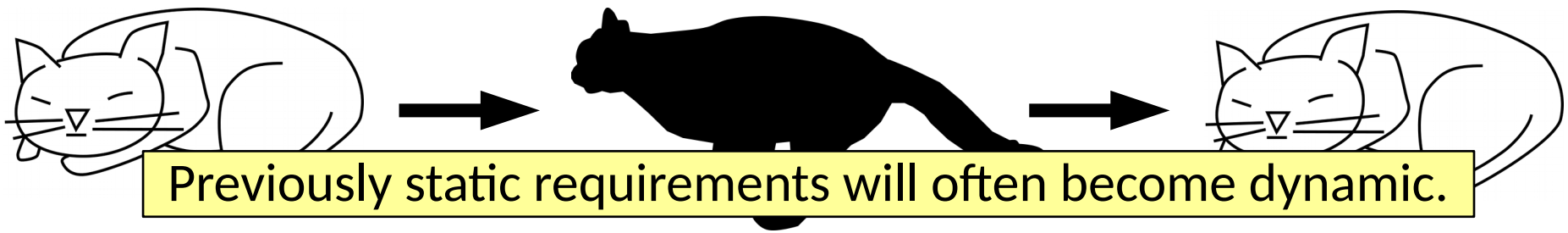
- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies

Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies
- Enables dynamic selection & configuration of which policies are desired
  - e.g. A **`Cat`** may start out **`Stationary`**, then **`Run`**, then be **`Stationary`**

Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies
- Enables dynamic selection & configuration of which policies are desired
  - e.g. A **`Cat`** may start out **`Stationary`**, then **`Run`**, then be **`Stationary`**



Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies
- Enables dynamic selection & configuration of which policies are desired
  - e.g. A `Cat` may start out `Stationary`, then `Run`, then be `Stationary`
- Directly identifies & addresses risks of change in class design

Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies
- Enables dynamic selection & configuration of which policies are desired
  - e.g. A `Cat` may start out `Stationary`, then `Run`, then be `Stationary`
- Directly identifies & addresses risks of change in class design
- We will see shortly how this interacts with other forms of polymorphism

Summary

- Inheritance is a powerful tool, but it requires care.

Summary

- Inheritance is a powerful tool, but it requires care.
- Good inheritance simplifies design & both expresses and isolates regions of change

Summary

- Inheritance is a powerful tool, but it requires care.
- Good inheritance simplifies design & both expresses and isolates regions of change
- There is no best design. Be pragmatic.

Summary

- Inheritance is a powerful tool, but it requires care.
- Good inheritance simplifies design & both expresses and isolates regions of change
- There is no best design. Be pragmatic, but smart.