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

![Diagram of inheritance relationship between List and ArrayList]
What does good inheritance look like?

- Two common guidelines:
What does good inheritance look like?

- Two common guidelines:
  - Prefer composition to inheritance
What does good inheritance look like?

- Two common guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\phi$ is true for the base, then $\phi$ is true for the derived
What does good inheritance look like?

- Two common guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\phi$ is true for the base, then $\phi$ is true the derived

Derived is *substitutable* for Base
What does good inheritance look like?

- Two common guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general
What does good inheritance look like?

- **Two common guidelines:**
  - Prefer composition to inheritance
  - **Liskov Substitution Principle**
    - If $\phi$ is true for the base, then $\phi$ is true for the derived
    - Arguments in the subtype may be more general
    - **Return values** in the subtype may be more constrained

Base

A \texttt{foo}(B \ b)

Derived

C \texttt{foo}(D \ d)
What does good inheritance look like?

- Two common guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ 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)       assert(x != 0)
```

```
Base  A foo(B b)
Derived C foo(D d)
```
What does good inheritance look like?

- Two common guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ 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

```plaintext
assert(result != 0)  assert(result > 0)
```
What does good inheritance look like?

- Two common guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If \( \phi \) is true for the base, then \( \phi \) 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?

- Two common guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true for 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

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

```
Person  Car
  ___/  \___
  \    /   \    \\
   is-a  is-a
        CarOwner
```
So let’s try it out...

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

```java
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...

```
Person is-a Car
```

```
CarOwner has-a Car
```

```
Person is-a CarOwner
```

```
CarOwner has-a Car
```
So let’s try it out...

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

```
Person  Car
  is-a   is-a
       CarOwner

Person  CarOwner
  is-a

CarOwner  Car
  has-a

Even simpler?
```
So let’s try it out...

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

```
Person  Car
  is-a  is-a
        
CarOwner
```

Even simpler?

```
Person  Car
  has-a
```

```
Person  Car
  has-a
```
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?
- 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
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

Diagram:

- **Researcher**
- **Professor**

  `is-a`

  Graph:

  ```
  Researcher --> 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

Note, these are now *roles*, not *people*. 
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

Note, these are *roles*, not *people*. 
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 \textit{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

- **Guide 2:** Might the type be used polymorphically?
  - Composition doesn’t aid it (but it may be unnecessary...)

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 doesn’t aid it (but it may be unnecessary...)
  - 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 doesn’t aid it (but it may be unnecessary...)
  – Inheritance enables it
  – Use inheritance if a reference to a general type may point to a more specific one.
So let’s try it out...

- I need
  - Many different types of animals.
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.
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?

Parrot  Cat  Professor  Corgi
Maine Coon  Bengal

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.
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?

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

Animal

has-a Movement
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

```
Animal
  has-a Movement
  has-a Vocalization
  Movement
    has-a Crawl
    has-a Fly
    has-a Saunter

I need
- Many diferent typmes of anieals.
- 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.

Can we do better?

Recall:
identify & isolate change

```
Animal
    has-a
    Movement
        Crawl
        Fly
        Saunter
    has-a
    Vocalization
        Tweet
        Meow
        Ramble
        Bark
```
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?

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

  Movement m;
  void move() {
    m.move();
  }

  // Can we do better?
  // Identify & isolate changes
```
How about a simpler case... 

- I need
  - Frogs
  - Both male and female frogs
How about a simpler case...__________

- I need
  - Frogs
  - Both male and female frogs

I can think of at least 3 designs.
How about a simpler case... 

- I need
  - Frogs
  - Both male and female frogs
How about a simpler case...

- I need
  - Frogs
  - Both male and female frogs
How about a simpler case...

- I need
  - Frogs
  - Both male and female frogs
How about a simpler case...

- I need
  - Frogs
  - Both male and female frogs

Use inheritance when it is significant to the intrinsic behavior
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.