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

```
List + add()
<--- s-a
ArrayList + add()
```
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 $\phi$ is true for the base, then $\phi$ is true the derived
What does good inheritance look like?

- Initial 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?

- Initial 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
What does good inheritance look like?

- Initial 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

\[
B <: D
\]

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 \( \phi \) 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

```
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 $\phi$ 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

C <: A

Base
A foo(B b)

Derived
C foo(D d)

Return types are **covariant**
What does good inheritance look like?

- **Initial 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**
    - $\text{assert (x > 0)}$  
    - $\text{assert (x != 0)}$

\begin{align*}
\text{Base} & : A \text{ foo}(B \ b) \\
\text{Derived} & : C \text{ foo}(D \ d)
\end{align*}
What does good inheritance look like?

- **Initial 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

```plaintext
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 $\varphi$ 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
  - Invariants must still hold

<table>
<thead>
<tr>
<th>Base</th>
<th>Derived</th>
</tr>
</thead>
<tbody>
<tr>
<td>A foo(B b)</td>
<td>C foo(D d)</td>
</tr>
</tbody>
</table>
What does good inheritance look like?

- Initial 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)

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

```java
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

```cpp
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

```cpp
class Parent {
    virtual void foo() { bar(); }
    virtual void bar() {}
};
class Child : public Parent {
    public:
        virtual void bar() { foo(); }
};
[Block, “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:
    virtual void foo() { bar(); }
    virtual void bar() {}
};

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

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

[Non Virtual Interfaces (NVI) help clarify & are common in C++.]
[Bloch, “Effective Java”]

[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:
    virtual void foo() { bar(); }
    virtual void bar() {};
};

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

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

[Bloch, "Effective Java"]

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

Other patterns help even more...
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...

![Diagram showing the relationship between Person, Car, and CarOwner]

Person is-a Car

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

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

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

How could you make it better?
So let’s try it out...

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

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

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

That a car is amongst a **Person’s** possessions does not make them a special **Person**
So let’s try it out...

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

That a car is amongst a **Person**’s possessions does not make them a special **Person**.

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?

Frogs can be male or female
So why is inheritance hard?

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

**Frogs** can be male or female

```
Frog
  ├─ MaleFrog
  ├─ FemaleFrog
```
So why is inheritance hard?

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

Frogs can be male or female

Frog

MaleFrog  FemaleFrog

Frog

-sex:{male,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
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

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

Note, these are now roles, not people.

- 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

Frogs 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

Frogs 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
  - 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.

```c++
std::vector<People*> folks;
```

- 0) Student
- 1) Student
- 2) Lecturer
- 3) Professor
- 4) Student
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

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.
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 Cat serve a purpose?
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?

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.

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

`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

Animal has-a Movement

Movement selects from the ways any Animal can move.
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 Crawl
  has-a Vocalization
```

```py
Animal
  move()
  speak()
```
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
  has-a Vocalization
```
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

---

![Diagram](image-url)

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

![Diagram of Animal class with has-a relationships to Movement and Vocalization classes. Movement has subclasses Crawl, Fly, Saunter. Vocalization has subclasses 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?

Recall: identify & isolate change

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

Previously static requirements will often become dynamic.
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.
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**.