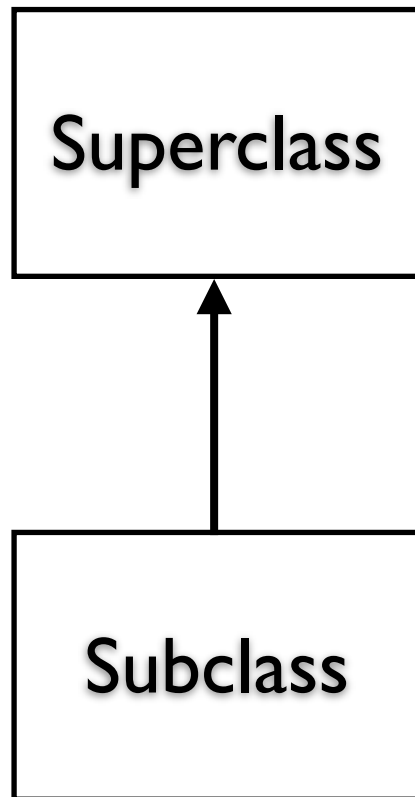


# Polymorphism, Dynamic Binding, and other fun words

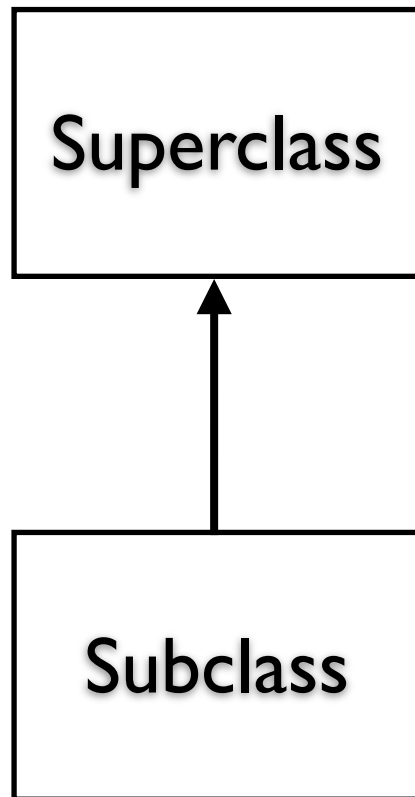
# Class Hierarchies

## Review



# Class Hierarchies

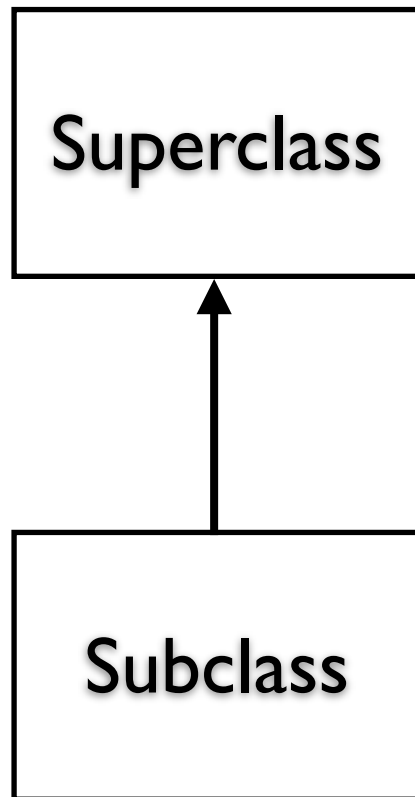
## Review



- Inheritance Relationship

# Class Hierarchies

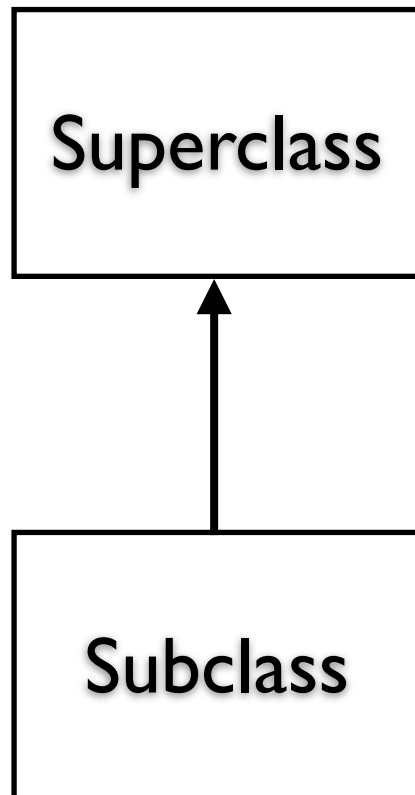
## Review



- Inheritance Relationship
- Subclass inherits every feature of superclass

# Class Hierarchies

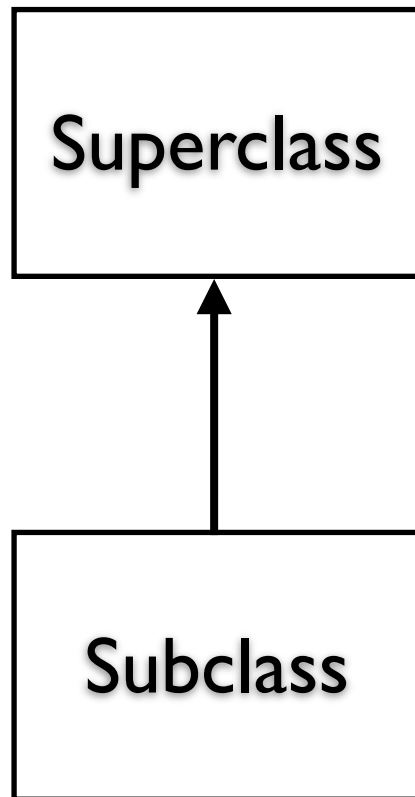
## Review



- Inheritance Relationship
- Subclass inherits every feature of superclass
- Subclass may override behavior and add new methods

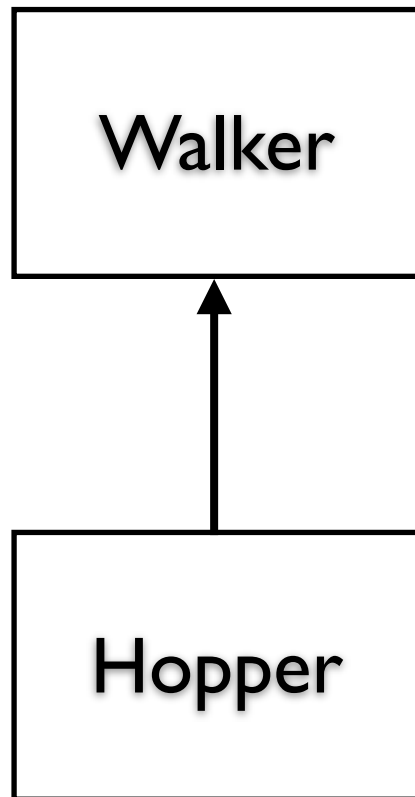
# Class Hierarchies

## Review

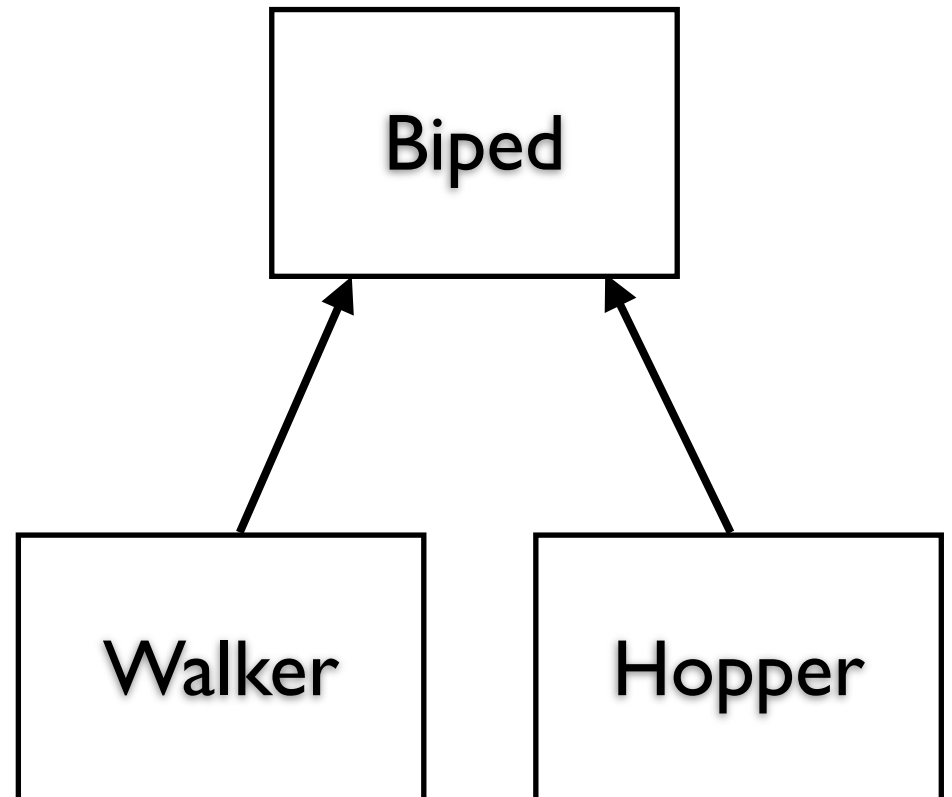


- Inheritance Relationship
- Subclass inherits every feature of superclass
- Subclass may override behavior and add new methods
- Subclass **extends** Superclass

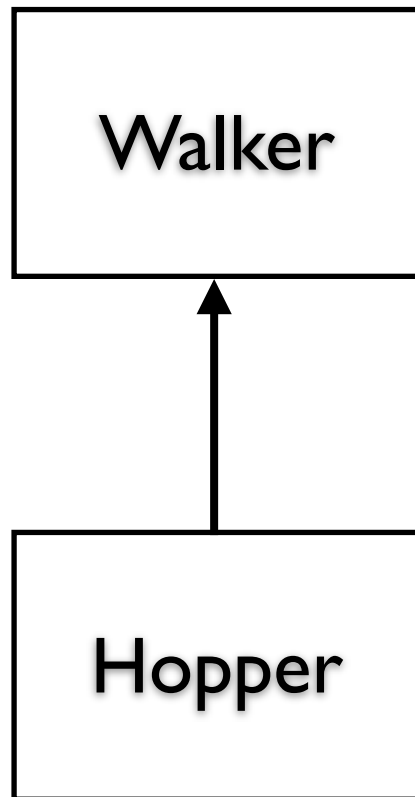
# Hierarchy Choices



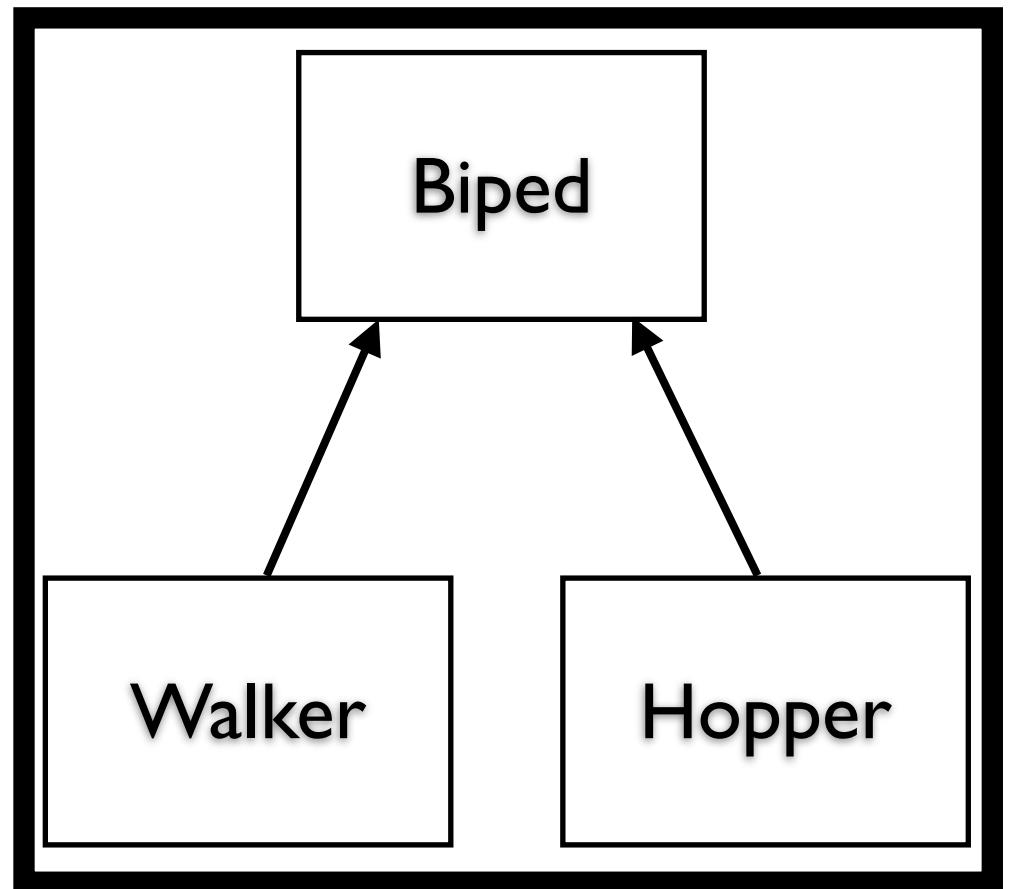
**VS**



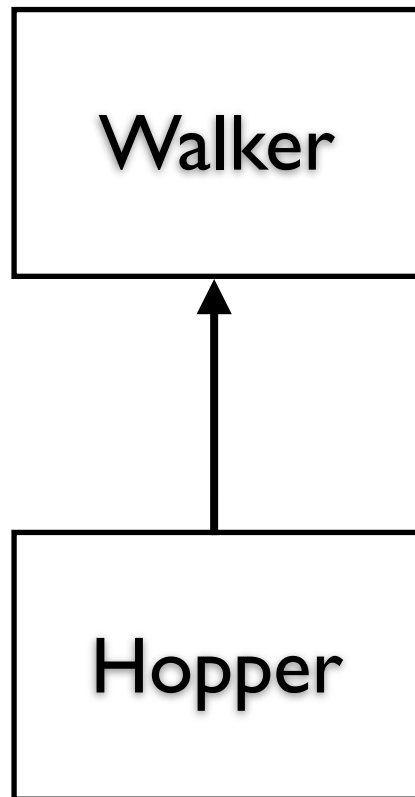
# Hierarchy Choices



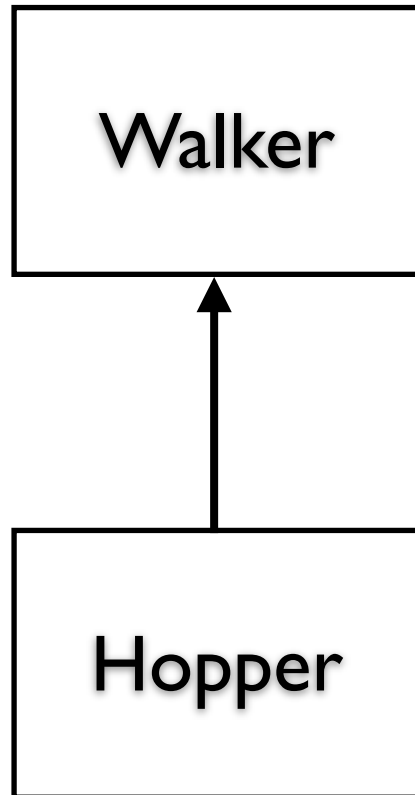
**VS**



# Analysis

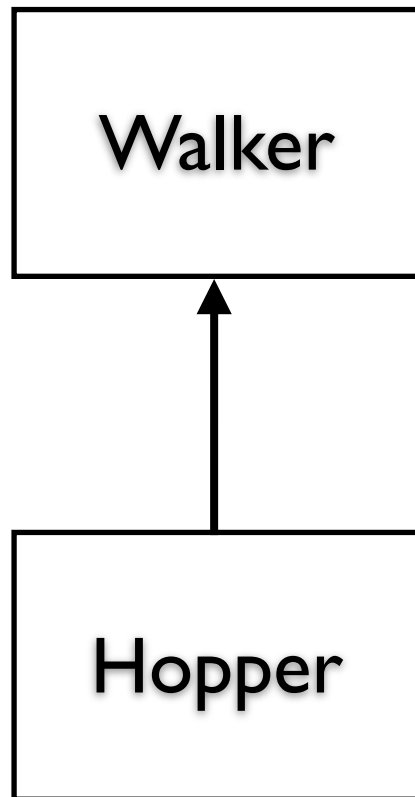


# Analysis



What motivates this set up?

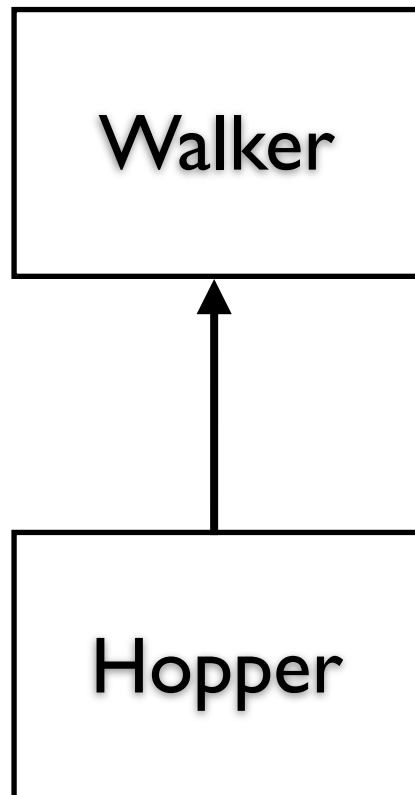
# Analysis



What motivates this set up?

- Hopper's do most of the same things that a Walker does

# Analysis

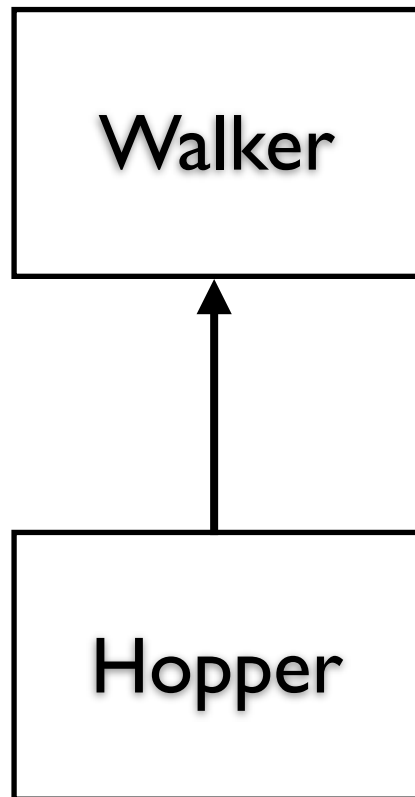


What motivates this set up?

- Hopper's do most of the same things that a Walker does

Why is this a bad idea?

# Analysis



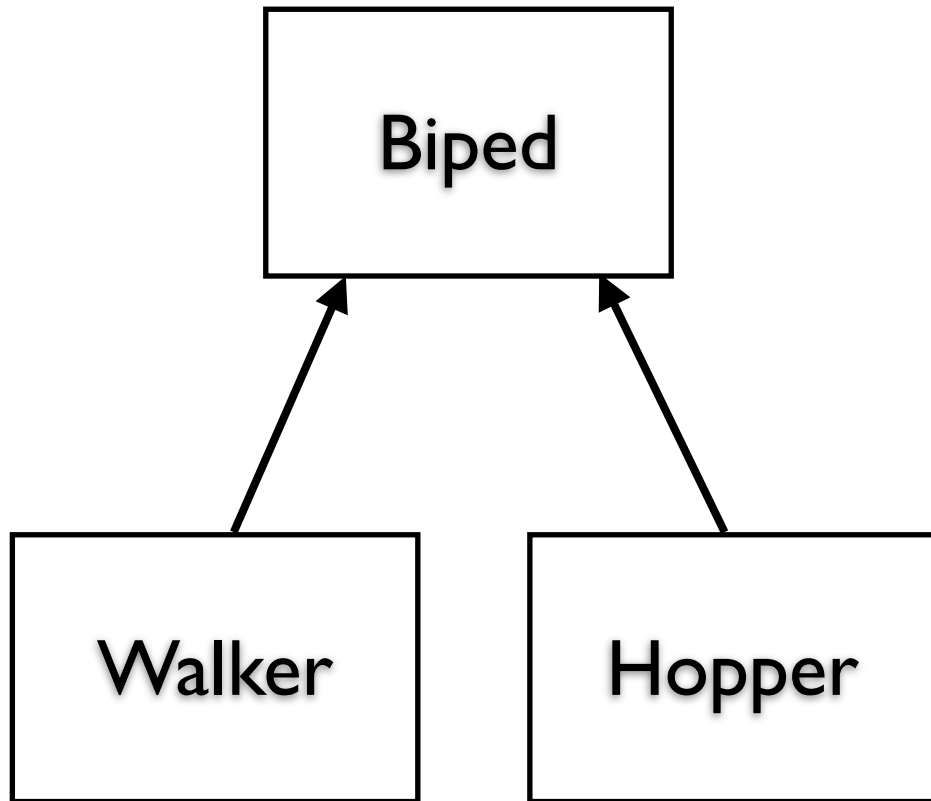
What motivates this set up?

- Hopper's do most of the same things that a Walker does

Why is this a bad idea?

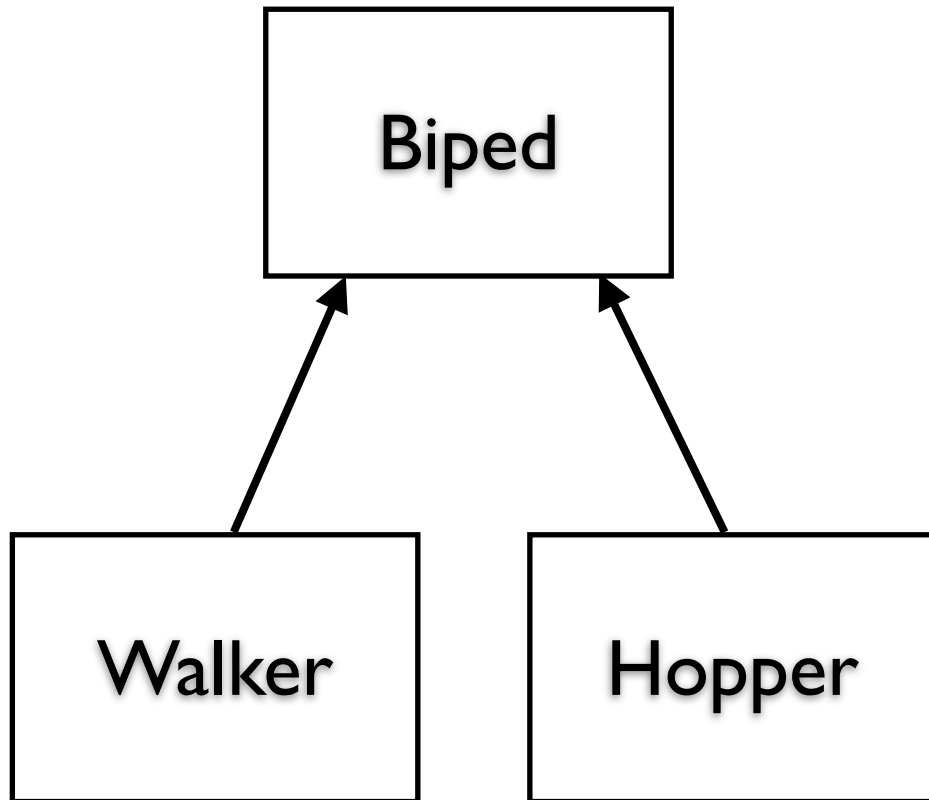
- You can't say a Hopper "is a" Walker

# Analysis



# Analysis

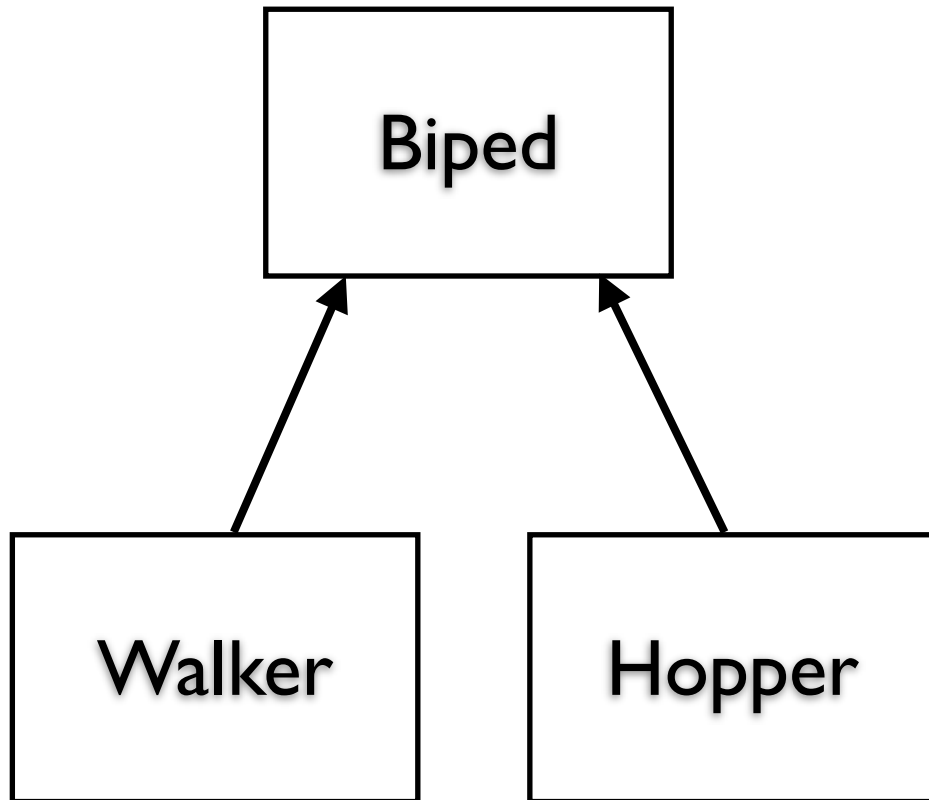
What motivates this set up?



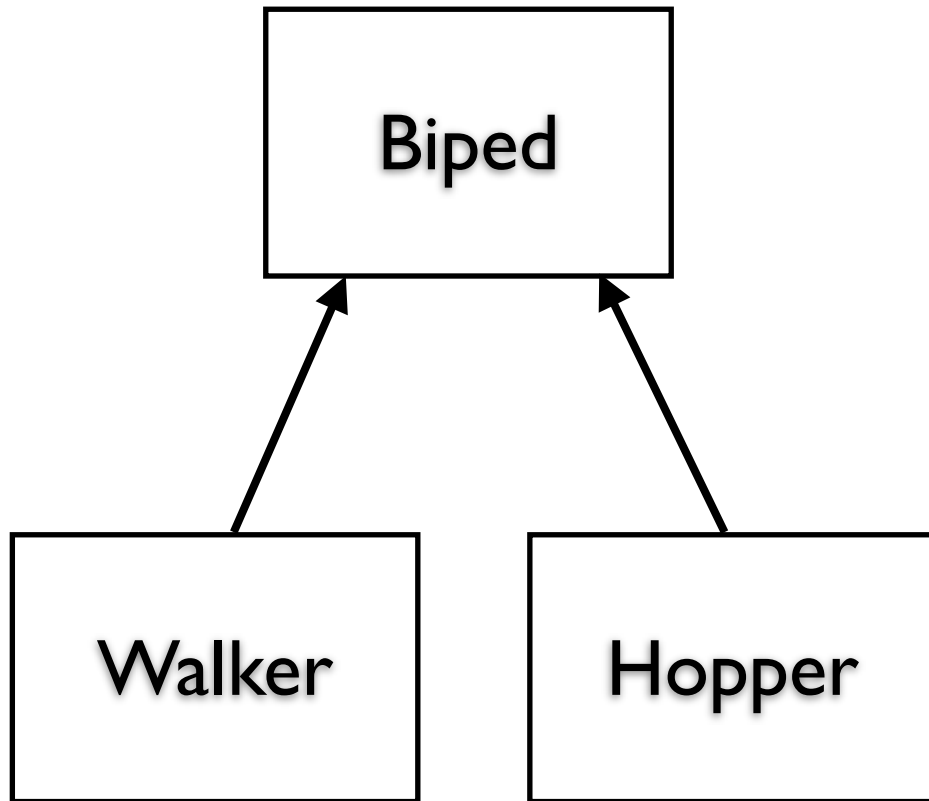
# Analysis

What motivates this set up?

- Biped holds the code both Walker and Hopper share



# Analysis

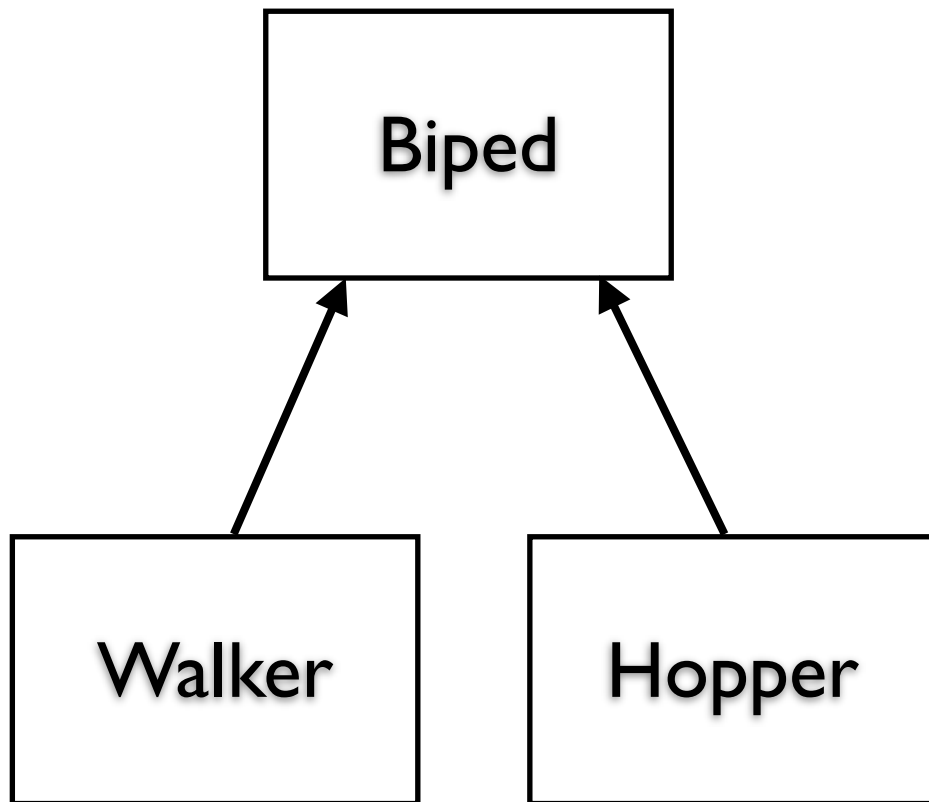


What motivates this set up?

- Biped holds the code both Walker and Hopper share

Why is this a good idea?

# Analysis



What motivates this set up?

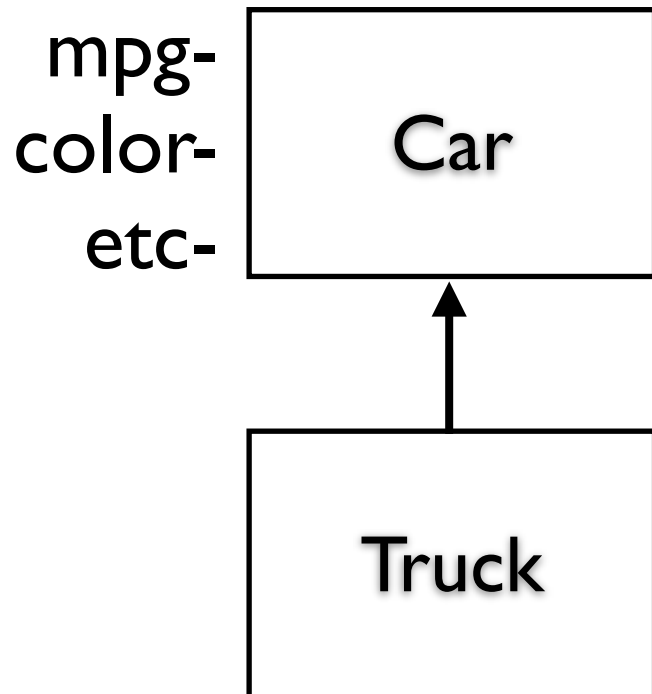
- Biped holds the code both Walker and Hopper share

Why is this a good idea?

- You can say a Walker is a biped
- Less duplicate code

# Inheritance Principle

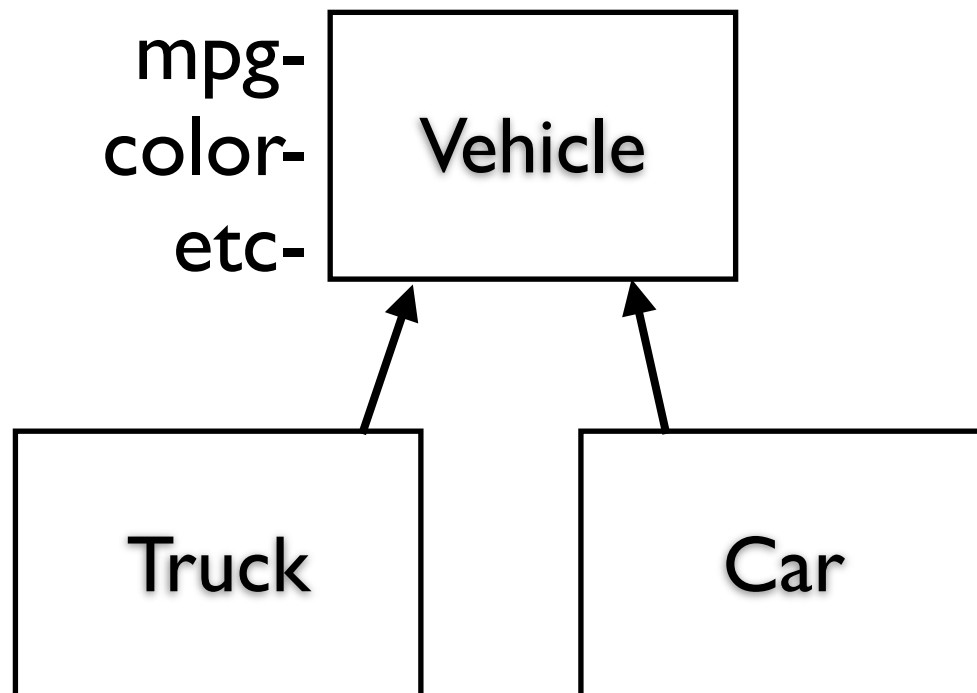
- You can avoid duplicate code and bad hierarchies by “factoring out” common code from subclasses into a common superclass.



Is this okay? If not, how would you fix it?

# Factoring Out

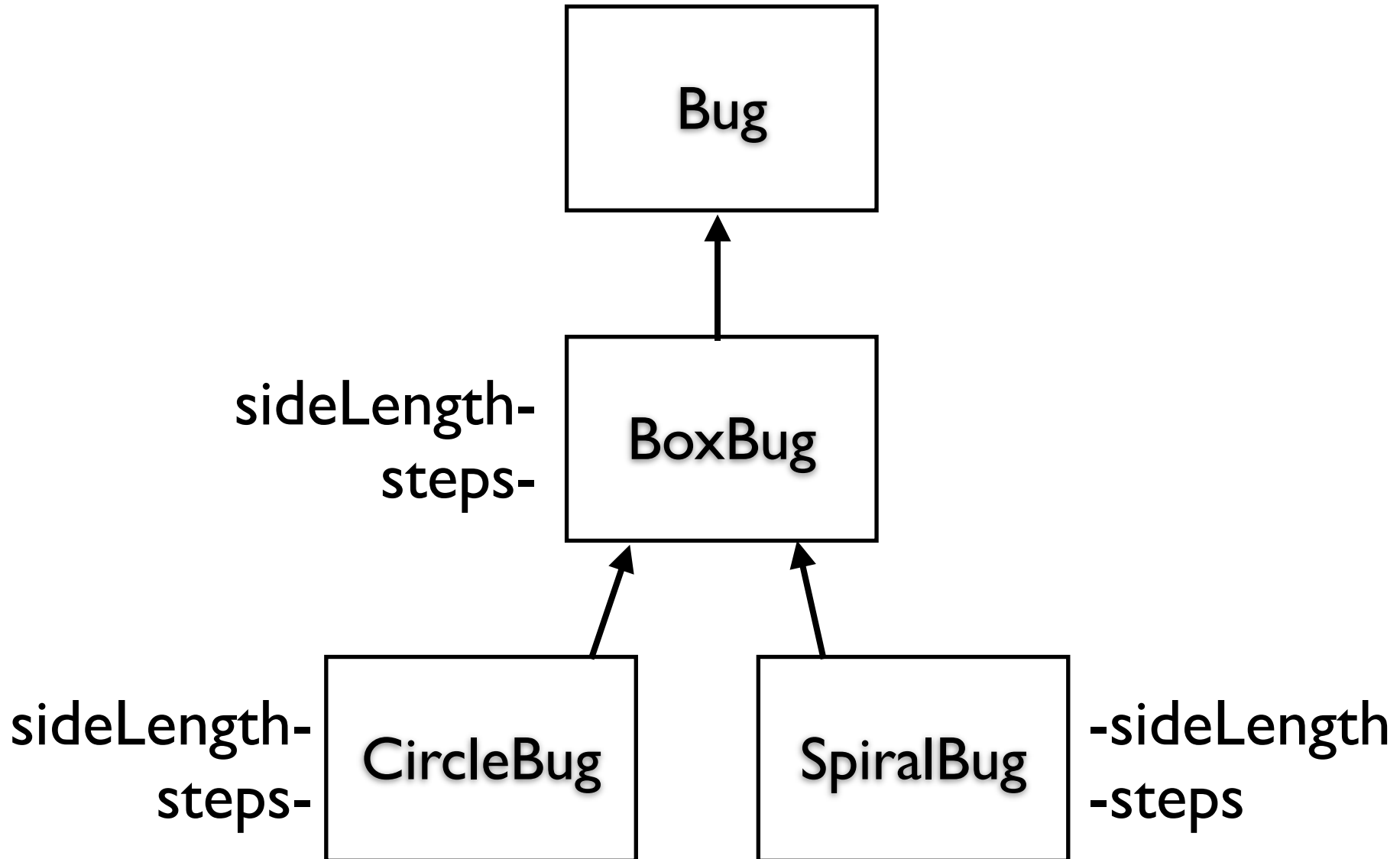
- You can avoid duplicate code and bad hierarchies by “factoring out” common code from subclasses into a common superclass.



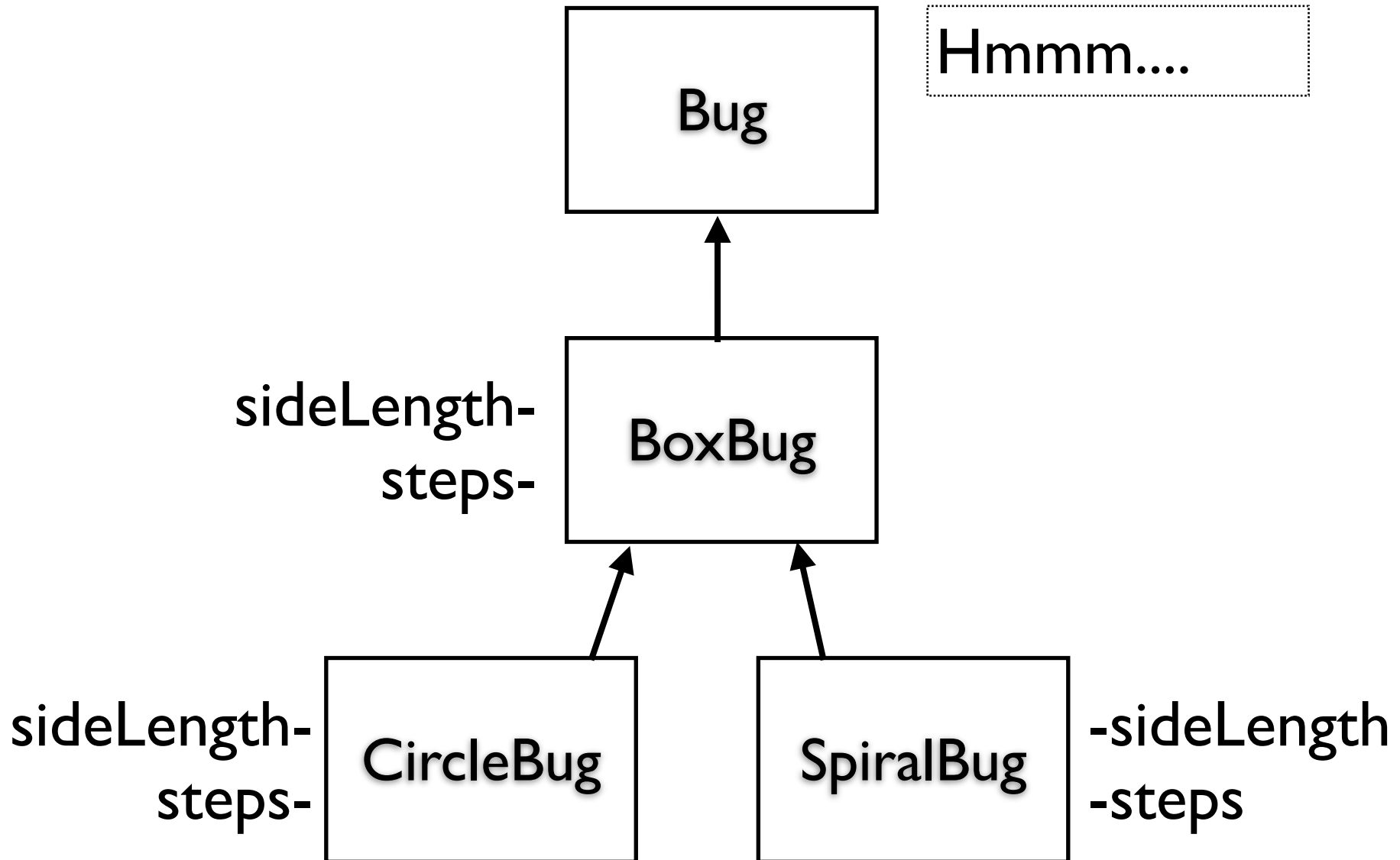
That's better!

Adding other types of Vehicles makes more sense this way

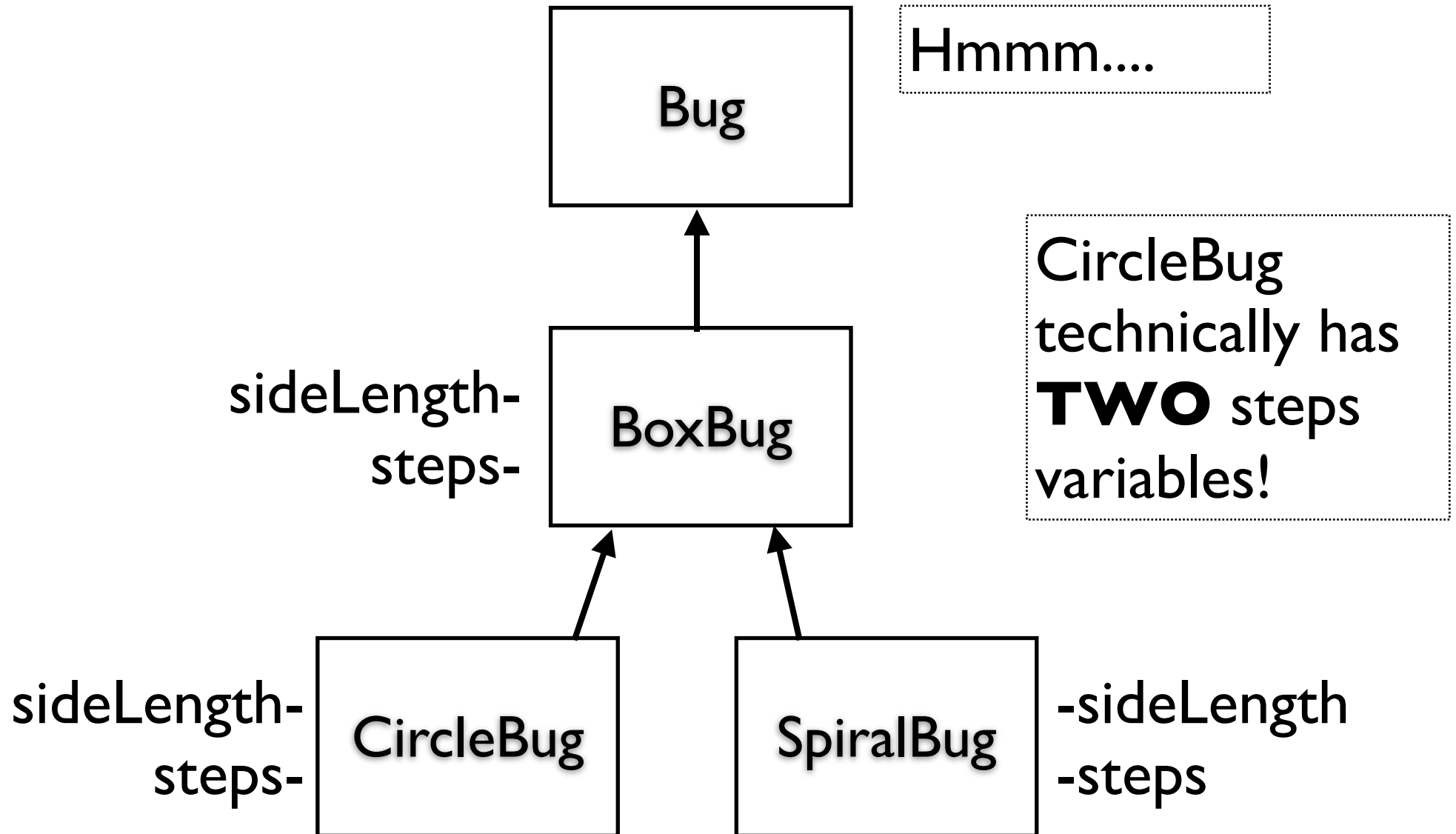
# Factoring Out



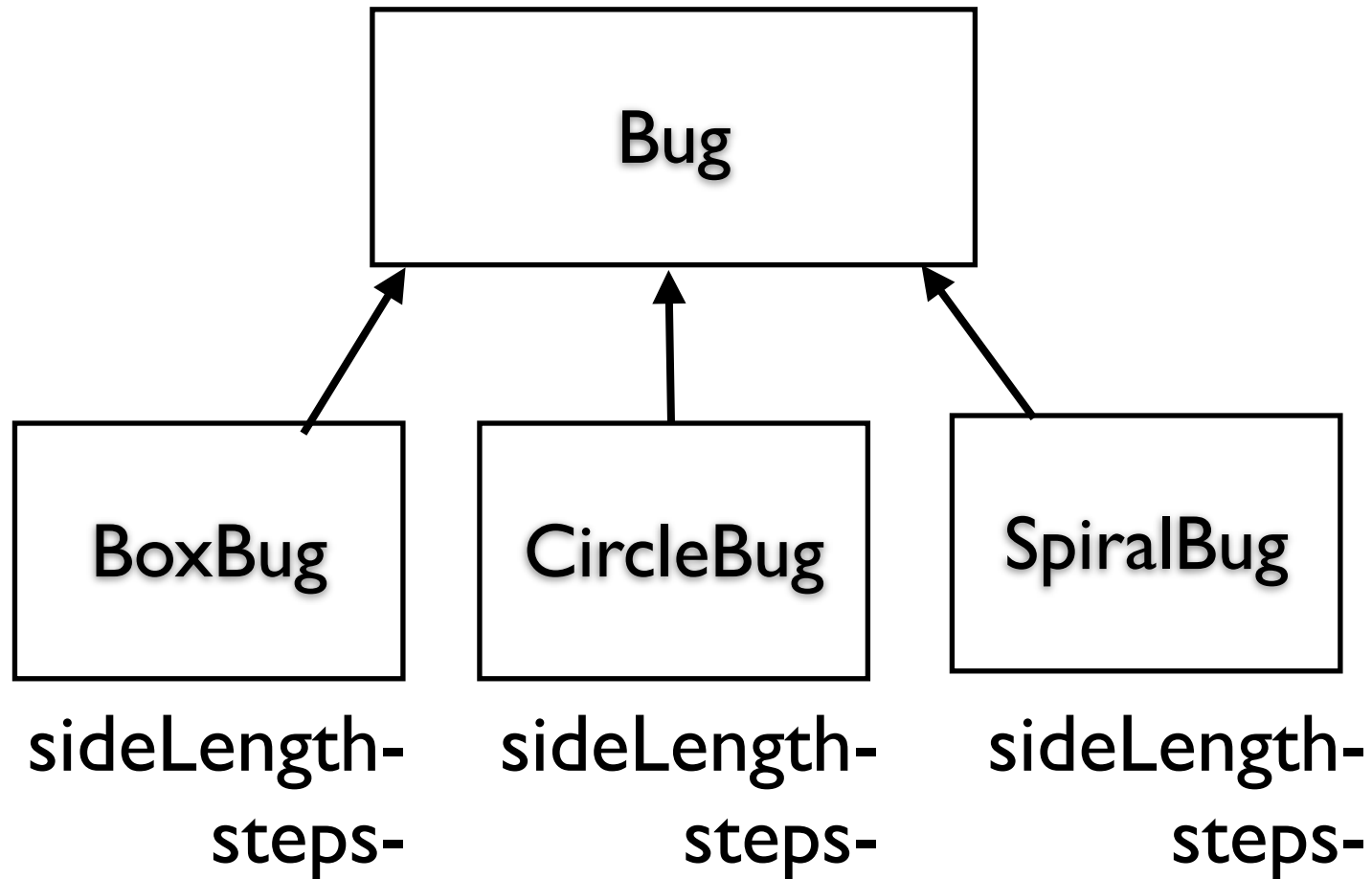
# Factoring Out



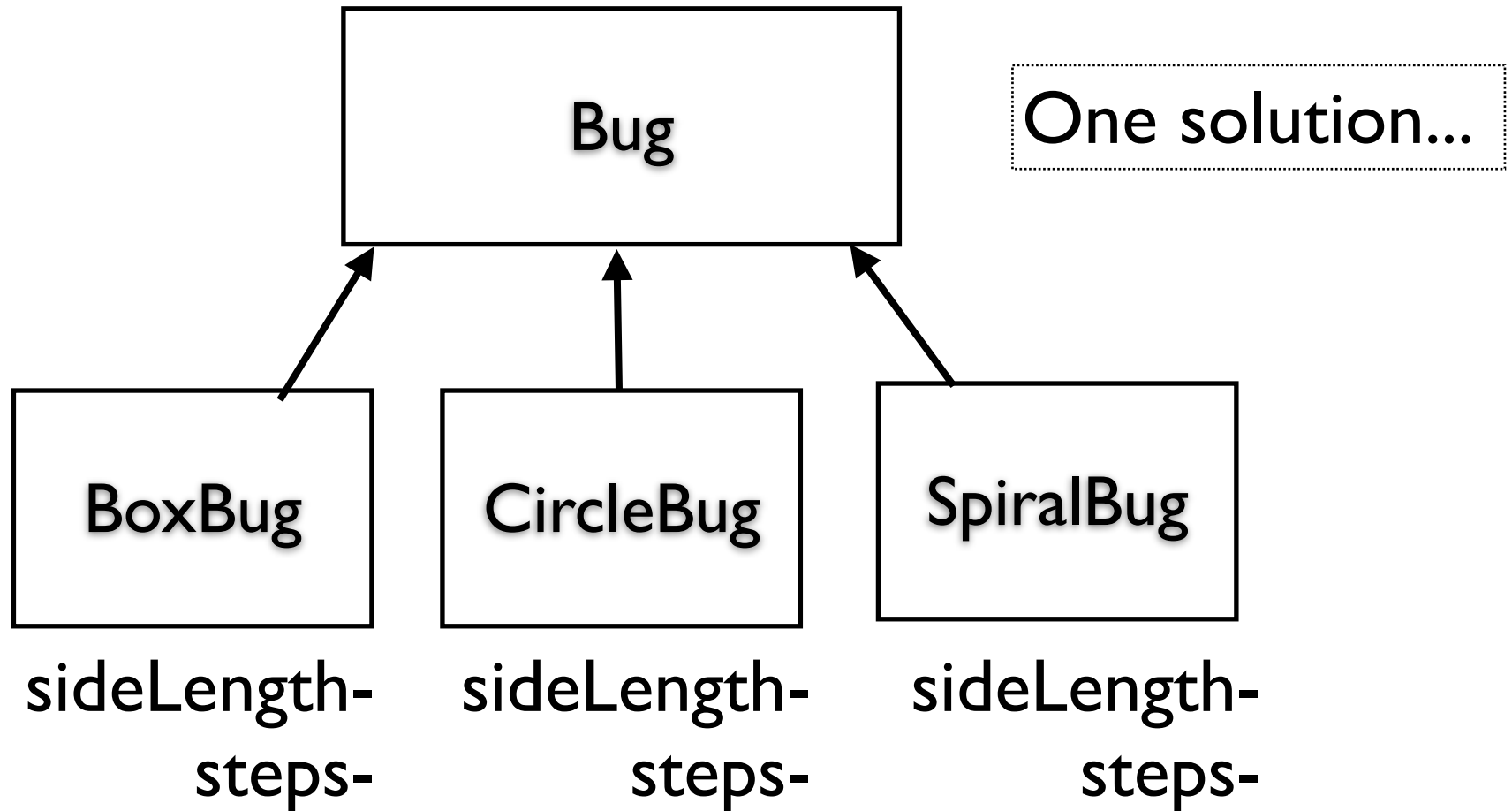
# Factoring Out



# Factoring Out



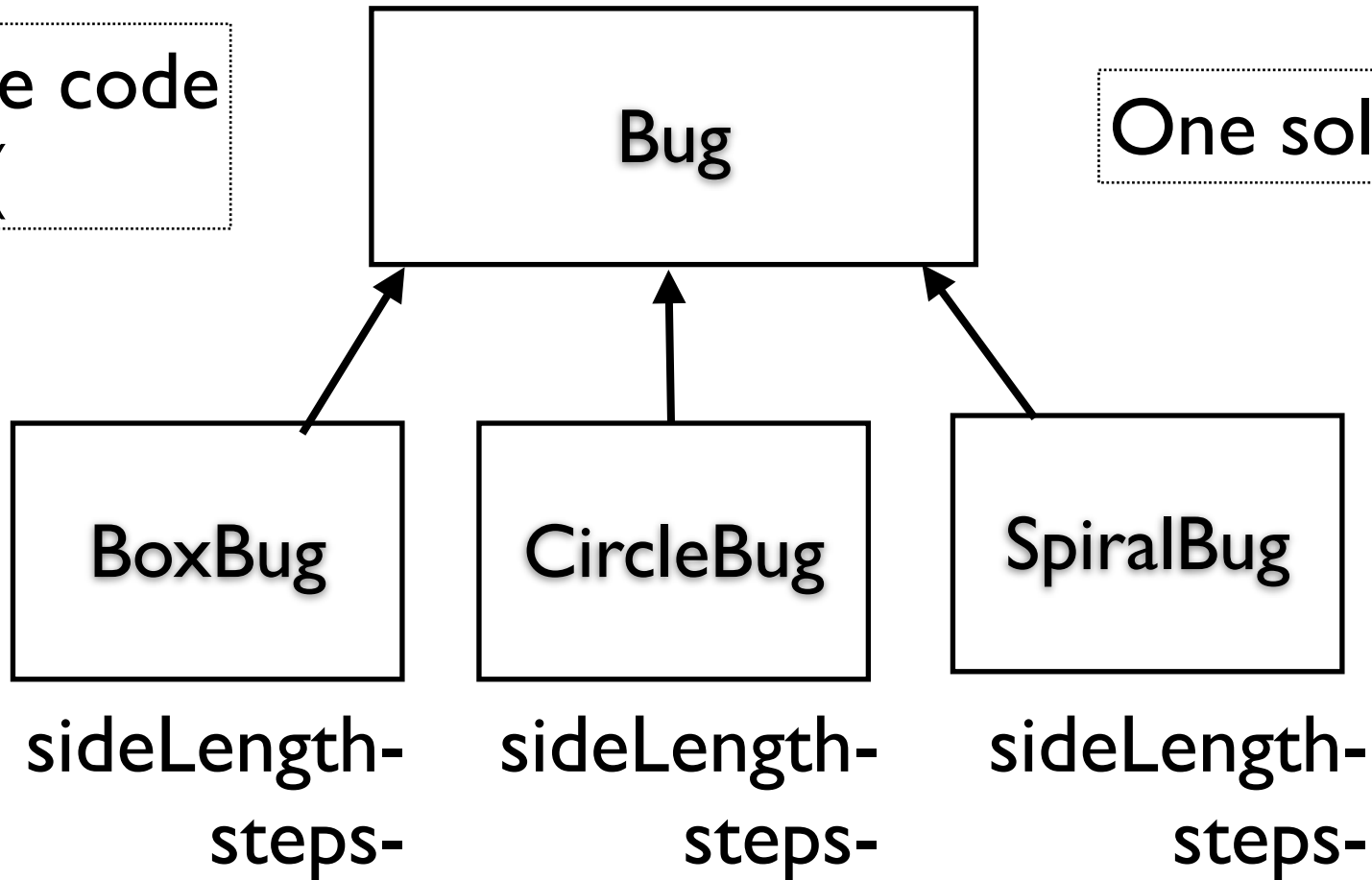
# Factoring Out



# Factoring Out

Duplicate code  
=(

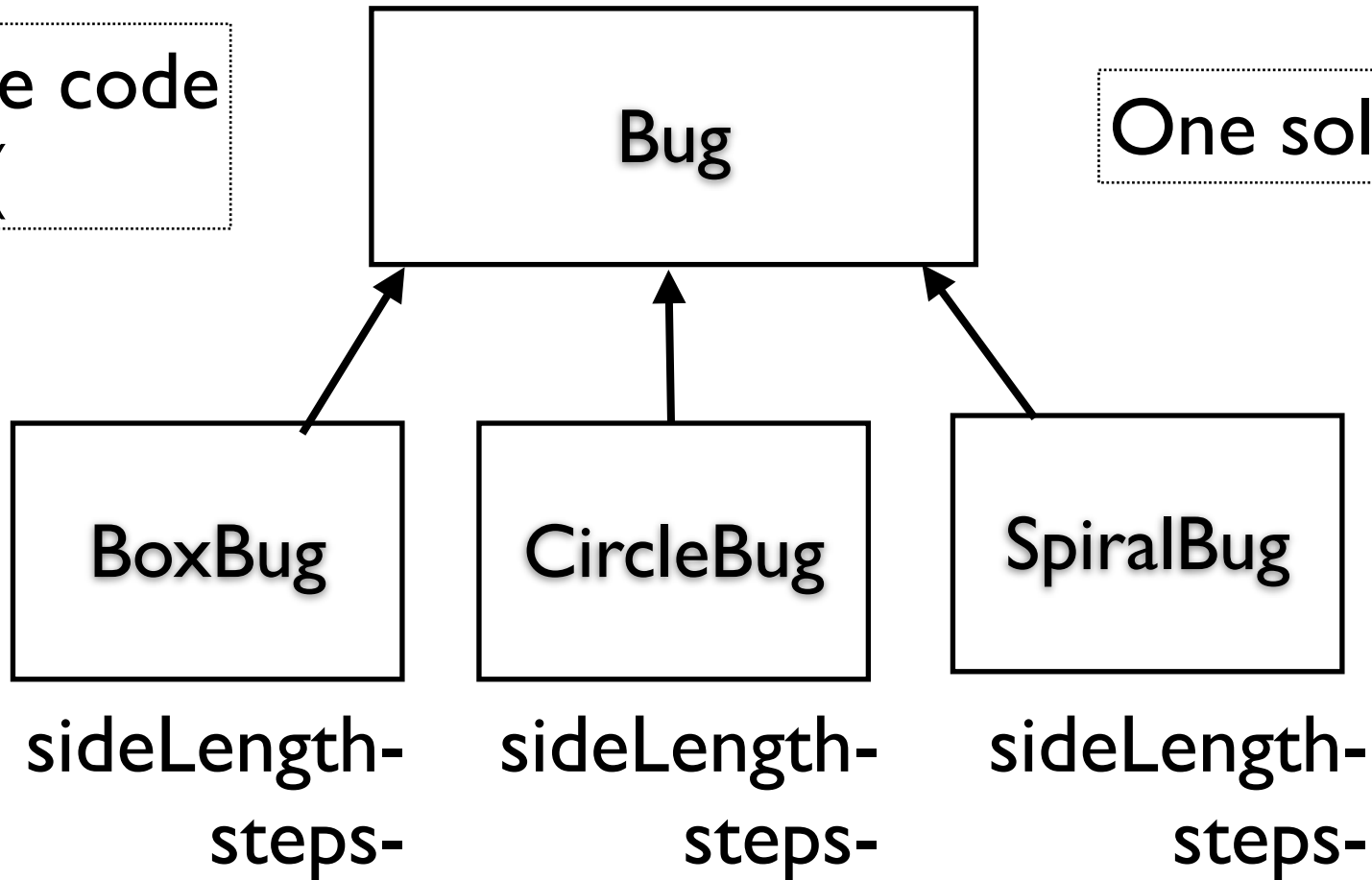
One solution...



# Factoring Out

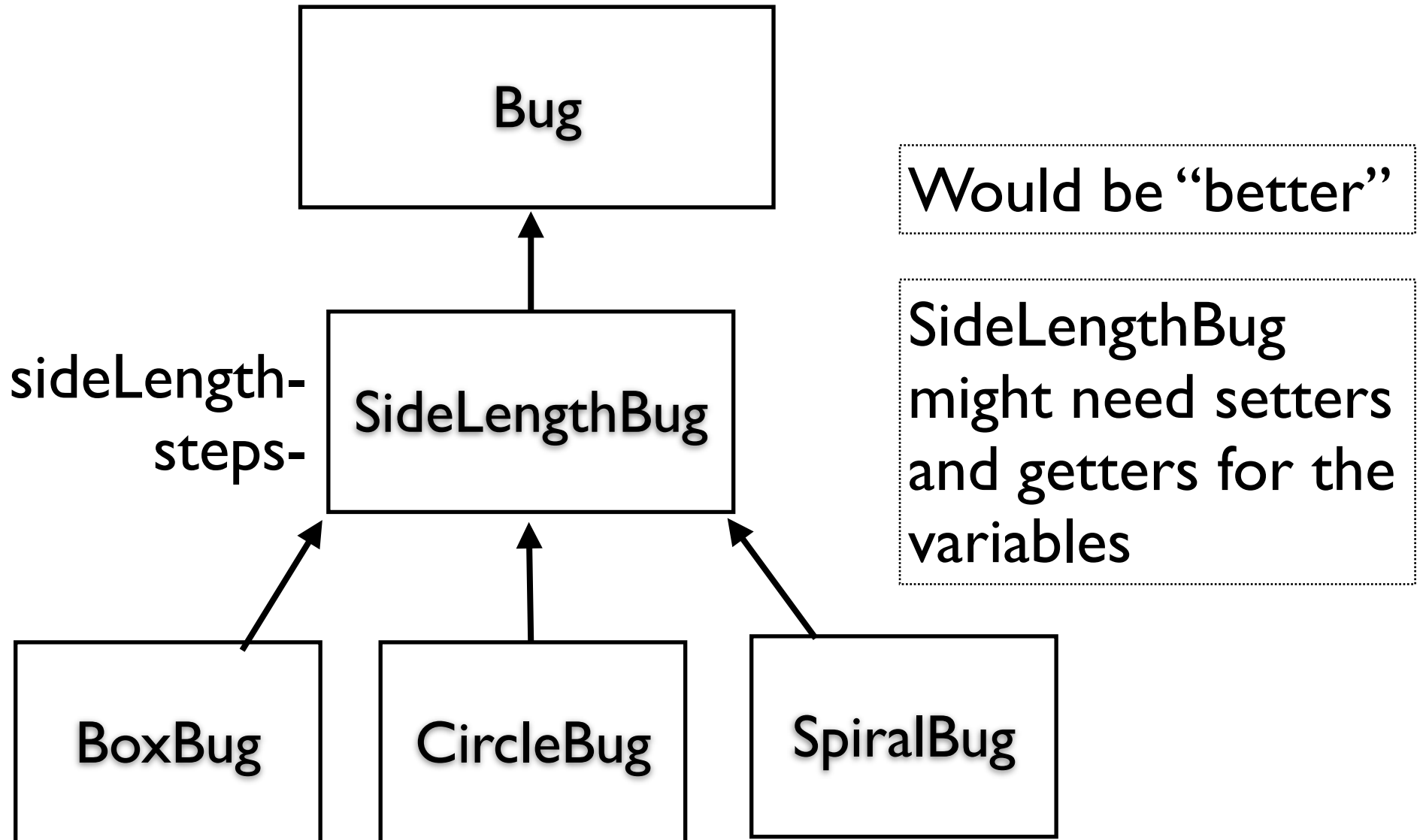
Duplicate code  
=(

One solution...



We can't modify the Bug class =(

# Factoring Out



# Actor and Bug

```
public class Actor
{
    public void act()
    {
        setDirection(getDirection() + 180);
    }
}
```

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }

    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

# Actor and Bug

```
public class Actor
{
    public void act()
    {
        setDirection(getDirection() + 180);
    }
}
```

**Bug overrides inherited  
act() method**

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }

    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

# Somewhere in code...

pseudocode



```
for(ever)
{
  for each(Actor a)
  {
    a.act();
  }

  redrawScreen();
}
```

Anything that is a subclass of Actor in the Grid is told to act().

# LeftHandedBug

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }

    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```

# LeftHandedBug

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }

    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```

The act() method is **NOT** overridden, and yet the LeftHandedBug will behave differently!

Why?

# LeftHandedBug

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }

    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```

# LeftHandedBug

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn(); ?
    }

    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```

# LeftHandedBug

```
public class Bug extends Actor
```

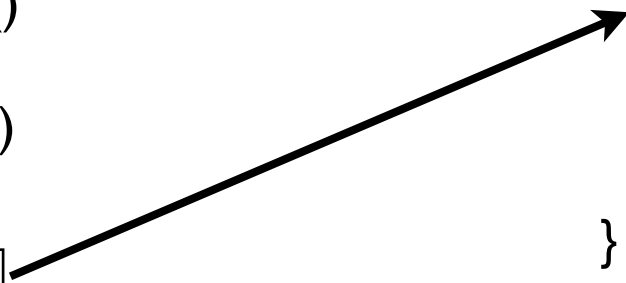
```
{  
    public void act()  
    {  
        if(canMove())  
            move();  
        else  
            turn();  
    }  
}
```

?

```
public class LeftHandedBug extends Bug
```

```
{  
    public void turn()  
    {  
        setDirection(getDirection() - 45);  
    }  
}
```

```
    public void turn()  
    {  
        setDirection(getDirection() + 45);  
    }  
}
```

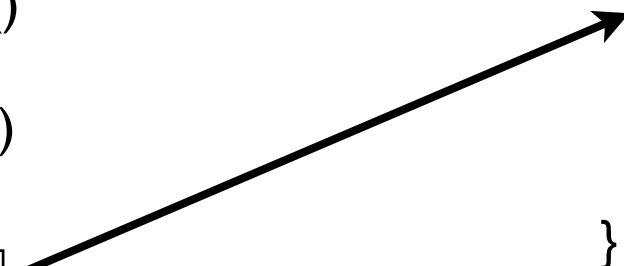


# LeftHandedBug

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }
    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```

?



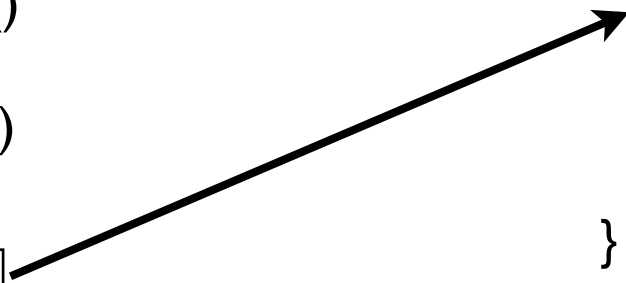
# LeftHandedBug

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }
    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

?

↓ Bug's

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```



# LeftHandedBug

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }
    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

?

↓ Bug's

LeftHandedBug's

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```

# LeftHandedBug

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }

    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

?

↓ Bug's

LeftHandedBug's

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```

The **inherited** act() method will call the **overriden** turn() method instead of the superclass version.

# Dynamic Binding

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }
    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

?

↓ Bug's

LeftHandedBug's

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```

This is called dynamic binding

The act() method's choice of turn() is decided as the program is running.

# Dynamic Binding

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }
    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

?

↓ Bug's

LeftHandedBug's

```
public class LeftHandedBug extends Bug
{
    public void turn()
    {
        setDirection(getDirection() - 45);
    }
}
```

By overriding only `canMove()`, `move()`, or `turn()`, we can insure that all subclasses of `Bug` follow the same acting procedure:

- check if you can move
- if so, then move
- otherwise turn

# Dynamic Binding

```
public class Bug extends Actor
{
    public void act()
    {
        if(canMove())
            move();
        else
            turn();
    }

    public void turn()
    {
        setDirection(getDirection() + 45);
    }
}
```

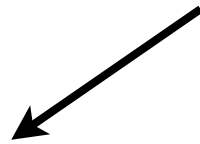
Example's:

- We could make a **NervousBug** that **canMove()** only if it's not within two spaces of another Bug
- We could make a **HopperBug** which has a **move()** method that goes two spots instead of one

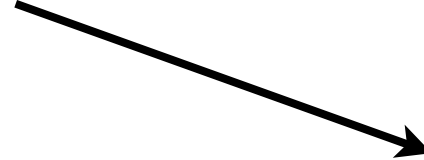
# Polymorphism

- Fancy word that means a method can be written for one type that will work for all subtypes.

## Not Polymorphism (overloading)



```
public void actNTimes(BoxBug b, int n)
{
    for(int i=0; i<n; i++)
        b.act();
}
```



```
public void actNTimes(CircleBug b, int n)
{
    for(int i=0; i<n; i++)
        b.act();
}
```

# Polymorphism

- Fancy word that means a method can be written for one type that will work for all subtypes.

## Polymorphism



```
public void actNTimes(Bug b, int n)
{
    for(int i=0; i<n; i++)
        b.act();
}
```

# Polymorphism

- Fancy word that means a method can be written for one type that will work for all subtypes.

## Polymorphism



```
public void actNTimes(Bug b, int n)
{
    for(int i=0; i<n; i++)
        b.act();
}
```

This method now works for all types of Bugs, including BoxBug's and CircleBug's.

# Method Overloading versus Polymorphism

```
Bug a = new BoxBug();
```

```
a.act();
```

```
((Bug)a).act();
```

```
mystery(a);
```

```
mystery((BoxBug)a);
```

```
public void mystery(Bug b)
{
    System.out.println("Bug");
}
```

```
public void mystery(BoxBug b)
{
    System.out.println("Box");
}
```

# Method Overloading versus Polymorphism

```
Bug a = new BoxBug();
```

```
a.act(); ← BoxBug's act()
```

```
((Bug)a).act();
```

```
mystery(a);
```

```
mystery((BoxBug)a);
```

```
public void mystery(Bug b)
{
    System.out.println("Bug");
}
```

```
public void mystery(BoxBug b)
{
    System.out.println("Box");
}
```

# Method Overloading versus Polymorphism

```
Bug a = new BoxBug();
```

```
a.act(); ← BoxBug's act()
```

```
((Bug)a).act(); ← BoxBug's act()
```

```
mystery(a);
```

```
mystery((BoxBug)a);
```

```
public void mystery(Bug b)
{
    System.out.println("Bug");
}
```

```
public void mystery(BoxBug b)
{
    System.out.println("Box");
}
```

# Method Overloading versus Polymorphism

```
Bug a = new BoxBug();
```

```
a.act(); ← BoxBug's act()
```

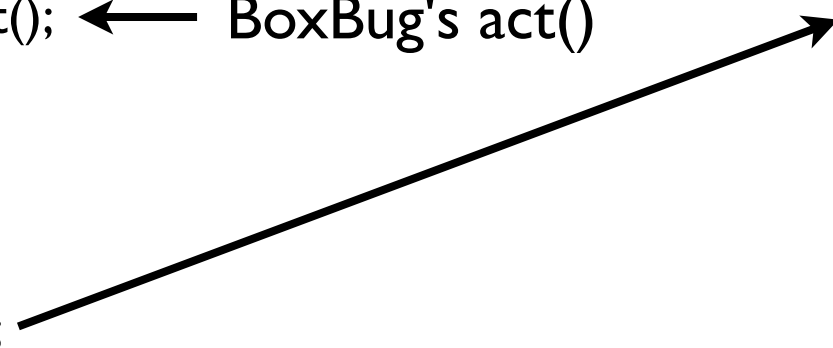
```
((Bug)a).act(); ← BoxBug's act()
```

```
mystery(a);
```

```
mystery((BoxBug)a);
```

```
public void mystery(Bug b)
{
    System.out.println("Bug");
}
```

```
public void mystery(BoxBug b)
{
    System.out.println("Box");
}
```



# Method Overloading versus Polymorphism

Bug a = new BoxBug();

a.act(); ← BoxBug's act()

((Bug)a).act(); ← BoxBug's act()

```
public void mystery(Bug b)
{
    System.out.println("Bug");
}
```

mystery(a);

```
public void mystery(BoxBug b)
{
    System.out.println("Box");
}
```

mystery((BoxBug)a);

# Summary

- Use of a common superclass can reduce duplicate code
- Dynamic binding allows you to change a method by overriding a method that it uses
- Polymorphism allows a method to accept a type of variable and any of its subtypes.