Project Process

CS 420/520

Don’t plan ahead!

- Well, don’t plan *too far* ahead
  - You don’t yet know enough about your application to plan effectively

    ➡ Plan no more than 2 iterations ahead
- Deliver a working system on each iteration
  - Don’t leave integration until the end

Typical Proposal

1. Design the Core Classes
2. Implement the Core Classes
3. Design “Nice to have” feature
4. Implement “Nice to have” feature
5. Design GUI
6. Implement GUI

Typical Proposal

1. Design the Core Classes
2. Implement the Core Classes
3. Design “Nice to have” feature
4. Implement “Nice to have” feature
5. Design GUI

---

6. Implement GUI

Instead: Step 1

- Design and Implement 1st feature
- Design and implement GUI for 1st feature
- “Deliver” and test 1st feature

Instead: Step 2

- Design and Implement 2nd feature
- Design and implement GUI for 2nd feature
- “Deliver” and test 1st and 2nd features

Instead: Step *n*

- Design and Implement n^{th} feature
- Design and implement GUI for n^{th} feature
- “Deliver” and test 1st thru n^{th} features

What's a "Feature"?

What's a "Feature"?

- Smaller than you thought!

What's a "Feature"?

- Smaller than you thought!
- Example: home security system

What's a "Feature"?

- Smaller than you thought!
- Example: home security system
  - One room house

What's a "Feature"?

- Smaller than you thought!
- Example: home security system
  - One room house
  - add "owner"

What's a "Feature"?

- Smaller than you thought!
- Example: home security system
  - One room house
  - add "owner"
  - add a door

What's a "Feature"?

- Smaller than you thought!
- Example: home security system
  - One room house
  - add "owner"
  - add a door
  - add a window

What's a "Feature"?

- Smaller than you thought!
- Example: home security system
  - One room house
  - add "owner"
  - add a door
  - add a window
  - add "intruder"

What's a "Feature"?

- Smaller than you thought!
- Example: home security system
  - One room house
  - add "owner"
  - add a door
  - add a window
  - add "intruder"
  - add a second room

What's a "Feature"?

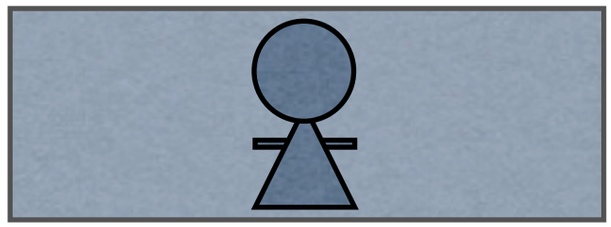
- Smaller than you thought!
- Example: home security system
  - One room house
  - add "owner"
  - add a door
  - add a window
  - add "intruder"
  - add a second room
  - add a second window

What's a "Feature"?

- Smaller than you thought!
- Example: home security system
  - One room house
  - add "owner"
  - add a door
  - add a window
  - add "intruder"
  - add a second room
  - add a second window
  - add "guest"

What's a "Feature"?

- Smaller than you thought!
- Example: home security system
  - One room house
  - add "owner"



  - add a door
  - add a window
  - add "intruder"
  - add a second room
  - add a second window
  - add "guest"

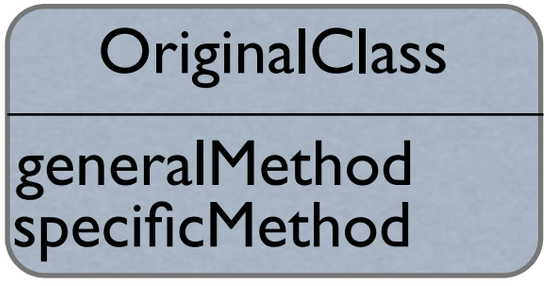
Inheritance is Complex

- Don’t try to design an inheritance hierarchy up front
- Add classes as you need them
- When two objects that you thought were the same start to exhibit different behavior, *then*

```
Person ›› openDoor
    self isIntruder ifTrue: [ … ].
    self isResident ifTrue: [ … ].
    …
```

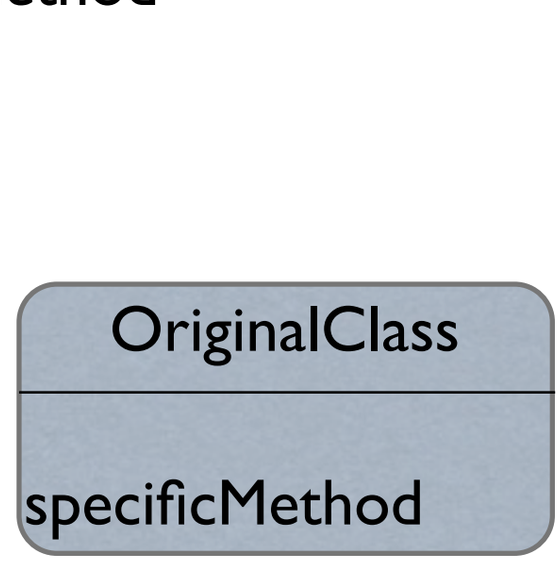
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



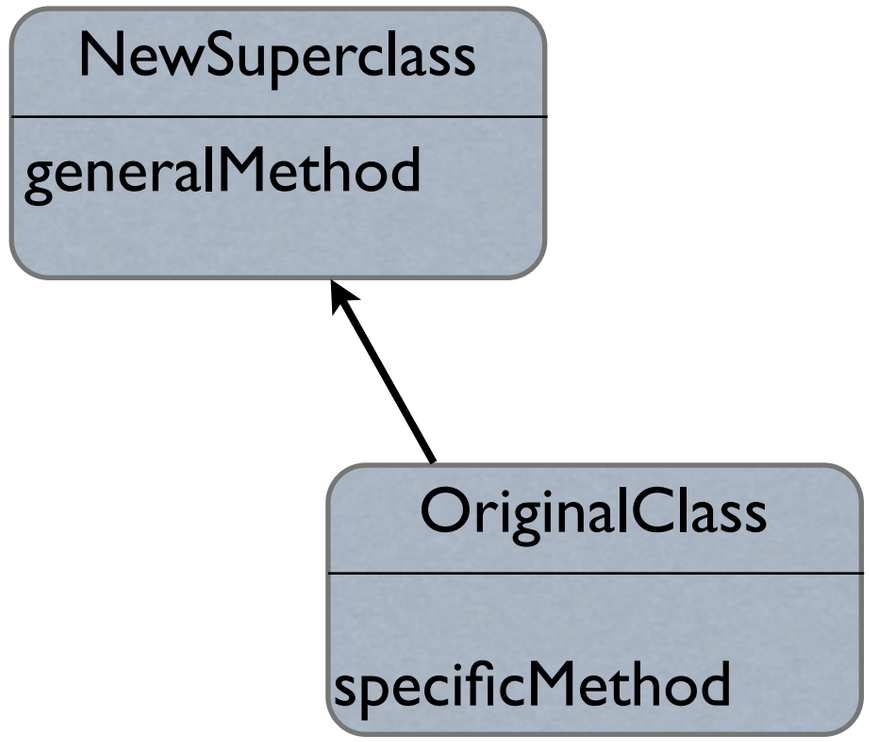
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



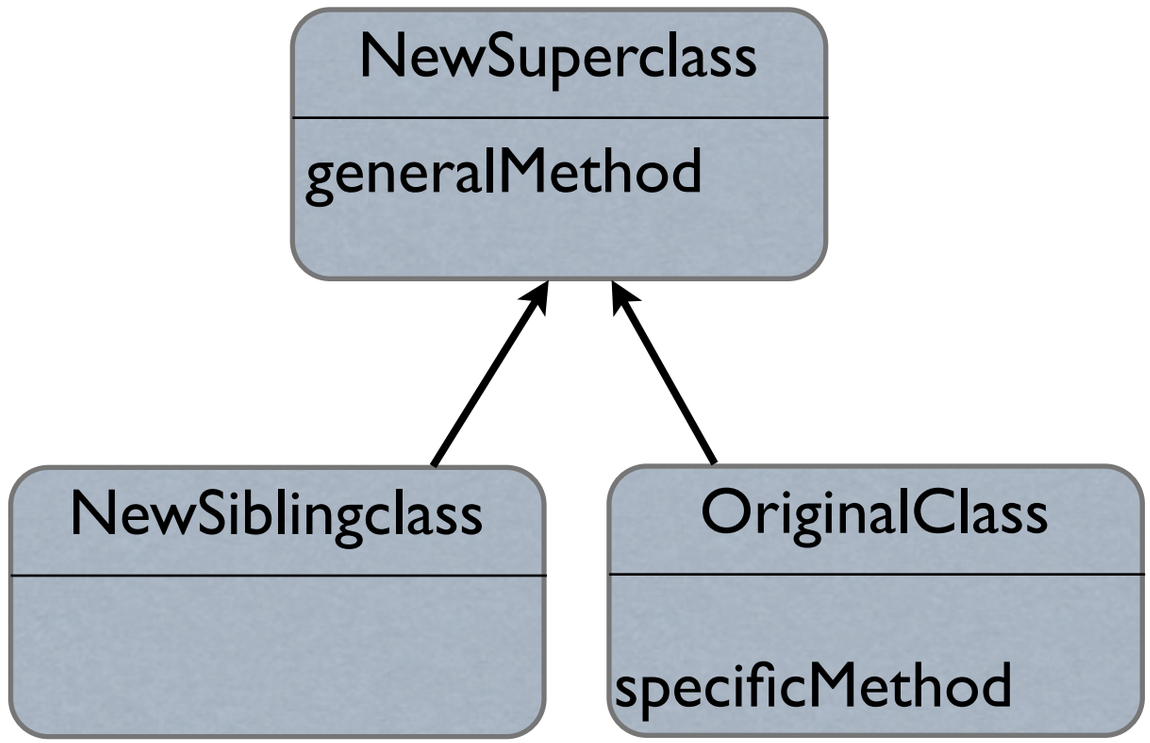
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



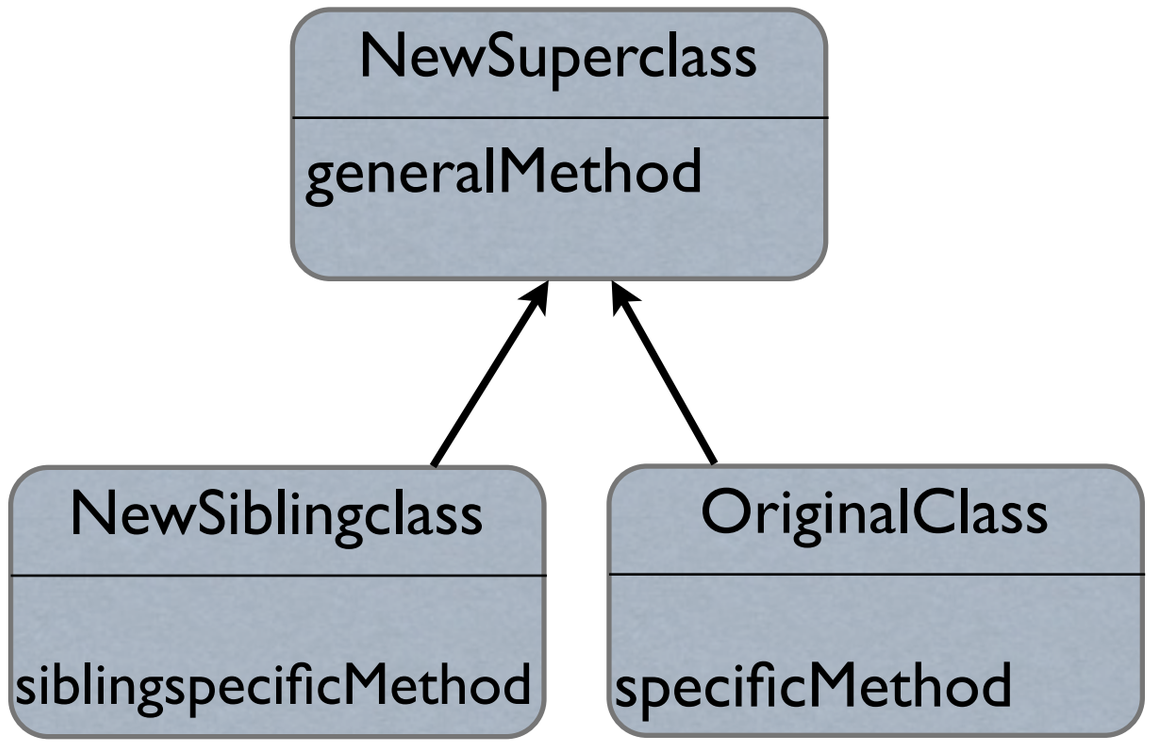
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



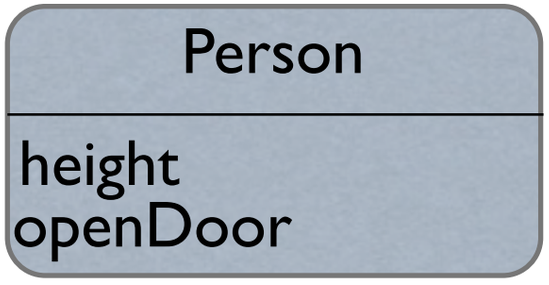
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



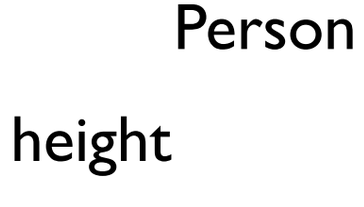
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



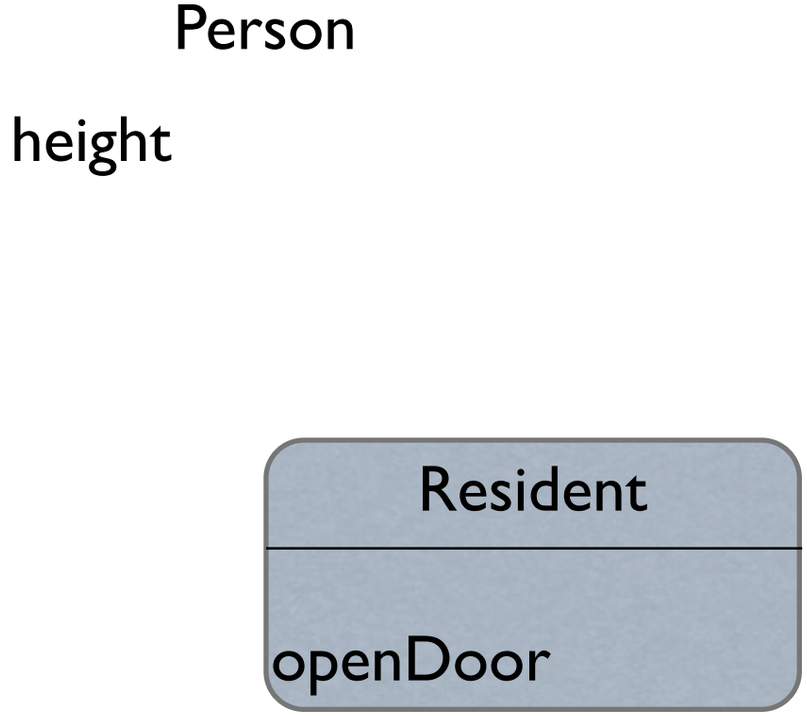
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



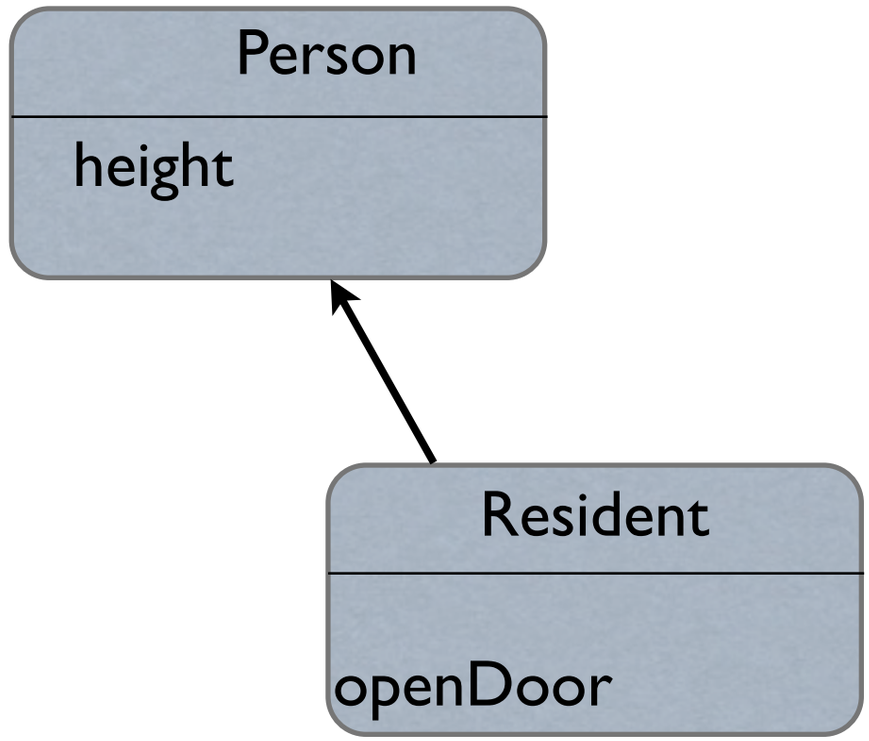
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



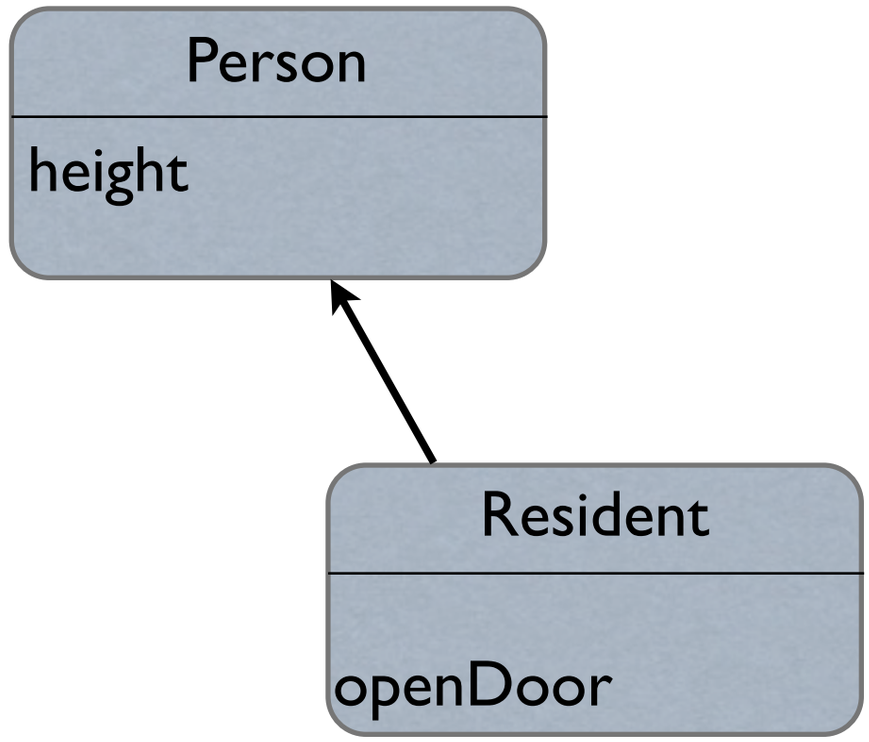
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



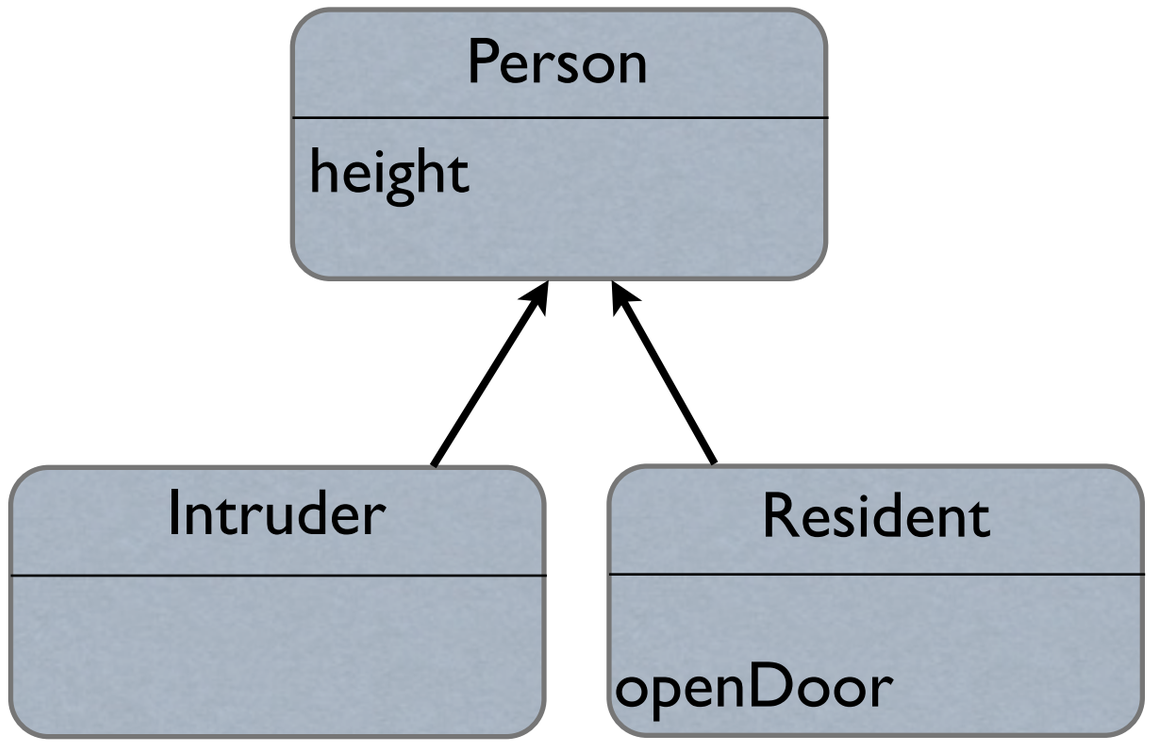
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



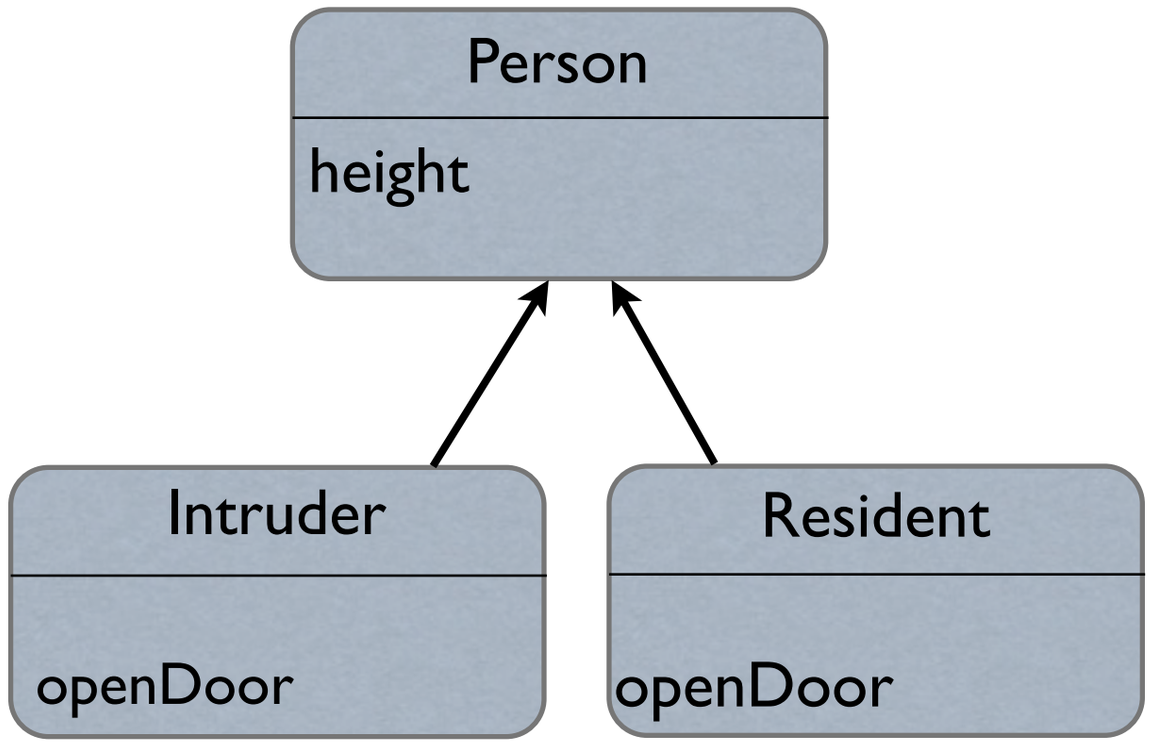
Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



Evolving a class Hierarchy

- add an abstract superclass and make your class a subclass of it
  - refactor class ▷ create superclass
- *add a sibling class*
- *move methods up when both subclases can share it*
- *duplicate and edit a method when subclasses differ*



Object-Oriented GUIs

- In the beginning, there was Chaos
- And then, there was MVC
  - Model
  - View
  - Controller

Model–View–Controller

- The key idea is to separate the application logic — the Model — from
- The View — one or more visualizations of the model on the display, and
- The Controller — which handles user input on the view, and causes the model to change in response.

Why MVC?

- Manage complexity
- Re-use models with different views
- Re-use views with different models

Warning: MVC ≠ MVC

- There used to be a package, in Squeak, called MVC.
  - It implemented MVC
  - It has been removed from Pharo
- Morphic is the only User-interface framework in Pharo

Morphic

- Morphic is the name of Pharo's UI framework
  - Morph is also the name of the base (abstract) class that implements *Morphs*
  - Morphs combine View and Control
  - The Model can be a separate collection of objects, or the Morphs can be their own model.

Model-View separation

When should you separate Model and View?

Why MVC?

- Manage complexity
- Re-use models with different views
- Re-use views with different models

Model-View separation

When should you separate Model and View?

- if the complexity is high
- if there is a chance for re-use

Dancing Boxes

- joeTheBox is an example of a Morph with its own behavior
- behavior is very simple — no need to separate the graphical part from the behavioral part
- Hence, we gave Morphs application-specific behavior, such as danceWith: