Open Advising Sessions for Non-Concentrators

Kathi Fisler will review common advising scenarios and course decisions, with time for Q&A

**Sophomores**
Tues, 10/22
4:30-5:30, CIT 368

**First-years**
(and others new to CS)
Wed, 10/23
4:30-5:30, CIT 368

**Anyone:** Concentration Fair, Wed 10/23 @ 7 in Sayles
Open Office Hours

- Mondays 4-5pm – CIT 465

- Opportunity to talk about:
  - Course concerns/feedback
  - OOP / Lecture Concepts
  - Life/Opportunities at Brown
  - anything else

- Exclusively handled at TA hours:
  - assignment help
  - code / debugging
Lecture 13

Design Patterns

From xkcd Webcomics: https://xkcd.com/974/
Overview

- Design in a Nutshell
- Patterns
- Interfaces, Inheritance, Polymorphism
- Interfaces vs Abstract Classes
- Using Composition
Design in a Nutshell (1/3)

● Up to now, focused on how to program
  o be appropriately lazy: re-use code and ideas

● Design Patterns are proven solutions to common problems
  o used successfully before by others, refined by experience
  o generally language-independent – learn once, apply everywhere
Design in a Nutshell (2/3)

● Increasingly we learn about good design

● Some designs are better than others
  o “better” means, for example:
    ▪ more efficient in space or time required (traditional criteria)
    ▪ more robust, the “ilities” – usability, maintainability, extensibility, scalability…

● These are central concerns of Software Engineering
  o discussed in detail in CS32 (CSCI0320)
Design in a Nutshell (3/3)

- There are trade-offs to make everywhere
  - architect balances aesthetics, functionality, cost
  - mechanical engineer balances manufacturability, strength, maintainability, cost

- Need to defend your trade-offs
  - no perfect solution, no exact rules
  - up to now designs rather straight-forward, not concerned about performance because not dealing with larger collections of data
Designing Pattern Bibles

The two “bibles” of Design Patterns:

- **A Pattern Language** by Christopher Alexander (1977)

- **Design Patterns** by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994) (“gang of four”)

Reuse Designs Rather Than Redesign (1/2)

- Libraries are predefined classes that are re-usable
  - components, like architect’s windows and doors
  - examples: cs015.prj, javafx.scene
  - like components, no indication on how to use them in a program
Patterns are more general than libraries
- specify some relationships between classes
- one pattern may represent many interacting classes
- general, so they must be applied to specific problem
- no actual code re-use

Progression in abstraction/generality vs. code re-use
- methods with parameters
- classes with generics
- abstract classes with concrete methods (mix of code-reuse and contract)
- interfaces (no code-reuse, just contract for methods)
- patterns (idea re-use)

Design pattern is defined by name, abstract, and key aspects of a design pattern’s structure
Design patterns you’ve used before (1/3)

- **Pattern Name:** Composition

- **Abstract:** models objects that are composed of other objects or are associated with peer objects

- **Key Aspects:** Factors similar code out into a different class accessed via references in instance or local variables. These references provide access protection through encapsulation
  - components, typically initialized in the constructor, e.g., engine
  - associations to peer objects, e.g., driver (via parameter passing)
Composition

- You’ve used **composition** from the beginning
- Compose one object out of other, more specialized objects
  - factor out similar code into a separate class
  - instantiate an instance of this class if you need that functionality
  - allows specialists to design components that you can build on
    - black boxes that expose only limited functionality
    - this is a form of delegation—don’t rewrite code that specialists can do for you
- Design your classes based on what they do
  - an **Engine** can `startUp()`
  - a **Radio** can `tune()`
- A **Car** class would use instances of these classes to model a Car’s capabilities
  - would contain one or more instances of an **Engine** class, a **Brake** class, etc.
  - **Car** can delegate `startUp()` to the **Engine`
Design patterns you’ve used before (2/3)

- **Pattern name**: Inheritance

- **Abstract**: models objects that share identical capabilities but have some differences

- **Key features**: factors code *up* into a superclass, can extend from this class into more specialized subclasses and (rarely) override parent’s concrete methods
Inheritance

- FruitNinja gave you experience using **inheritance** hierarchies
  - e.g., `CS15Fruit` and `Fruit`, `CS15Blade` and `Blade`...
- Inheritance models an “is-a” relationship
  - identical capabilities shared by classes factored into superclass
- Design your classes based on what they are
  - a `CS15Mobile` is a `Car`
  - a `CS15Student` is a `BrownStudent`
Design patterns you’ve used before (3/3)

- **Pattern name**: Interfaces

- **Abstract**: models objects that share similar capabilities but implement them in different ways

- **Key features**: declares a contract that implementing classes must obey. Classes must implement all methods declared in an interface
Interfaces

- FruitNinja also gave you experience implementing interfaces
  - i.e., Choppable interface for Fruit and Bomb
- Interfaces model an “acts-as” relationship
  - group together similar capabilities/functions of different classes
- Design your classes based on what they do
  - a Car acts as a Transporter
  - a Laptop acts as a Typeable
public class Pianist extends Musician {
    private Piano _piano;
    public Pianist (Piano p) {
        _piano = p;
    }
}

Which design pattern(s) does this code use?
A. Inheritance
B. Composition
C. Both
D. Neither
Inheritance, Interfaces, Polymorphism (1/6)

- This lecture is about reinforcing what you learned in FruitNinja on how to combine interfaces, inheritance, and polymorphism. We will also cover unintended consequences of overriding in inheritance.
- Let’s return to our Race example from the Interfaces and Inheritance lectures:
  - Lucy and Angel want to race to the CIT using any transportation:
    - Bike
    - Convertible
    - CS15Mobile
    - Van
    - Hoverboard
    - HorseAndBuggy
    - PogoStick
Inheritance, Interfaces, Polymorphism (2/6)

- Right now, we have two relationships to model
  - lots of different methods of transportation that move in different ways
    - Transporter interface
  - a few methods of transportation that move in really similar ways
    - Car superclass

- We need a way to refer to both models at once
  - then use polymorphism and Racer’s useTransportation() method
  - start by making Car superclass implement Transporter
• If Car implements Transporter, then Van, CS15Mobile, and Convertible can be considered Transporter objects
• The “lowest common denominator” between all our transportation classes will be Transporter
Inheritance, Interfaces, Polymorphism (4/6)

- Recall from Interfaces lecture, where we had Bikes, Skateboards, etc., that we used Transporter as lowest common denominator, and therefore as the parameter type in Racer’s `useTransportation()` method

```java
public class Racer {

    public Racer() {
    }

    public void useTransportation(Transporter transport) {
        transport.move();
    }
}
```
Inheritance, Interfaces, Polymorphism (5/6)

- Let's modify Car to implement Transporter, i.e., implement move()

```java
public class Car implements Transporter {
    public Car() {}

    public void drive() {
        //code elided
    }

    @Override
    public void move() {
        this.drive();
    }

    //other methods elided
}
```
Inheritance, Interfaces, Polymorphism (6/6)

- Why is this a good design?
  - Car superclass allows us to have code re-use
    - no repetitive code
  - Transporter interface ensures that every transportation class has a move() method
    - this interface & polymorphism means you need only one method in the Racer class
- By combining interfaces, inheritance, and polymorphism, we have an extensible design for our program
  - very easy to have a race with various different transportation modes
Is this legal?

`Transporter convertible = new Convertible(); _lucy.useTransportation(convertible);`

`Car convertible = new Convertible(); _lucy.useTransportation(convertible);`

`Convertible convertible = new Convertible(); _lucy.useTransportation(convertible);`

Even though `convertible` is stored as a `Convertible`, the compiler knows that `Convertible` as a `Car` implements `Transporter`, so this is still acceptable.
Inheritance, Composition, and Your Design

- When designing a program, you should ask these questions:
  - **before designing**: How extensible do I want my design to be?
    - if someone decided to add to my project, should they be able to change it a lot or a little? (e.g., adding new types of transportation like flying cars)
  - **after designing**: How extensible is my design?
    - if someone decided to add to my project, would it be easy or hard for them to do so?

- The answers will help you decide whether to use **inheritance** and/or **interfaces** in addition to **composition**
  - the real world is almost always modeled by component hierarchies, hence composition is ubiquitous
Case Study: Design these Cars

Van

CS15Mobile

Convertible
How extensible should this design be?

- It should be easy to add different types of cars to this program
  - easy → adding one or more classes does not result in a lot of changes to the original design
Detailed Look at Original Design

- **CS15Mobile, Van, and Convertible** have many identical capabilities and share a lot of the same components
  - start/stop engines
  - play radio

- Let’s create a **Car** superclass
  - Car contains instances of Engine, Radio, Brake, etc.
  - CS15Mobile, Van, and Convertible extend from Car
How Extensible is this Design?

- Let’s add a Jeep class
  - do we need to change the Car superclass or the other subclasses in any way?
  - nope! Jeep will extend from Car and use the same Engine, Radio, Brake, etc. that CS15Mobile, Van, and Convertible do (ok, not in real life 😊)

- Let’s add an ElectricCar class
  - do we need to change the Car superclass or the other subclasses in any way?
  - yes
    - an ElectricCar doesn’t use the standard Engine inherited from Car
    - let’s modify our design to allow for electric cars
Modifying the Design

- Why can’t ElectricCar just override Car’s methods that make use of Engine?
  - you could do this, but it could be dangerous
  - when you subclass Car, the instance of Engine, _engine, is hidden from you
    - a parent’s private variables stay private
  - you inherit these methods that use _engine, but their implementation is hidden from you
    - you do not know which methods use _engine, let alone how they do that
Ex: Unintended Consequences of Overriding (1/3)

- Assume `Car` uses its method `revEngine()` (which uses `Engine`'s `rev()`) inside its definition of `drive`

```java
public class Car {
    private Engine _engine;
    private Brakes _brakes;
    public Car() {
        _brakes = new Brakes();
        _engine = new Engine();
    }

    public void revEngine() {
        _brakes.engage();
        _engine.rev();
    }

    public void drive() {
        this.revEngine();
        _brakes.disengage();
        //remaining code elided
    }
}
```

```java
public class Brakes {
    public void engage() {
        //code elided
    }

    public void disengage() {
        //code elided
    }
}
```

```java
public class Engine {
    public void rev() {
        //code elided
    }
}
```

Andries van Dam © 2019 10/17/19
Ex: Unintended Consequences of Overriding (2/3)

- Now we override `revEngine` in `ElectricCar`
  - notice `revEngine` no longer calls `brakes.engage()`
- Recall that `drive()` calls `revEngine`; if you call `drive()` on `ElectricCar`, it will call `Car`'s inherited `drive()` that uses `ElectricCar`'s `revEngine` implementation

```java
public class ElectricCar extends Car {
    private Battery _battery;
    public ElectricCar() {
        super();
        _battery = new Battery();
    }

    @Override
    public void revEngine() {
        _battery.usePower();
    }
}
```
Ex: Unintended Consequences of Overriding(3/3)

- This could pose a problem
  - `drive()` relies on `revEngine` to engage the brakes, so that `drive()` can disengage them, but you don’t know that – hidden code
  - `ElectricCar` also has 2 engines now
    - its own `Battery` and the pseudo-inherited `_engine` from `Car`

- It might be fine if you write all your own code and know exactly how everything works

- In CS15 we don’t use inheritance to extend `javafx` or other library classes; instead extensive use of interfaces
  - can’t know implementation and don’t want to cause unintended consequences

```java
public class Car {
    // code elided
    public void revEngine() {
        _brakes.engage();
        _engine.rev();
    }
    public void drive() {
        this.revEngine();
        _brakes.disengage();
        // remaining code elided
    }
}
```

```java
public class ElectricCar extends Car {
    private Battery _battery;
    public ElectricCar () {
        _battery = new Battery();
    }
    @Override
    public void revEngine() {
        _battery.usePower();
    }
}
```
TopHat Question

public class BrownClass {
    private Student[] _students;
    //code elided
    public Student getStudent(int index) {
        if(index < 0 || index >= _students.length) {
            System.out.println("you are trying to access an incorrect index");
            return null;
        }
        return _students[index];
    }
}

public class CS15Class extends BrownClass {
    private CS15Student[] _cs15Students;
    //code elided
    @Override
    public Student getStudent(int index) {
        return _cs15students[index];
    }
}

What would happen if the following code is called?

CS15Class cs15 = new CS15Class();
cs15.getStudent(-1);

A. A CS15Student is returned
B. An array out of bounds error is thrown
C. Null is returned
D. A Student is returned
Unintended Consequences of Overriding

- Parent method may implement some form of error checking that is private, so you can’t know about it and may lose functionality and safety
Modifying the design

- Furthermore, is ElectricCar really much of an exception?
  - a Convertible would use a TurboEngine, not the standard Engine in Car that a CS15Mobile might use
  - is our inheritance hierarchy still a natural fit?
    - if we focus purely on engine-related methods, doesn’t seem like it
    - but we may still want to use inheritance for methods that don’t need to be overridden
So... Where do we go from here?

- We want to model a series of different Cars
  - they have some similar capabilities, but the implementations of these capabilities use different components
  - e.g., all Cars need to know how to `startUp()`, but they may `startUp()` in different ways, using different engines
    - an ElectricCar would `startUp()` differently from a CS15Mobile
- We want to model same capability, implemented in different ways
  - abstract methods
  - interfaces if we see same problem for most, let alone all, methods
Abstract Classes vs. Interfaces (1/3)

- At what point does an interface become preferable to an abstract class?
- Easier to motivate with a deeper hierarchy, e.g., animals, mammals, dogs
- Let’s start by modeling a Dog hierarchy
  - many dogs do the same things in the same way
  - this is a typical inheritance hierarchy → no abstract classes or interfaces required

```
Dog
  eat()
  drink()
  sleep()
  fetch()

Poodle
Labrador
Pug
```
Let’s be more generic and try to model Mammals

- not every Mammal does everything the same way
- however, they all reproduce() and protectOffspring() in the same way
  - use an abstract class with these concrete methods
Let’s be even more generic and model **Animals**
- while animals share capabilities, they do not typically implement them in the same way
- could have an entirely abstract class or an interface
Remember...

- Only difference between an entirely abstract class and an interface is that interface will never have instance variables.
- Thus we favor interfaces over wholly or even mostly abstract superclasses.
- But as Java evolves, interfaces are becoming more concrete:
  - In a recent release of Java, you can have concrete methods in interfaces.
  - CS15 will not make use of this feature.
Interfaces or Abstract Classes?

• When should you still use inheritance with an abstract superclass and when should you use an interface?
  o discuss with a partner for a minute!
Inheritance of Abstract Classes vs. Interfaces

• In CS15:
  o use interfaces when a class is so abstract that there is essentially no code re-use, i.e., no concrete methods
  o use inheritance with an abstract superclass if you don’t need to override concrete methods – overriding can have potentially unwanted consequences
    ▪ overriding, and especially partially overriding, are still useful design choices, but must be done very carefully and sparingly
    ▪ If you are the designer of both the superclass and subclasses, you should be able to guarantee safety; but if you’re overriding someone else’s concrete methods, it may well be too risky
Back to Our Car Design

- The cars we want to model share a lot of same capabilities
  - can still factor this re-usable code up into a **superclass**
- Abstract superclass thus is better for this common case
  - can still define concrete methods
    - these cars still have some identical capabilities (with the way we intend for them to be used for this program)
    - e.g., playing the radio, braking, etc.
  - only methods that involve specialized components and/or processing logic are abstract
- So what does this design look like?
The Code

Our code might look like this, with lots of instance variable and concrete methods elided!

```java
public abstract class Car {
    public Car() { //code elided}
    public abstract void startUp();
    public abstract void turnOff();
    public void brake() { //code elided }  // More concrete methods elided }

public class ElectricCar extends Car {
    private Battery _battery;
    public ElectricCar() {
        //instantiation elided
    }
    @Override
    public void startUp() {
        _battery.powerUp();
    }
    @Override
    public void turnOff() {
        _battery.powerDown();
    }
}

public class CS15Mobile extends Car {
    private Engine _engine;
    public CS15Mobile() {
        //instantiation elided
    }
    @Override
    public void startUp() {
        _engine.turnOn();
    }
    @Override
    public void turnOff() {
        _engine.turnOff();
    }
}

public class Convertible extends Car {
    private TurboEngine _turboEngine;
    public Convertible() {
        //instantiation elided
    }
    @Override
    public void startUp() {
        _turboEngine.purr();
    }
    @Override
    public void turnOff() {
        _turboEngine.turnOff();
    }
}
```
How extensible is this design?

- Let’s add a `HybridCar` class
  - uses both battery and a standard engine
  - will we need to change the design in any way?
    - nope!
    - `HybridCar` will implement its own `startUp()` and `turnOff()` → the inheritance hierarchy fits

- `Car` leaves the implementation of `startUp()` and `turnOff()` to its subclasses
  - more extensible than before
  - the superclass mandates that the capability must exist, not how it must be implemented

- You can add more types of cars inheriting many concrete methods, without overriding them, and overriding only the few abstract methods
Composition

● Guidelines on using composition with inheritance
  o the superclass should not implement functionality unless all subclasses have the same functionality and perform it in the same way – then it can be factored up into the superclass
  o the superclass should instead lay out a list of capabilities
    ▪ this means your superclass often will be an abstract class
  o the implementation of these capabilities is up to subclasses
    ▪ the subclass can then use composition to implement these capabilities
    ▪ all subclasses contain their proper Engine or Battery (composition)
    ▪ Van and CS15Mobile both use standard Engine to startUp()
    ▪ Convertible uses TurboEngine to startUp()
    ▪ ElectricCar and HybridCar respectively use a Battery and Battery + Engine to startUp()
Reviewing Composition

- “Van and CS15Mobile both use standard Engine to `startUp()`”

Van and CS15Mobile are subclasses of Car

This is common code of a component that has been factored out into a separate class (Engine). Van and CS15Mobile explicitly create and contain an instance of Engine - Composition

`startUp()` is a capability the superclass (Car) has declared. Van and CS15Mobile are using their instance of Engine to define this capability
public class Cupcake {
    //constructor and other methods elided
    public void addChocolate() {
        this.setFlavor(Constants.CHOCOLATE);
        this.setSugarPerServing(this.getSugarPerServing() + Constants.CHOCO_SUGAR_SERVING);
    }
}

This codes a superclass Cupcake for potential subclasses, but as of now you only want to create ChocolateCupcake and VanillaCupcake subclasses. Should this code be changed?
A. No – superclass contains all identical functionality for each of its subclasses
B. Yes – superclass should have an addVanilla() method
C. Yes – superclass shouldn't have addChocolate() since only one subclass uses it
D. Yes – superclass should keep addChocolate() since you might make more chocolate flavored subclasses
Explanation of the Cupcake Question

● While any answer could be correct, we prefer not to be permissive and factoring any code that could potentially be useful to multiple children up into the superclass

● Instead, we advocate only factoring up methods that the superclass wants to enforce having its children inherit
  o thus, neither addChocolate() nor addVanilla() should be in the Cupcake superclass
Further Features: Radio

- Let’s say that all types of Cars have Radios.
- However, CS15Mobile and Convertible also have USB ports so you can plug in your phone.
- If we wanted these properties and their features, where should we put them?
Shared Capabilities: Radio

- Will every subclass use this?
  - If we assume that every current and future car will have a radio,
  - and if all subclasses share play capability, add `_radio` and concrete `playRadio()` method to the superclass

- Alternatively, when should you make `playRadio()` an abstract method? (by extension, `Car`, an abstract class?)
  - if you want subclasses to use different types of radios
  - subclasses will use their own instance of a `Radio` to define that capability (composition)

```
Car
abstract playRadio()
...
```
Shared Capabilities: **USB**

- Will every **subclass** use a **USB**?
  - **nope**
    - e.g., **Van** does not have a **USB** port
  - **do not** add this instance variable and its capability to **superclass**

- Instead, use composition to include this capability in **Convertible** and **CS15Mobile**
Avoid “The Bucket Class”

- It’s easy to fall into the following trap:
  - “some of my subclasses use this functionality, so let me throw it into the superclass so the subclasses get code re-use”
  - your superclass can very easily become a bucket or a “god class”
  - this is not the purpose of inheritance
    - every subclass must need the functionality factored into the superclass

- Bucket problem is also called the gorilla-banana problem
  - “you request a banana, and end up with a gorilla holding the banana and the whole jungle with it”
Careful... (1/2)

- When thinking about design, it’s hard not to go overboard
  - leads to really complex classes that may include unnecessary functionality
  - makes it hard for people to build on top of what you’ve created

“I don’t know what my users will want, so I’m going to design and implement every possible feature”
Careful... (2/2)

- When thinking about design, it’s hard not to go overboard
  - this leads to your `Car` superclass only having a concrete `drive()` method
    - if you only have concrete methods, you will lose the advantage of polymorphism
      - without a `startUp()` method defined in the `Car` superclass, if `Car` is the type of your parameter, can we call `startUp()` on a `CS15Mobile`?
      - Nope! Limitation of polymorphism
    - add abstract methods for any capabilities you want to ensure your children to have. Then polymorphism will work fine, but each child has to provide an implementation for each abstract method

“I don’t know what my users will want, so I’m not going to design further than the most basic requirement”
We want to use the following code

```java
private TimeHandler implements EventHandler<ActionEvent> {
    private Clock _clock;
    public TimeHandler(Clock c) {
        _clock = c;
    }

    @Override
    public void handle(ActionEvent e) {
        _clock.tick();
    }
}
```
Case Study 1 (2/2)

- Will both of these solutions work if we pass in a `GrandfatherClock` object to `TimeHandler(…)` in the previous slide?

```java
public class Clock {
    public Clock () {//code elided}
    public void tick() { // code to update time}
}

public class GrandfatherClock extends Clock {
    public GrandfatherClock () {//code elided}

    @Override
    public void tick() {
        super.tick();
        if (this.isEvenHour()) {
            this.playDing();
        }
    }
}

public interface Clock {
    public void tick();
}

public class GrandfatherClock implements Clock {
    private TimeKeeper _timer;
    public GrandfatherClock() {
        _timer = new TimeKeeper();
    }

    @Override
    public void tick() {
        _timer.updateTime();
        if (this.isEvenHour()) {
            this.playDing();
        }
    }
}
```
Different Implementations, Same Result

● Both of these implementations result in a \texttt{GrandfatherClock} animating correctly
  o in solution 1, \texttt{Clock} is a superclass
  o in solution 2, \texttt{Clock} is an interface
  o both can be used polymorphically

● But pros and cons to each solution
Inheritance Design: Pros and Cons

Pros:
- Better code reuse
  - by extending superclass, methods are automatically inherited in subclasses, so no need to re-implement functionality (tick(), in this case, where it is simple but it could be arbitrarily complex. It delegates via composition to a TimeKeeper)

Cons:
- Less flexible
  - forced to accept superclass properties, difficult to make changes, may have to (partially) override concrete methods
- Problematic to extend a superclass someone else wrote
  - because you don’t know how hidden functionality in superclass will affect your code
Interfaces + Composition

- Solution 2 uses a combination of an interface and composition to delegate functionality to _timer

- GrandfatherClock signs the contract (agrees to have tick() functionality) but delegates most of the responsibility to another class

```java
public interface Clock {
    public void tick();
}

public class GrandfatherClock implements Clock {
    private TimeKeeper _timer;
    public GrandfatherClock() {
        _timer = new TimeKeeper();
    }

    @Override
    public void tick() {
        _timer.updateTime();
        if(this.isEvenHour()) {
            this.playDing();
        }
    }
}
```
Interfaces+Composition Design Pros

- Very flexible
  - we completely control GrandfatherClock, and if we want to write a CuckooClock or DigitalClock class, it’s easier to implement that functionality
  - no overriding → no unintended consequences

- Easy to use classes written by others
  - if someone else wrote TimeKeeper, you can still delegate to it without knowing its code details
  - could also easily swap it out with a different component class that you wrote
Interfaces+Composition Design Cons

- Cons
  - both inheritance and interface use composition (i.e. delegate to other objects), but with inheritance you automatically get concrete methods from the superclass, whereas when you do explicit composition, you have to invoke the methods you want on the object to which you have delegated
Case Study 2 (1/2)

- Imagine we’re making a game
  - FlyingSuperhero
    - fly()
    - saveLives()
  - StrongSuperhero
    - liftCars()
    - saveLives()
  - SlimeMonster
    - scareCitizens()
    - oozeSlime()
  - Robber
    - scareCitizens()
    - robBank()
Case Study 2 (2/2)

- There are some similarities in implementation
  - FlyingSuperhero and StrongSuperhero both have a `saveLives()` method
  - SlimeMonster and Robber both have a `scareCitizen()` method
  - Let’s abstract this up into superclasses!
New Design

Hero
  saveLives()

FlyingSuperHero
  fly()

StrongSuperHero
  liftCars()

Villain
  scareCitizens()

SlimeMonster
  oozeSlime()

Robber
  robBank()
New Character

- We want to add a monster who flies
  - `FlyingMonster`
    - `fly()`
    - `scareCitizens()`

- Where do we fit this into our inheritance diagrams?
  - it can fly, but it does not save lives → won’t use methods defined in `Hero` superclass so that it can use `scareCitizens()`
  - could extend `Villain` superclass so that it can use `scareCitizens()`, but would need to reimplement code for `fly()`
New(er) Design

Hero
\textit{saveLives()}

\begin{itemize}
  \item FlyingSuperHero
    \textit{fly()}
  \item StrongSuperHero
    \textit{liftCars()}
\end{itemize}

SlimeMonster
\textit{oozeSlime()}

Villain
\textit{scareCitizens()}

Robber
\textit{robBank()}

FlyingMonster
\textit{fly()}

Andries van Dam © 2019 10/17/19
Let’s break this down

● **Separate classes by their capabilities**
  o *FlyingSuperHero*: flier + lifesaver
  o *StrongSuperHero*: carlifter + lifesaver
  o *SlimeMonster*: slimer + scarer
  o *FlyingMonster*: flier + scarer
  o *BankRobber*: robber + scarer

● **Inheritance**: model classes based on what they are
● **Composition**: model classes based on what they do
  o in this case, prefer composition over force-fitting inheritance
New(est) Design: Mix and Match Using Interfaces

- **FlyingSuperHero**
  - Flier
  - Lifesaver

- **StrongSuperHero**
  - CarLifter
  - Lifesaver

- **FlyingMonster**
  - Flier
  - Scarer

- **SlimeMonster**
  - Slimer
  - Scarer

- **BankRobber**
  - Robber
  - Scarer
Composition and Our Design

- As you can see, there are a lot more classes in this design
  - however we have extreme flexibility
    - could make a flying, strong, scary, bank robbing monster without changing or force fitting our new class into the current design
    - although you still have to implement the methods of the interface in your new class
Design Patterns (1/2)

- Serve as examples of good design
  - there are no “hard and fast” rules
  - there are concrete trade-offs to think about
  - they are tools to help you build your own designs

- Provide common vocabulary to discuss design at a more abstract level
  - give us a concise way to describe complex object interaction
  - discuss design at higher level because we don’t need to describe every object in the program
Design Patterns (2/2)

● Must be adapted to your program specification
  o may need to add extra relationships to your structure to augment a design pattern
  o may need to create a new pattern because none exists that exactly fits your needs

● Should be used in moderation
  o consider trade-offs carefully before using a pattern
  o consider added complexity—is it needed in your model?
The Challenges of Design (1/3)

- Design a solution to a problem such that it solves the problem efficiently, but also makes it easy to extend the solution to other problems
  - only define the capabilities that you know you will need to solve the problem at hand

- Your job in creating an interface/superclass is precisely to figure out the right abstractions
  - decision making under uncertainty—you have to do the best you can. And frankly, opinions may differ on what is “the best solution”
  - and experience (practice) really matters
The Challenges of Design (2/3)

- Extensibility is important, but only to a degree
  - you cannot design a program that solves every problem a user thinks of

- Define a scope for your project and defend it to yourself
  - “I am modeling a diverse number of cars for a racing game”
  - “I am modeling Ford cars for a program only used by Ford dealerships”

- You are designing this program to solve an existing problem
  - the abstractions you make via interfaces and inheritance are meaningful to this problem
  - a good design will solve similar problems that fit the existing pattern
The Challenges of Design (3/3)

- CS32 (Software Engineering) goes deeper into design decisions and tradeoffs
  - you can take it after you’ve completed CS15/CS16!
Announcements

• Cartoon deadlines:
  o On-time: tonight 11:59pm
  o Late: Saturday 10/19 11:59pm

• DoodleJump is out Saturday!
  o Mini assignment due Monday
  o You will have 2 weeks to do this assignment- make use of this time and start early, start today, start yesterday!!
  o Important note: TAs have the right to prioritize Cartoon questions over DoodleJump questions