Overview

- What are Patterns?
- Composition/Interfaces, Inheritance, Polymorphism
Design in a Nutshell (1/3)

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

● Design Patterns are proven solutions to common problems
  ○ used successfully before by others, refined by experience
  ○ generally language-independent – learn once, apply everywhere
Increasingly we learn about good design

- some designs are better than others
- “better” means, for example:
  - more efficient in space or time required (traditional criteria)
  - more robust, the “illities” – usability, maintainability, extensibility, scalability…
- these are central concerns of Software Engineering
  - discussed in detail in CS32 (CSCI0320)
Design in a Nutshell (3/3)

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

● Need to defend your trade-offs
  o no perfect solution, no exact rules
  o up to now, designs have been rather straight-forward, and we’ve not worried about performance since we haven’t yet dealt with potentially large collections of data
Designing Pattern Bibles

The two “bibles” of Design Patterns:

- **The Timeless Way of Building** by Christopher Alexander (1979)
  - design patterns in architecture (e.g., house styles: Craftsman, Colonial, Cape Cod Saltbox, Contemporary…)
  - Alexander’s patterns in architecture initiated the study of design patterns in software

- **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 you can reuse
  
  - components, like architect’s windows and doors
  
  - examples: cs015.prj, javafx.scene, Demos.Cars
  
  - like components, no indication on how to use them in a program
Reuse Design Rather Than Redesign (2/2)

● Patterns are more general than libraries
  o specify some relationships between classes
  o one pattern may represent many interacting classes
  o general, so they must be applied to specific problem
  o no actual code re-use

● Progression in abstraction/generality vs. code re-use, from concrete classes to abstract classes, to interfaces (no code-reuse, just contract for methods) to 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

- Key Aspects: Factors similar code out into a different class, store components as instance variables, initialize them in the constructor through instantiation or association (via parameter passing) and provide access protection through encapsulation
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 Door 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 methods
Inheritance

- TA SafeHouse gave you experience using **inheritance** hierarchies
- 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
Clicker Question

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)

- How can we use interfaces, inheritance, and polymorphism singly or in combination?
- Let’s return to our Race example from the Interfaces and Inheritance lectures
  - Sophia and Dan 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 can utilize 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() {} //code elided

    public void drive() {
        //code elided
    }

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

    //other methods elided
}
```
Is this legal?

Transporter bike = new Bike();
_sophia.useTransportation(bike);

Transporter convertible = new Convertible();
_sophia.useTransportation(convertible);

Convertible convertible = new Convertible();
_sophia.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, Interfaces, Polymorphism (6/6)

- Why is this a good design?
  - Car superclass allows us to have code reuse
    - no repetitive code
  - Transporter interface ensures that every transportation class has a move() method
    - this interface & polymorphism means 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
Inheritance, Composition, and Your Design

• When designing a program, you should ask these questions:
  o 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?
  o 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
  o 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
  - lock/unlock doors
  - play radio

- Let’s create a **Car** superclass:
  - **Car** contains instances of Engine, Door, 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, Brake, Door, 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 the 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 the method revEngine 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 Brakes(){
        //constructor elided
    }
    public void engage() {
        //code elided
    }
    public void disengage() {
        //code elided
    }
}
```
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`; it will use `ElectricCar`'s `revEngine` implementation

```java
public class ElectricCar extends Car {
    private Battery _battery;

    public ElectricCar (){
        _battery = new Battery();
    }

    @Override
    public void revEngine()
    {
        _battery.usePower();
    }
}
```

```java
public class Car {
    //code elided
    public void drive()
    {
        this.revEngine();
        _brakes.disengage();
        //remaining code elided
    }
}
```
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
  - 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 extend javafx or other library classes
  - lack information on construction 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();
    }
}
```
Clicker 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];
    }
}

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
Modifying the design

• Furthermore, is ElectricCar really that much of an exception?
  o a Convertible would use a TurboEngine, not the standard Car Engine a CS15Mobile might use
  o is our inheritance hierarchy still a natural fit?
    ▪ doesn’t seem like it 😐
So… Where do we go from here?

- We want to model a series of different Cars
  - they have some similar capabilities, but the implementation of these capabilities use different components
    - an ElectricCar would startUp() differently from a CS15Mobile
- All Cars need to know how to startUp(), but they may startUp() in different ways
  - the same capability, implemented in different ways → interfaces or abstract methods!
Interfaces or Abstract Classes?

- For this situation, is it better to have an interface or a partially abstract/partially concrete class?
  - discuss with a partner for a minute!
Abstract Classes vs. Interfaces (1/3)

- At what point does an interface become preferable to an abstract class?
- Consider 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
Abstract Classes vs. Interfaces (2/3)

- Let’s be more generic and try to model Mammals
  - not every Mammal does everything the same way
  - however, they all reproduce() and growHair() in the same way
    - use an abstract class with these concrete methods
Abstract Classes vs. Interfaces (3/3)

- 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.
- But as Java evolves, interfaces are becoming more concrete:
  - in Java’s latest release, can have concrete methods in interfaces.
  - CS15 will not make use of this new feature.
Back to Our Car Design

- The cars we want to model share a lot of the same capabilities
  - can still factor this code up into a **superclass**
- Abstract class is better
  - can still define concrete methods
    - these cars still have some identical traits (with the way we intend for them to be used for this program)
    - playing the radio, braking, opening and closing doors, etc.
  - only methods that involve specialized components are abstract
- So what does this 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 }
        public void lockDoors() { //code elided }
        public void unlockDoors() { //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 batteries 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 without overriding many methods in the `Car` superclass
Composition

- Guidelines on using composition with inheritance
  - the superclass should not implement functionality unless all subclasses have the same functionality and perform it in the same way
  - the superclass should instead lay out a list of capabilities
    - this often means your superclass will be an abstract class
  - the implementation of these capabilities is up to subclass
    - 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 resp. use a Battery and Battery + Engine to startUp()
Composition – Breaking it Down

- “Van and CS15Mobile both use 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.

startUp() is a capability the superclass (Car) has declared. Van and CS15Mobile are using their instance of Engine to define this capability.
Clicker Question

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. You 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.
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?
  - it’s safe to assume that every car on the road has a radio
  - if all subclasses share this capability, add _radio and non-abstract 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 this?
  - nope
    - ex. 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
  
  - This 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 `drive()` method
  - you will lose some of the advantages of polymorphism
    - Without a `startUp()` method defined in the `Car` superclass, can we call `startUp()` polymorphically?
    - Nope! It explicitly becomes more work

“I don’t know what my users will want, so I’m not going to design further than the most basic requirement”
Case Study 1

- 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

- 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 GrandfatherClock animating correctly
  - in solution 1, Clock is a superclass
  - in solution 2, Clock is an interface
  - both can be used polymorphically

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

Pros:
● Better code reuse
  o by extending superclass, methods are automatically inherited in subclasses, so no need to re-implement functionality

Cons:
● Less flexible
  o forced to accept superclass properties, difficult to make changes, may have too many (partial) overrides
● Problematic to extend a superclass someone else wrote, easy to implement someone’s interface
  o 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();
        }
    }
}
```
• **Very flexible**
  o we completely control *GrandfatherClock*, and if we want to write a *CuckooClock* or *DigitalClock* class, it’s easier to implement that functionality
  o no overriding → no unintended consequences

• **Easy to use classes written by others**
  o if someone else wrote *TimeKeeper*, you can still delegate to it without knowing its code details
  o could also easily swap it out with a different component class that you wrote
• Cons
  o no code reuse
    ▪ if we have a lot of different Clocks that only slightly vary, there will be some repetition between classes
    ▪ however, minor repetition is preferable to force-fitting an inheritance hierarchy
Clicker Question

What is the relationship between a MacBook and Laptop?

A. Inheritance: a Laptop "is a" MacBook

B. Inheritance: a MacBook "is a" Laptop

C. Composition: a MacBook does what a Laptop does

D. Composition: a Laptop does what a MacBook does
Case Study 2

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

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

Hero
- saveLives()

FlyingSuperHero
- fly()

StrongSuperHero
- liftCars()

SlimeMonster
- oozeSlime()

Robber
- robBank()

Villain
- scareCitizens()
New Character

• We want to add a monster who flies
  o FlyingMonster
    ▪ fly()
    ▪ scareCitizens()

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

- **Hero**
  - saveLives()
- **FlyingSuperHero**
  - fly()
- **StrongSuperHero**
  - liftCars()
- **SlimeMonster**
  - oozeSlime()
- **Villain**
  - scareCitizens()
- **Robber**
  - robBank()
- **FlyingMonster**
  - fly()
Let’s break this down

- **Separate classes by their capabilities**
  - FlyingSuperHero: flier + lifesaver
  - StrongSuperHero: carlifter + lifesaver
  - SlimeMonster: slimer + scarer
  - FlyingMonster: flier + scarer
  - Robber: bankRobber + scarer

- **Inheritance**: model classes based on what they are

- **Composition**: model classes based on what they do
  - in this case, prefer composition over force-fitting inheritance
New(est) Design

- FlyingSuperHero
  - Flier
  - LifeSaver
- StrongSuperHero
  - CarLifter
  - LifeSaver
- FlyingMonster
  - Flier
  - Scarer
- SlimeMonster
  - Slimer
  - Scarer

- Robber
  - BankRobber
  - 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
Design Patterns...

- 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 a higher level because we do not need to describe every object in the program

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

- Should be used in moderation
  - consider trade-offs carefully before using a pattern.
  - 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
  o 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
  o decision making under uncertainty - you have to do the best you can. And frankly, opinions may differ on what is “the best solution”
  o 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 on time deadline is tonight at 11:59pm
- Late deadline is Friday at 10pm
- DoodleJump is released!
- DoodleJump mini-assignment is due Monday at 2pm