Advanced OOD and the Strategy Pattern

I spent the first part of the week installing our new productivity software.

Then I used the rest of the week trying to make it interface with our time reporting system.

So far all it can do is tell me how much time I’m wasting in this meeting.
In our last episode...

(It was all about Object-Oriented Design)
A Concise Theory of Object-Oriented

- Object represents a “thing”
  - person, car, date, ...
  - (not two things, not ½ thing)

- Object responds to messages
  - (method calls)
  - Things it does to itself (SRP)
  - That is, other objects ask the object to do something to itself

- Objects are “opaque”
  - Can’t see each others’ data-vars
  - Messages (calls) are only way to get things done
Because objects are completely opaque, we don’t need to know what’s really inside them
- Each car object could be implemented with its own unique code

If two cars behave the same, then really should have same code
- Otherwise would violate DRY
- And a huge amount of coding work

So all cars are made from a common car template
- Template = class
- The car template is not a car, it’s a “blueprint” for a car
- Helps satisfy DRY
Single Responsibility Principle

- **Single Responsibility Principle (SRP)**
  - Each class should be responsible for one thing (capability, entity, computation, etc.)
  - Can phrase this as “mind your own business”
    - object do its own calculations
    - object should not do calculations for another
  - Easy to violate this because objects need to be connected to one another
    - e.g., Events happen as part of Dates
Object-Oriented Design, “wuggy style”

As a user
I want see my nearby friends on the map
So that I know who to contact for a spontaneous meet-up

1. Identify the “things”
   - me (previous user story)
   - friends, friend
   - map (previous, built-in)

2. Identify the messages
   - me.logged-in (previous)
   - friend.location-known
   - friend.location
   - friend.online
   - friend.near(me)
   - friend.image

3. Assemble into classes

4. Right things call right messages (TBD)
Object-Oriented Typing

Theory of Object-Oriented, III
Typing and Subtyping: “is a”

- In Object-Oriented, it’s all about behavior
- A particular behavior is described by a type
  - Mammalian behavior is the Mammal type
  - Tiger behavior is the Tiger type
- Because a Tiger always behaves like a Mammal, we say that Tiger is a subtype of Mammal

- Feline is a subtype of Mammal
- House Cat is a subtype of Feline
- s Subtype c means all s’s always act like a c but not all c’s act like an s
  - All Felines are Mammals
  - But not all Mammals are Felines
- Means that Feline object will always work wherever a Mammal is needed
Feline a *subtype* of Mammal means Mammal a *supertype* of Feline

A supertype variable can hold values of any of its subtypes

```
void petFur(Mammal m) {
    int coarseness = m.furValue();
    ...
}
```

- Use of a supertype variable like `Mammal` allows all current and future subtypes of Mammal to be petted
- `petFur` will work for Canine subclass Dog when it is added
- We say that `petFur` is *open for extension*
Three Ways to Declare Subtype in Java

- **Subclass** a class
  - `class Mammal {...}
    class Feline extends Mammal {...}
  - Feline is a **subtype** of Mammal
  - Feline gets all of Mammal’s implementation (can override)

- **Subclassing abstract class** is similar
  - Can’t “new” an abstract C b/c not all methods implemented
  - `abstract class Mammal {...}
    class Feline extends Mammal {...}
  - Feline must implement missing methods of Mammal
  - Abstract class asserts “I’m a category of thing, not a real thing”

- **Interface** has **no** methods implemented
  - Like a totally abstract class
  - `interface Cuddly {...}
    class HouseCat extends Feline implements Cuddly {...}`
More on Interfaces

- A class can extend just one superclass
- But a class can implement multiple interfaces

```java
interface Cuddly {
}
interface Spotted {
}
class Tabby extends Feline implements Cuddly, Spotted {
}
```

- Means interfaces are much more flexible
- Can “mix in” all the supertypes you want
- Not just a tree of types anymore
Not all classes represent a unique type

Maybe want another impl of same thing
  for performance, say

Consider two possible impl’s of Set
  class TinySet implements Set { /* linked list */ }
  class BigSet implements Set { /* uses hard disk */ }
  Identical interfaces, input/output behavior -> same type
  One faster for tiny sets, the other faster for big sets

Use interfaces for implementation alternatives
  Since TinySet real subtype, extends would be misleading
  Since no shared implementation, extends would be misleading
How represent Feline (and Why)?

A. Class
B. Abstract Class
C. Interface
D. Extends

Abstract class allows reusing some common implementation among Felines, and yet cannot be new’d (instantiated). Interface is a reasonable answer, too, as the sharing might not be too great.
A Few Things to Remember

- Avoid subclassing (extends)
  - Use when you have a real type hierarchy
  - And when you are lazy and want to share method implementations
  - But rare that non-leaf classes (like Mammal) can be instantiated (new’ed)

- Subclassing abstract superclass is better
  - Captures “is-a” relationship of real type hierarchy
  - Purpose is still to borrow method implementations
  - But asserts that non-leaf class can’t be new’ed

- Interfaces are the bomb
  - The most flexible
  - When you have multiple supertypes, or
  - When the subclasses are just implementation alternatives
Interfaces are Central to Java

- Java Collections Framework
  - Interface Collection
  - methods like `add(Object)` & `contains(Object)`
  - `interface List extends Collection` ...
  - `interface Set extends Collection` ...
  ...
  - Wherever `Collection` variable appears, allows for a ListArray or HashSet, etc.
  - Requires no particular implementation
  - Super-flexible code
Strategy Pattern

Composition + Delegation + Interfaces
abstract class CarDesc {
    Color paintColor();
    int wheelbase();
    ...
    int tireSize();
    boolean tirePerf();
    String tireSeason(); /* should be enum */
}

abstract class SmallCar extends CarDesc {
    ...
}

class KiaSoul extends SmallCar {
    ...
    int tireSize() { return 165; }
    boolean tirePerf() { return false; }
    String tireSeason() { return "All"; }

A. CarDesc and maybe SmallCar should be an interface

B. All these classes should “implement” a Tire interface

C. KiaSoul should have a Tire field

As implemented, KiaSoul only allows one set of values for tire behavior. By adding a Tire field and delegating to its function, then all sorts of tires will be possible. (See next slide.)

“A” is an OK answer, but it’s not nearly as big a deal as the inflexibility of the tire methods.
```
abstract class CarDesc { ... }
abstract class SmallCar extends CarDesc {
    ...
}

class KiaSoul extends SmallCar {
    Tire tires;
    ...
    int tireSize() { return tires.size(); }
    boolean tirePerf() { return tires.perf(); }
    String tireSeason() { return tires.season(); }
    
    delegates to Tire (easy!)
```

- KiaSoul has Tires
- Delegates tire method impls to Tire
- Tire impl is reusable in other settings
- Can change at runtime: mySoul.changeTires(new DunlopSP());
abstract class CarDesc { ...
abstract class SmallCar extends CarDesc {
...
}
class KiaSoul extends SmallCar {
Tire tires;  
composes Tire ("has-a")
...
int tireSize() { return tires.size(); }
boolean tirePerf() { return tires.perf(); }
String tireSeason() { return tires.season(); }

delegates to Tire (easy!)

KiaSoul has Tires

Delegates tire method impls to Tire

Tire impl is reusable in other settings

Can change at runtime: mySoul.changeTires(new DunlopSP());

Tire should be a:
A. Class
B. Abstract class
C. Interface

Why?
We want to leave KiaSoul open to having all kinds of tires, and an interface is most flexible. An abstract class is OK, too, if you think there is a Tire type hierarchy with significant shared implementation.
I’m going to disagree strongly with the prior semester on this one. MichelinPilot and DunlopSP should be subclasses of Tire. Tire should not be an interface. The “is a” relationship clearly holds and “acts like” is clearly too weak. Just because methods aren’t shared right at this second, doesn’t mean that they won’t be later. Design needs to be principled, not fit the moment. We shouldn’t put to an interface, just because the present requirements require no methods. We certainly can envision plenty. We’ll make a mess if we later need to add roll(), “loseAir()”, “heatUp()”, etc, methods, which would be implanted the same across all tires, and Tire is an interface not a base class.
Widely Used for Algorithm Alternatives
Take-Aways

- Start thinking about objects
- Then think about types
- Then “is a” vs “has a”
- Then, consider the hierarchy
  - Are behaviors re-implemented because they don’t apply to all peer sub-types?
  - Are behaviors appearing where they shouldn’t, or doing things they shouldn’t, by inheritance?
    - Fix by overriding, if necessary
    - Better idea: Can the behaviors be moved down? An intermediary added?
  - Would pure abstract base classes be better served by interfaces [Java]?
    - Same behaviors, but not a common implementation
    - Behavior exhibited by other classes to which neither “as a” nor “is a” apply (just plain different classes)
    - Consider “is a” vs “has a” and then, as a refinement, “is a” versus “act like a”