Tuesday 9/23/14 4156

divide into groups & develop acceptance test cases for last year's prelim asgn

what, if anything, did you find hard? - ask each group

---

design patterns - someone somewhere has already solved your problem or a very similar problem

sometimes this results in a library or framework where you can directly reuse open source code

design patterns are instead a way of organizing (your) code or code interactions, so you're reusing experience not code

patterns are "discovered" rather than "invented", the idea is to find solutions that many different developers have already used on many different projects

Tuesday 9/23/14 4156

three types of design patterns

structural - relationships between entities

creational - provide instantiation mechanisms

behavioral - communications between entities

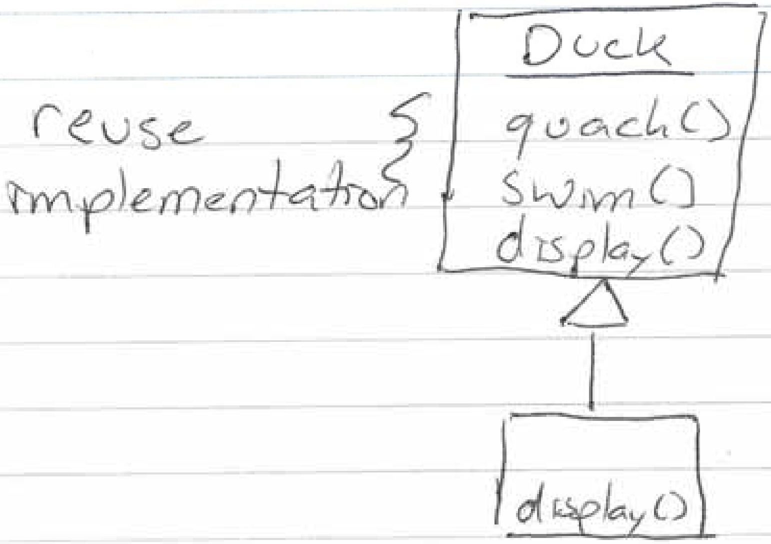
I'll present following the Head First book, but there are many other approaches

many presentations are essentially catalogs - lists of patterns + the contexts in which you should consider using them

for each pattern, book presents with a programming problem, typically first shows a potential solution that seems good but doesn't work for some reason, then ultimately solve using design pattern

Tuesday 9/23/14 4156

duck simulator example

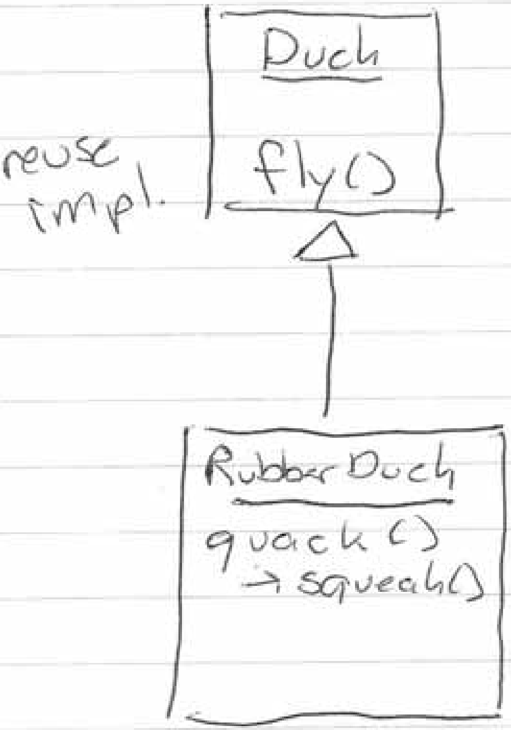


all ducks quack + swim
display is abstract since look different

each duck subclass implements display

new requirement for ducks to fly

1st try - add to parent duck class



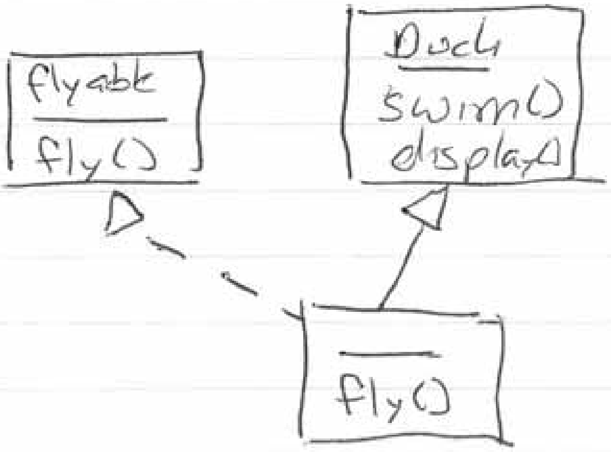
but some kinds of ducks shouldn't fly

so would need to ~~one possibility is~~ override in each relevent subclass - already done for quack

Tuesday 9/23/14 4156

2nd try - separate interface for flyable() & quackable()

would only be implemented by duck subclasses that are supposed to fly or quack



but there will be a LOT of duplicate code since every flying or quacking subclass needs to implement separately

and all that code will need to be changed separately if we ever want to modify flying ~~to~~ or quacking

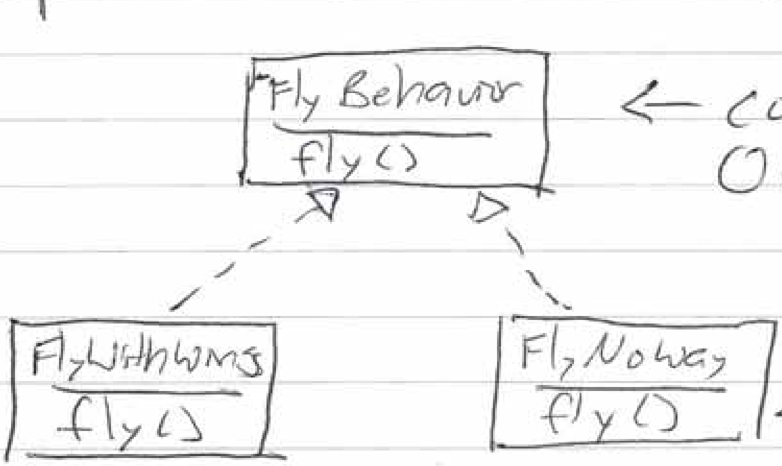
- so inheritance is not a good solution
- & interfaces is not a good solution

Tuesday 9/23/14 4156

need a solution that allows part of a system to vary independent of other parts

design principle - identify the aspects of your application that vary & separate than from what stays the same

since fly & quack vary, but rest of duck does not - except for display, which is indeed specific to each subclass, make fly & quack separate *classes* (with code inheritance/reuse) ~~rather than separate interfaces (no reuse)~~ ← have those classes implement the separate interfaces (otherwise no reuse)



← could be superclass OR interface

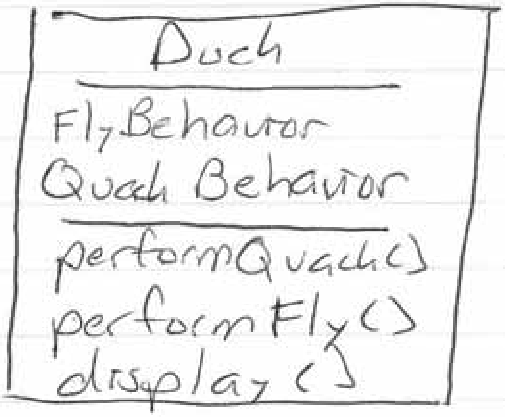
← do nothing

Tuesday 9/23/14 4156

in a language with multiple inheritance, could have each duck subclass be a subclass also of appropriate flying & quacking superclasses

but if only single inheritance, like Java, need another way

implement as new instance variables that point to a separate object of the flying or quacking behavior subtype → delegation



← behavior variables that can be changed at runtime

```
performQuack()
    quackBehavior.quack()

performFly()
    flyBehavior.fly()
```

polymorphism

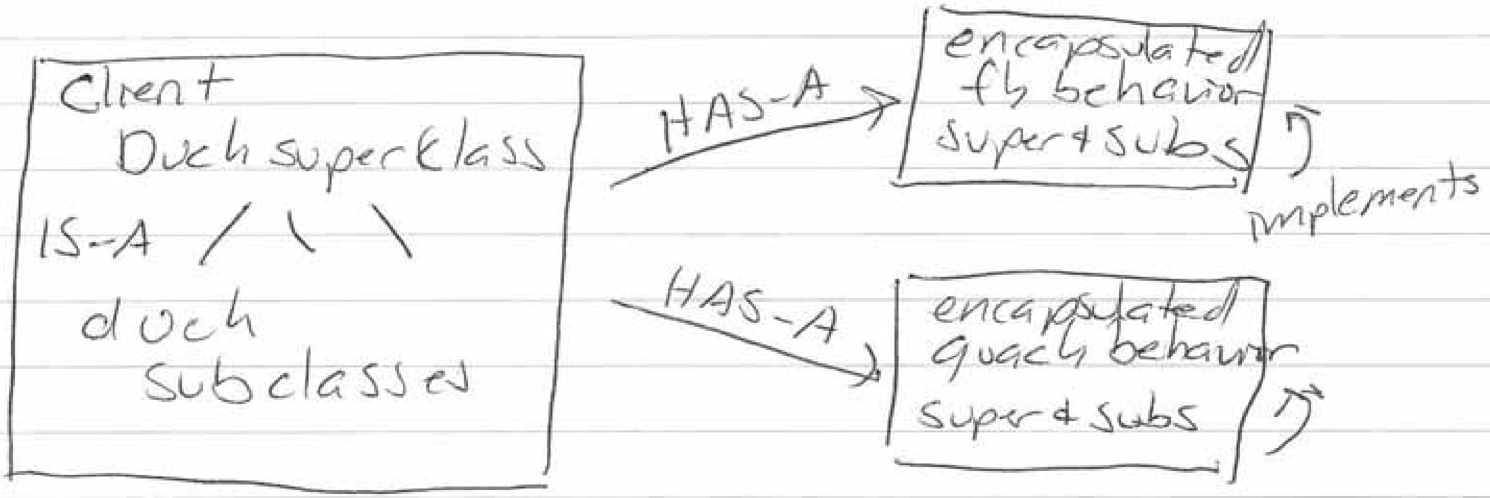
Tuesday 9/23/14 4156

modify constructor for each duck subclass to initialize ~~set~~ these behavior instance variables to an object of the appropriate flyable & quackable subtype

```
public class MallardDuck ~~E~~extends Duck {

        public MallardDuck() {
            quackBehavior = new Quack();
            flyBehavior = new FlyWithWings();
        }

        =

}
```

↑ can change at runtime with new (getter &) setter methods

encapsulated behavior
can add new types of flying & quacking with new flyable & quackable classes



Client
Duck superclass
IS-A
duck subclasses
HAS-A
HAS-A
encapsulated fly behavior super & subs
implements
encapsulated quack behavior super & subs

Tuesday 9/23/14 4156

behavior super/subclass hierarchy can be thought of as a family of algorithms

HAS-A (not IS-A or implements) relationship between class hierarchies
→ composition not inheritance & delegation
→ more flexible - can change behavior at runtime

strategy design pattern

defines a family of algorithms, encapsulates each one, & makes them interchangeable.
strategy lets the algorithm vary independently from the clients that use it

(need not be ducks!!)

design patterns provide shared vocabulary among developers

but not shared code, higher level than libraries & frameworks