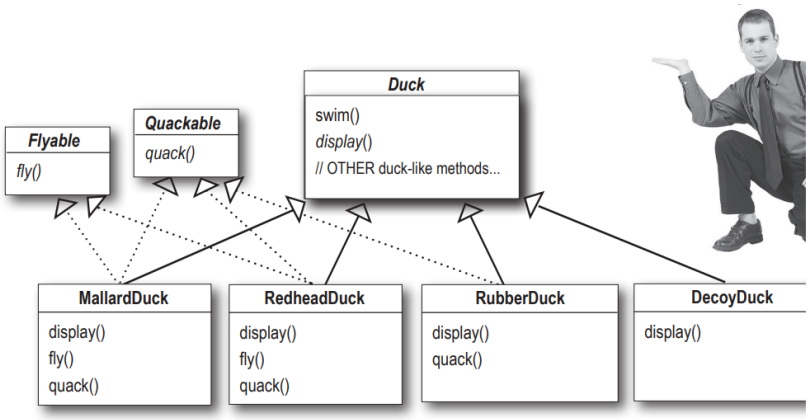
1.Question:

Which of the following are disadvantages of using inheritance to provide Duck behavior? (Choose all that apply.) (PAGE 5)

A. Code is duplicated across subclasses.
B. Runtime behavior changes are difficult.
C. We can't make ducks dance.
D. Hard to gain knowledge of all duck behaviors.
E. Ducks can't fly and quack at the same time.
F. Changes can unintentionally affect other ducks.

2.Question



What do you think about this design?

This design makes sure that the ducks which are supposed to quack and fly will fly. This approach makes it actually easier to understand because, if you are a duck which can fly then you will have the fly method and vice versa.

Question 3:
Lots of things can drive change. List some reasons you've had to change code in your applications (we put in a couple of our own to get you started)

I have created the 23 design pattern code. I needed to keep adjusting the classes, so they could run properly on the main application. Like recently I have created the duck code and then there was an issue with the static. So my solution was to make the classes static and it worked

Question 4:
Using our new design, what would you do if you needed to add rocket-powered flying to the SimUDuck app?

If I were to implement a rocket-powered duck then I could just create a class FlyRocketPowered which would implement the FlyBehaviour interface. The interface has a fly method.
I would change the implementation in that class to, instead of "I can fly" to "I'm flying with a rocket"
Then I would use that class then any duck where I would implement it would fly rocket-powered.

Question #5"
Can you think of a class that might want to use the Quack behavior that isn't a duck?

The class which would want to use the quack behavior could be the bath duck. They don't quack they will squeak.

CHAPTER 2

Based on our first implementation, which of the following apply? (Choose all that apply.) (PAGE 42)

A. We are coding to concrete implementations, not interfaces.
B. For every new display element we need to alter code.
C. We have no way to add (or remove) display elements at run time.
D. The display elements don't implement a common interface.
E. We haven't encapsulated the part that changes.
F. We are violating encapsulation of the WeatherData class

The observer pattern is like

Publishers + Subscribers = Observer Pattern

The observer pattern is like when the state of the objects changes all the dependents are notified.

Q: What does this have to do with one-to-many relationships? (PAGE 52)
In the Observer pattern, the subject is the object that contains the state and controls it.
There are many observers that rely on the Subject to tell them when its state changes.
This is what they mean with one-to-many relationships = ONE Subject to the MANY Observers.

Q: How does dependence come into this? (PAGE 52)
Because the subject is the sole owner of the data the observers are dependent on the subject to update them when the data changes.

The power of loose Coupling:

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.
The observer pattern provides an object design where subjects and observers are loosely coupled.

We can add new observers at any time. We don't need the subjects to add new types of observers, also we can re-use subjects or observers independently of each other.

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same. (Page75)

A: In the observer pattern what varies is the state of the Subject and the number of the Observer. I can use this pattern and decide which objects are dependent on that state without having to change that subject.

Program to an interface, not an implementation. (Page75)

A: Both the Subject and the Observer Implement an interface. The Objects implement the interface. The observer gets notified by the Subjects interface.