Design Patterns

Conexión

¿?

Concepto

Entender el Problema

- Cada grupo debe leer y analizar los casos prácticos
- Responder las preguntas
- 15 Min

Encontrar el Patrón para el problema

- Cada grupo es dueño de una solución, se puede quedar con ella o intercambiarla con la solución de otro grupo. El objetivo es que al final se quede con la solución correcta.
- 20 Min

Concreción

Preparar Presentación

- Cada grupo debe preparar una presentación
  - Requerimiento a implementar.
  - Qué problemas encontraron en el diseño inicial.
  - Qué patrón utilizaron, explicación del patrón y porqué.
- 20 Min

Presentación y Diagrama de Clases

- Un grupo presenta el patrón.
- Duración: 10 Min
- Todos los grupos crean el diagrama de clases final para el problema.
- Duración: 10 Min

Match the Pattern (5 Min)

Categorias: Buscar las categorías (5 Min)

Strategy vs State: Explicar la diferencia (5 Min)

Conclusión

¿?

Duck Simulator

La Solución

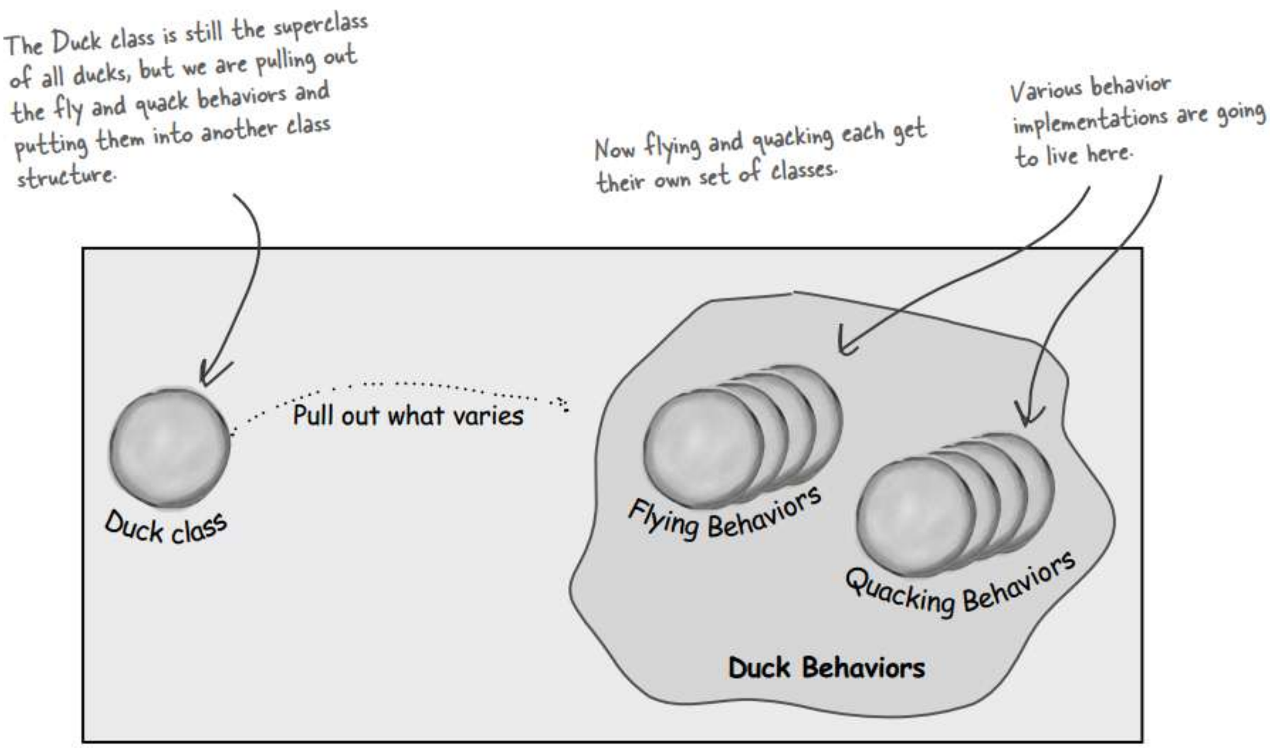
Sabemos que la herencia no está funcionando ya que los comportamientos de los patos se mantienen en cambiando y no es apropiado que todas las subclases tengan esos comportamientos.

Design Principle "Encapsulate what varies": Identificar los aspectos de la aplicación que cambian con nuevos requerimientos y encapsularlos, de la manera que luego podamos alterar o extender estar partes sin afectar el resto del código que se mantiene constante.

¿Qué partes de la aplicación varían o cambian frecuentemente?

Los métodos fly() y quack() de la clase Duck varían entre patos.

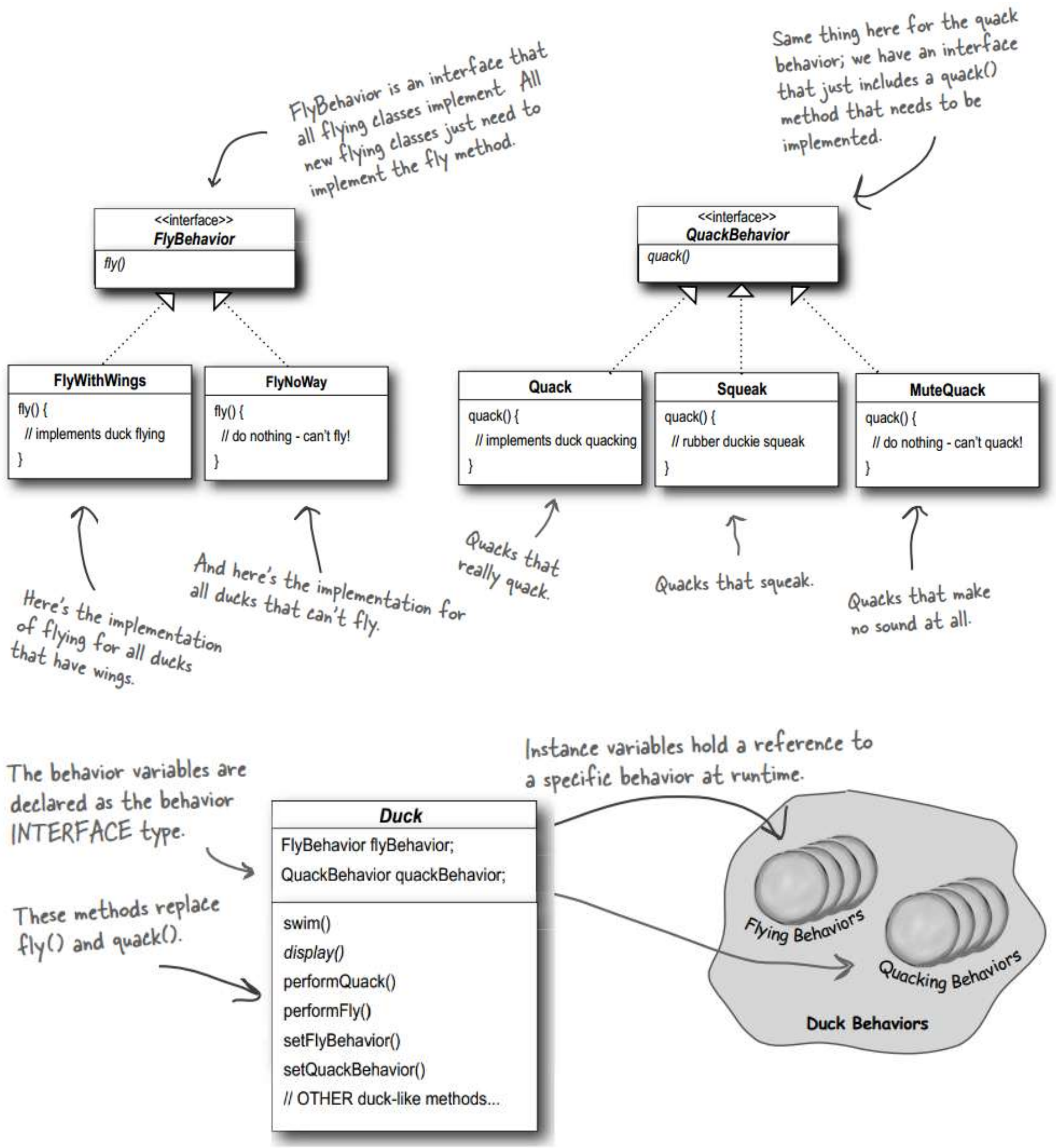
Sacamos los comportamientos "fly" y "quack" fuera de la clase Duck y creamos un nuevo conjunto de clases para representar cada uno de estos comportamientos.



Dibuja un diagrama de clases que represente el nuevo diseño de la aplicación. Debe cumplir lo siguiente:

- Utilizar una interfaz para representar cada comportamiento (FlyBehaviour, QuackBehaviour), cada implementación de un comportamiento debe implementar alguna de estar interfaces.
- Por el momento existen 2 comportamientos "fly" y 3 comportamientos "quack".
- Poder asignar los comportamientos a las subclasses de Duck. Ejemplo, instanciar un MallardDuck e inicializarlo con un tipo específico de FlyBehaviour.
- Las subclases de Duck delegan sus comportamientos "fly" y "quack" a las instancias de FlyBehaviour y QuackBehaviour.
- Cambiar dinámicamente los comportamientos "fly" y "quack" en las subclases.

Class Diagram



Escribe la implementación de la clase MallardDuck.

```java
public class MallardDuck extends Duck {

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

   public void display() {
       System.out.println("I'm a real Mallard duck");
   }
}
```

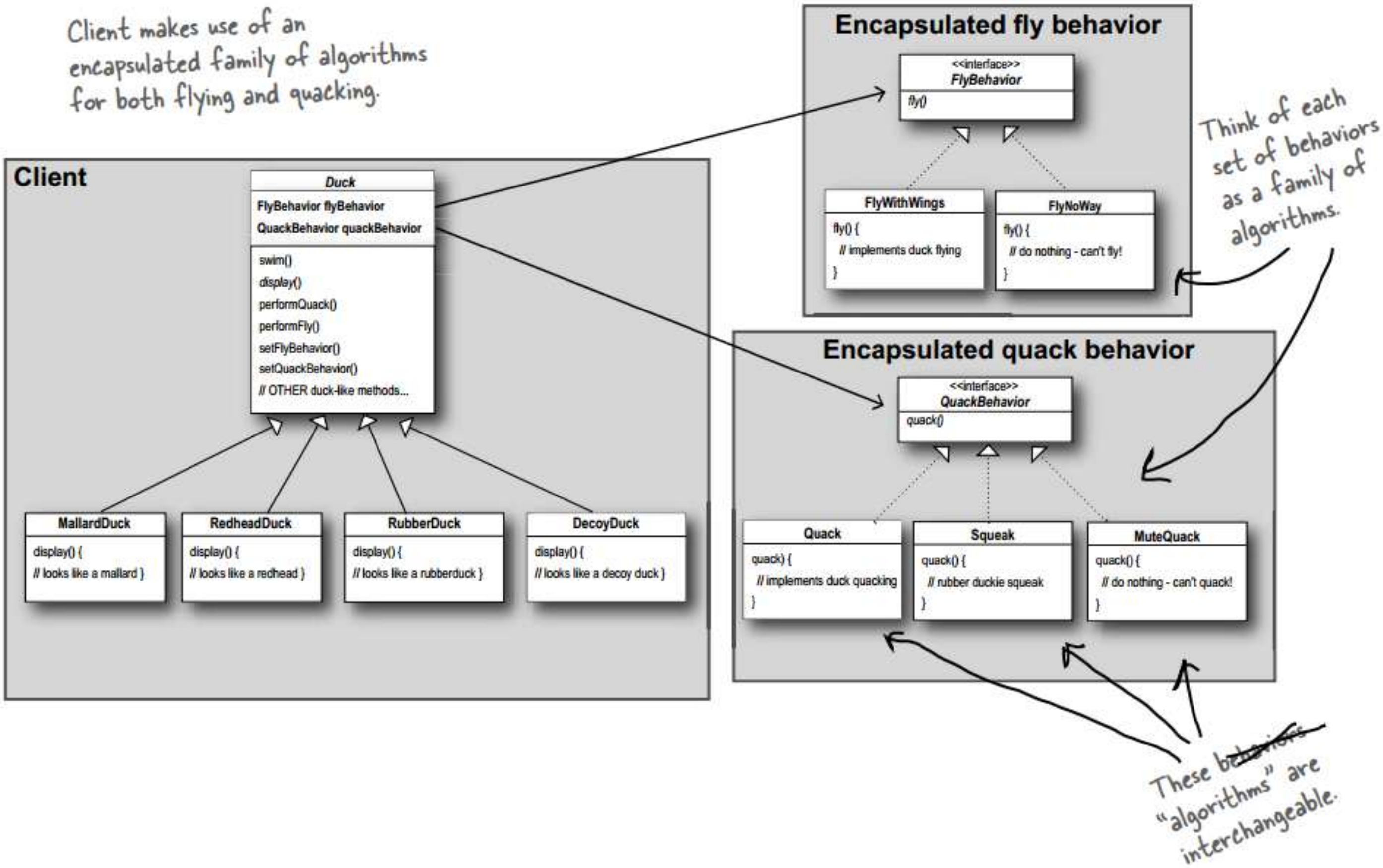
A MallardDuck uses the Quack class to handle its quack, so when performQuack is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quack-Behavior and flyBehavior instance variables from class Duck.

Qué beneficios tiene este nuevo diseño.

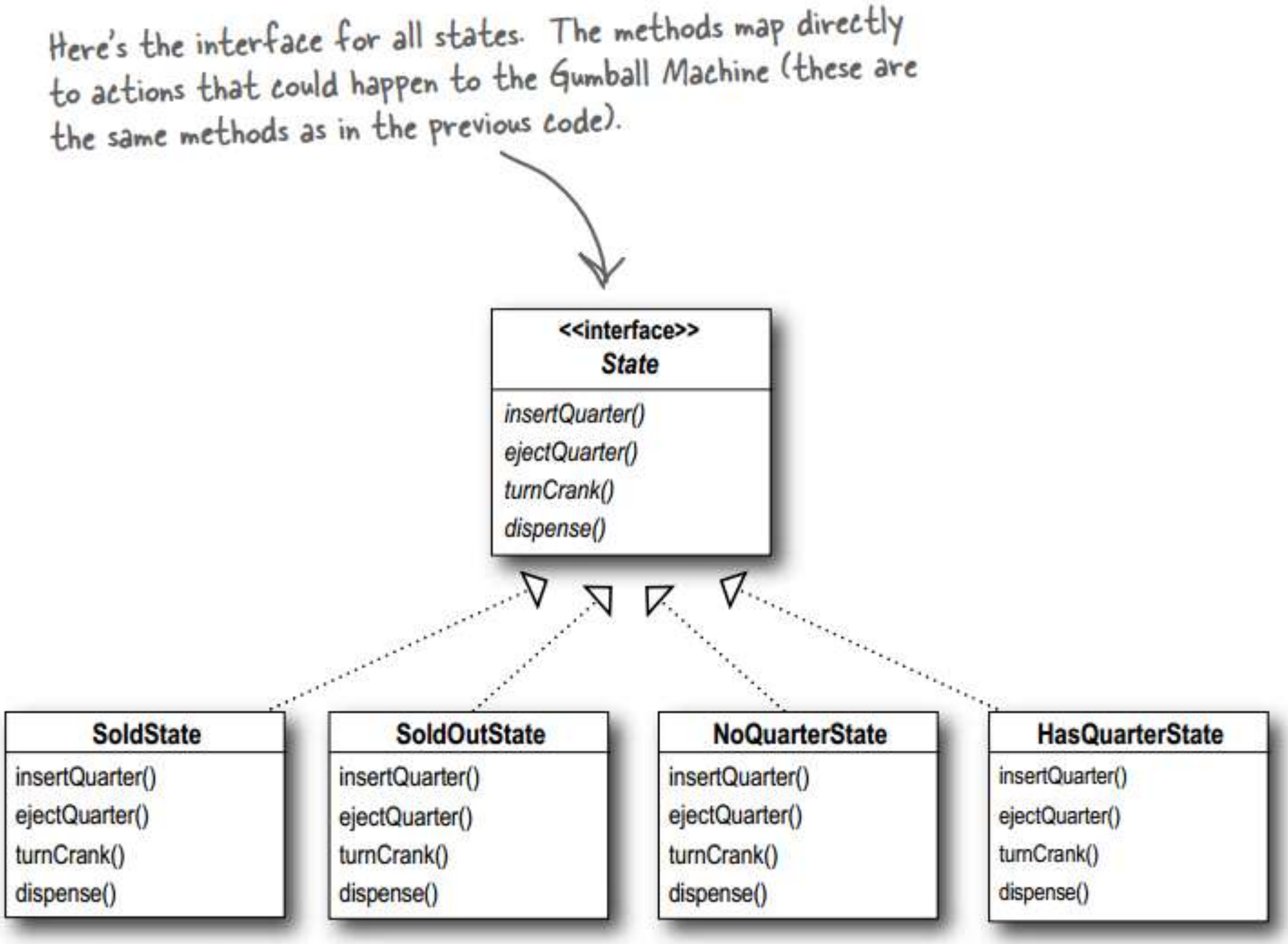
- Otros objetos pueden reutilizar los comportamientos “fly” y “quack”. Tenemos los beneficios de la reutilización sin los problemas de la herencia.
- Agregar nuevos comportamientos sin modificar o tocar ninguna de las clases ya existentes.
- Cambiar los comportamientos en tiempo de ejecución.



Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack) {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These ~~behaviors~~ "algorithms" are interchangeable.

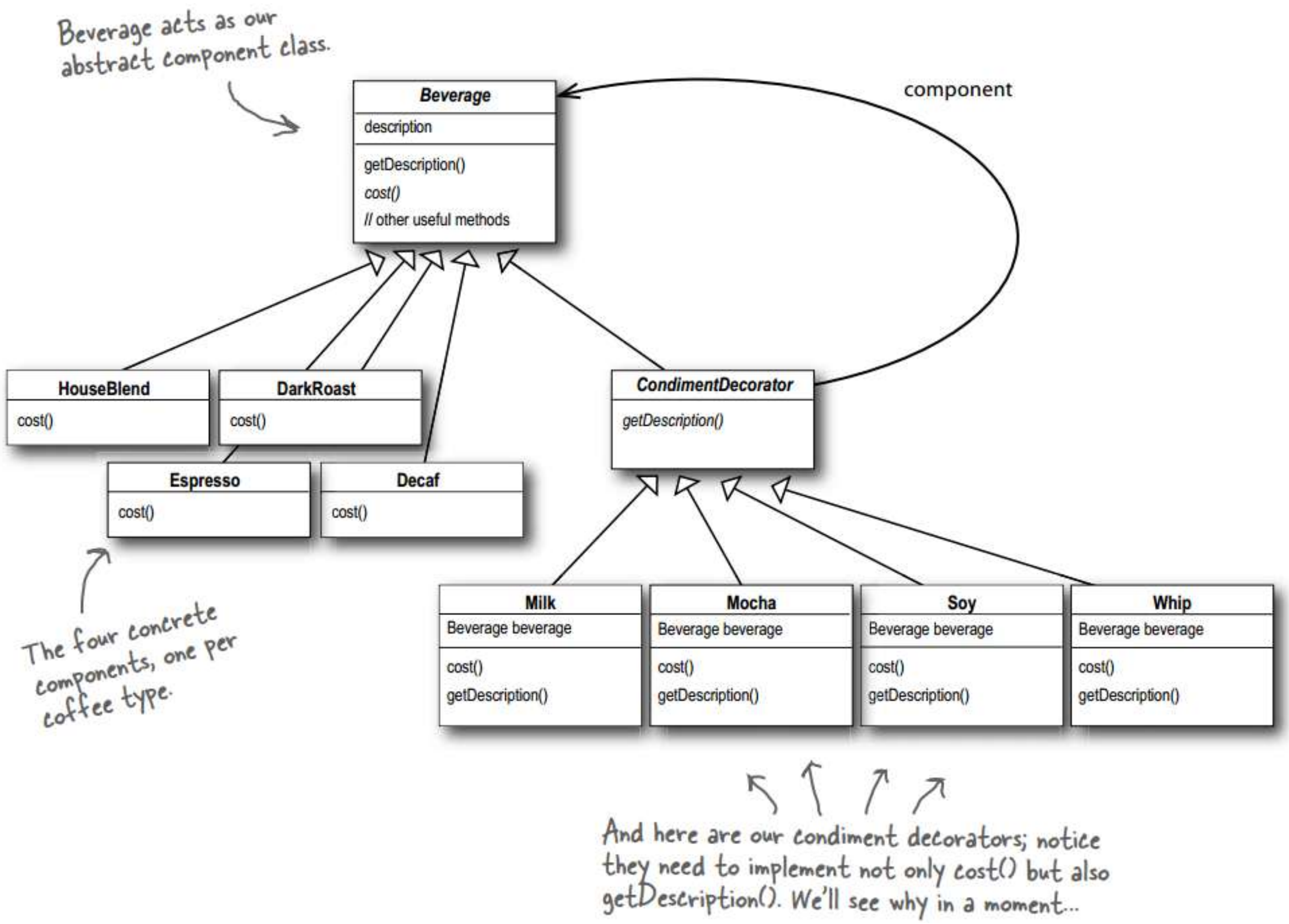
Gumball Machine

Class Diagram



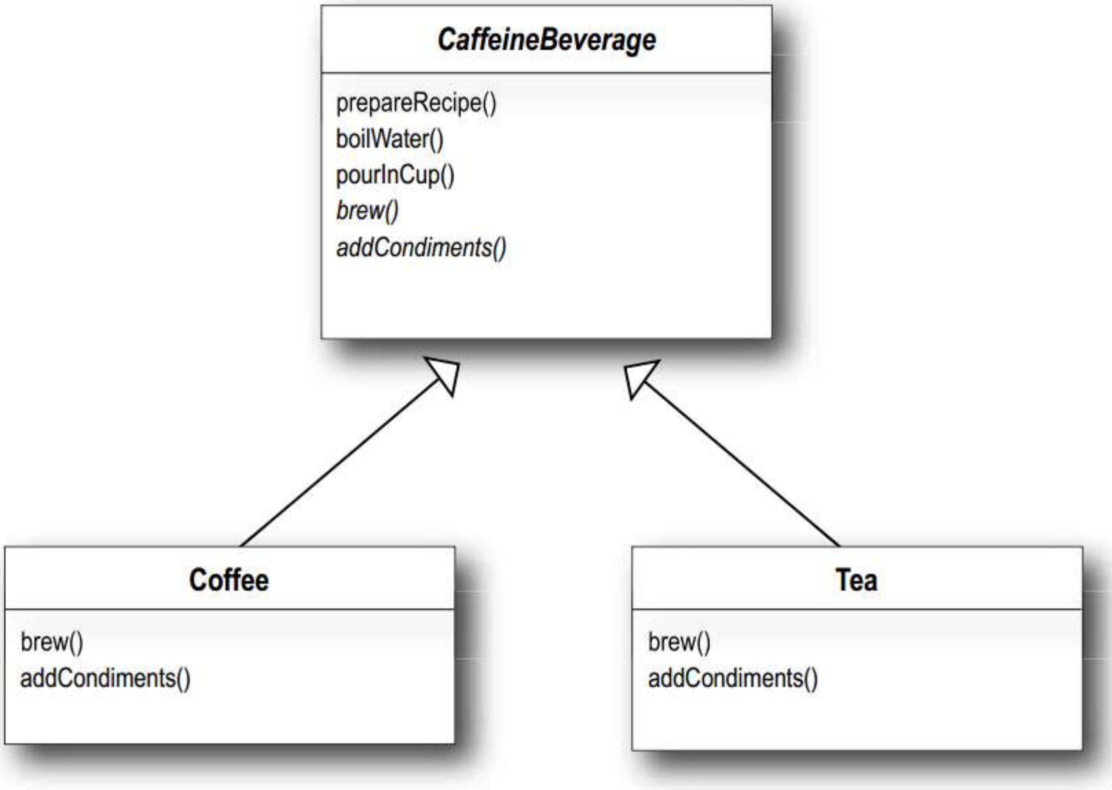
Starbuzz Coffee

Class Diagram



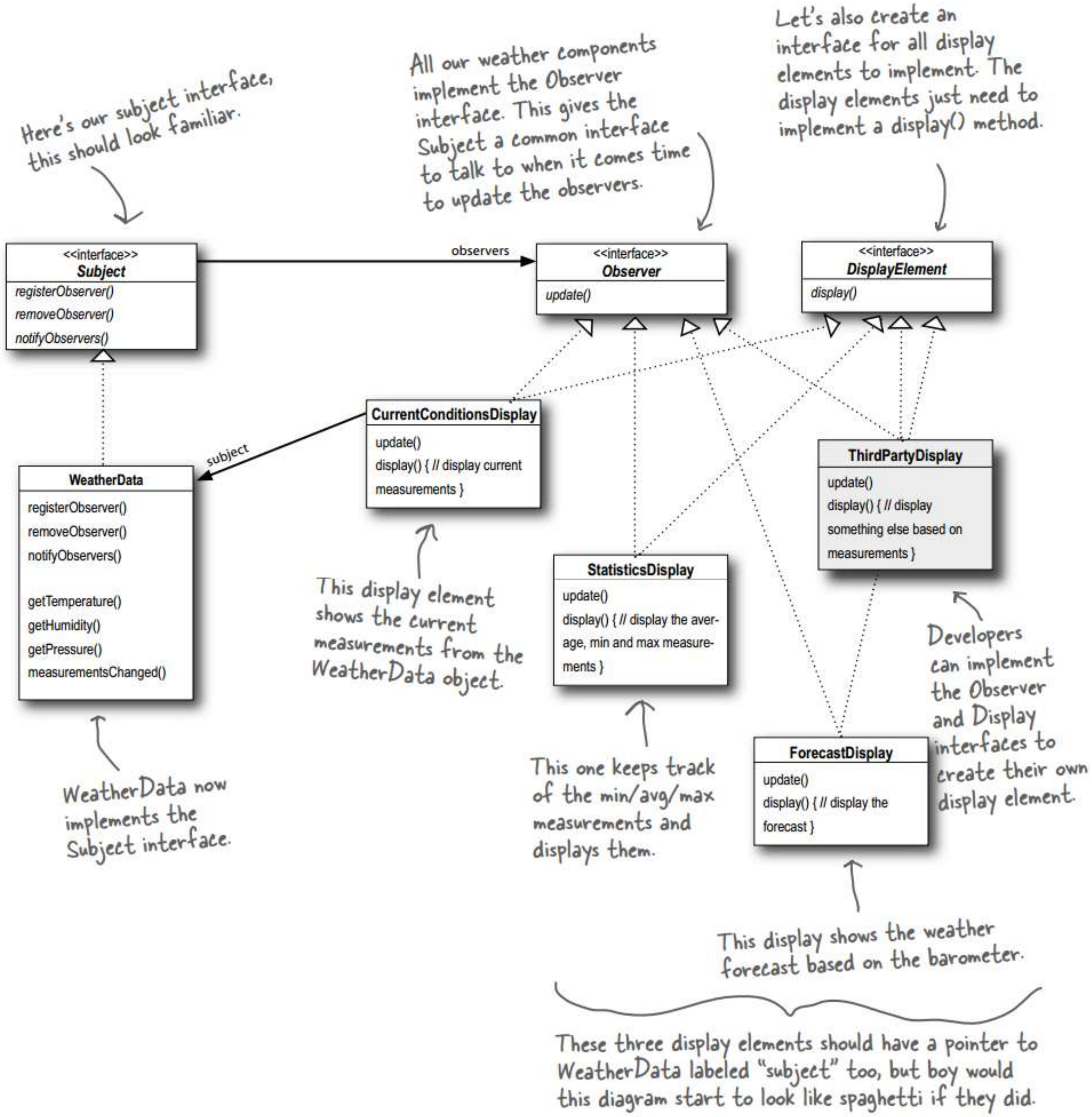
Starbuzz Coffee Recipes

Class Diagram



Internet-based Weather Monitoring Station

Class Diagram



State vs Strategy

With the State Pattern, we have a set of behaviors encapsulated in state objects; at any time the context is delegating to one of those states. Over time, the current state changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time as well. The client usually knows very little, if anything, about the state objects.

With Strategy, the client usually specifies the strategy object that the context is composed with. Now, while the pattern provides the flexibility to change the strategy object at runtime, often there is a strategy object that is most appropriate for a context object. For instance, in Chapter 1, some of our ducks were configured to fly with typical flying behavior (like mallard ducks), while others were configured with a fly behavior that kept them grounded (like rubber ducks and decoy ducks).

In general, think of the Strategy Pattern as a flexible alternative to subclassing; if you use inheritance to define the behavior of a class, then you're stuck with that behavior even if you need to change it. With Strategy you can change the behavior by composing with a different object.

Think of the State Pattern as an alternative to putting lots of conditionals in your context; by encapsulating the behaviors within state objects, you can simply change the state object in context to change its behavior.

Enlaza cada patrón con su definición

Pattern	Description
Decorator	Wraps an object and provides a different interface to it.
State	Subclasses decide how to implement steps in an algorithm.
Iterator	Subclasses decide which concrete classes to create.
Facade	Ensures one and only object is created.
Strategy	Encapsulates interchangeable behaviors and uses delegation to decide which one to use.
Proxy	Clients treat collections of objects and individual objects uniformly.
Factory Method	Encapsulates state-based behaviors and uses delegation to switch between behaviors.
Adapter	Provides a way to traverse a collection of objects without exposing its implementation.
Observer	Simplifies the interface of a set of classes.
Template Method	Wraps an object to provide new behavior.
Composite	Allows a client to create families of objects without specifying their concrete classes.
Singleton	Allows objects to be notified when state changes.
Abstract Factory	Wraps an object to control access to it.
Command	Encapsulates a request as an object.

Design Patterns Categories