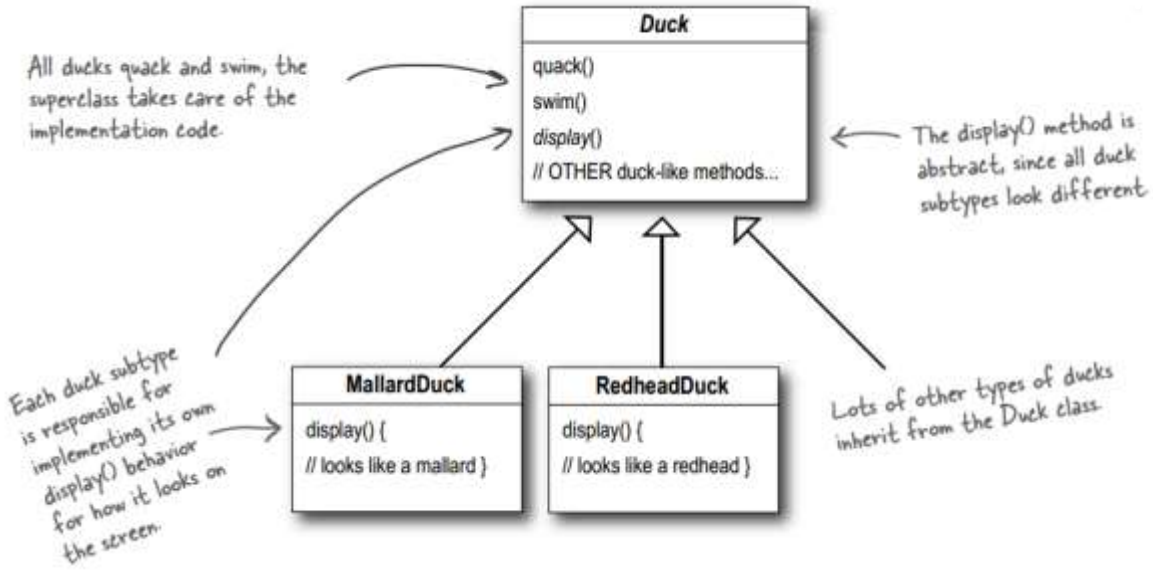
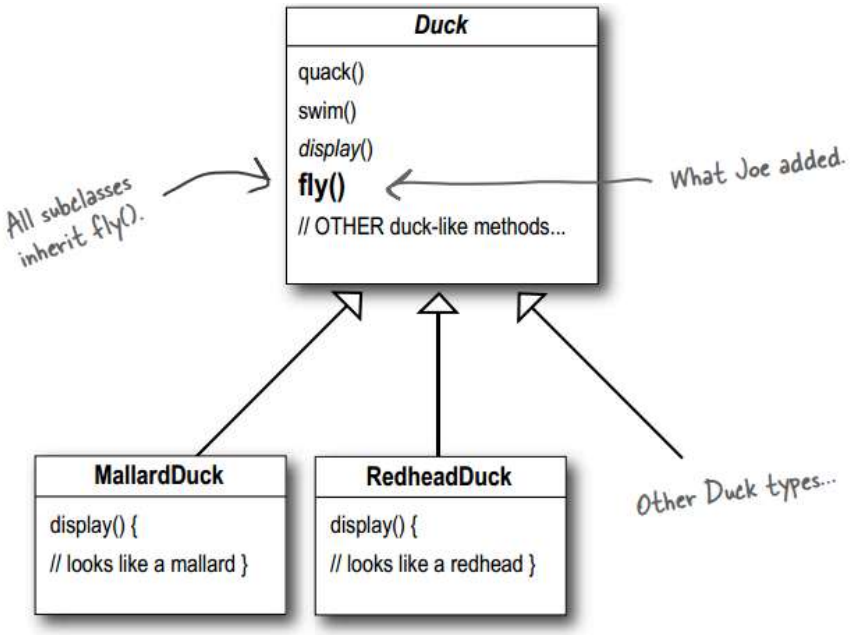
Case Studies

Duck Simulator

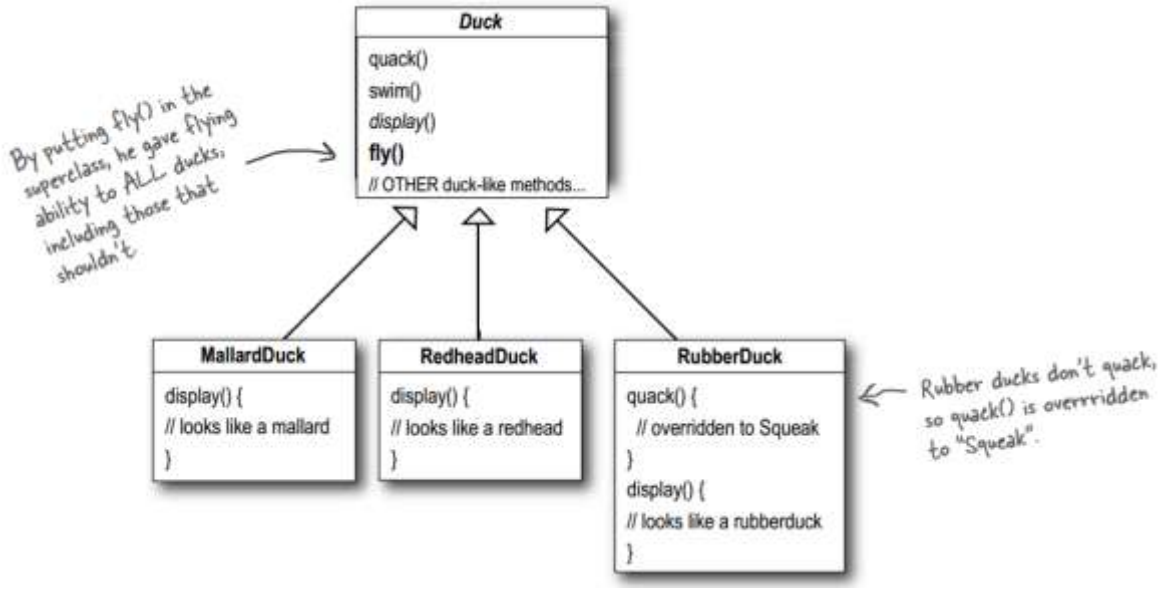
Joe para una compañía que ha creado un juego de simulación muy exitoso. Este juego muestra diversas especies de patos “nadando” y “graznando”. Los diseñadores iniciales del juego utilizaron técnicas OO estándar y crearon el siguiente diseño.



Debido a la competencia, la compañía cree que es tiempo de una gran innovación. Los ejecutivos decidieron que “patos voladores” es lo que necesitan para acabar con sus competidores. Joe piensa que es una solución muy sencilla y realiza el siguiente cambio en el diseño:

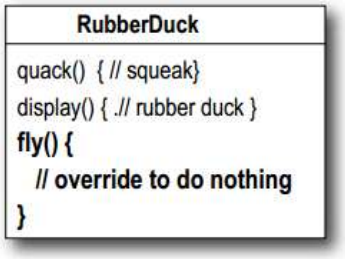


Pero ocurre un gran problema, en una revisión del producto con los stakeholders observaron que los patos de goma empezaron a volar por toda la pantalla. Joe no se dio cuenta que no todos los patos deben volar.

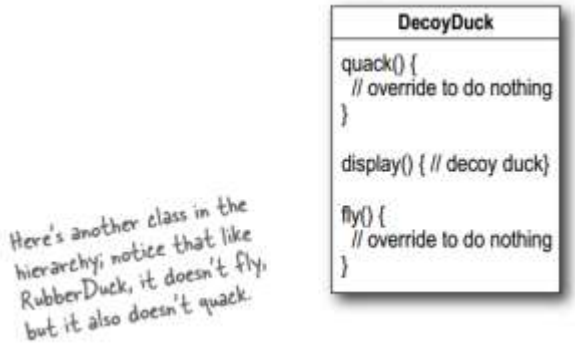


Lo que él pensaba que era un gran uso de la herencia para reutilizar comportamiento, no resultó tan bien a nivel de mantenimiento.

Joe piensa que podría solucionar el problema sobre-escribiendo métodos.



Pero que sucede cuando tiene que agregar un nuevo pato que no deba volar o hacer sonido….

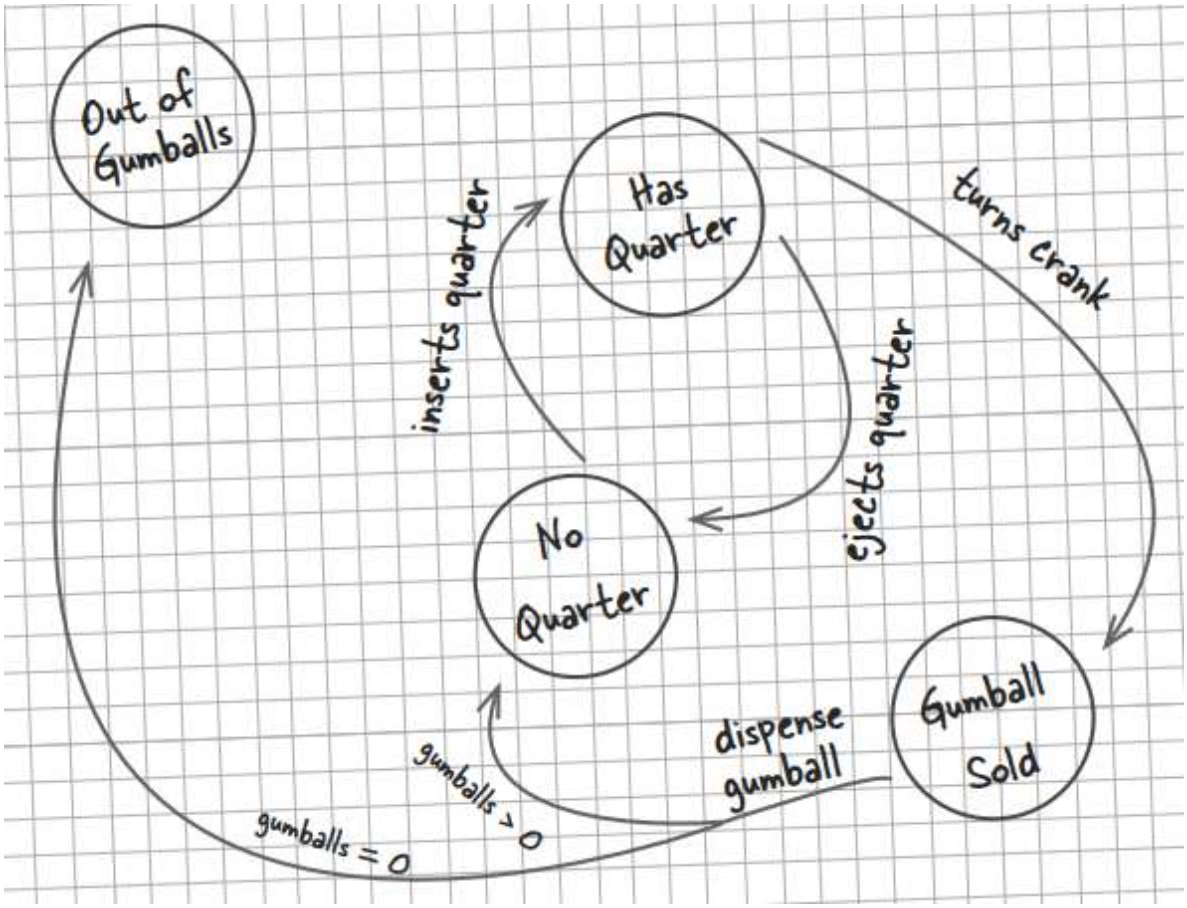


¿Cuáles son las siguientes desventajas del diseño anterior?

Gumball Machine

La empresa Mighty Gumball Inc. nos ha dado la responsabilidad de implementar el software para sus máquinas de goma de mascar.

Los especialistas de Mighty Gumball Inc. esperan que el controlador de la máquina de goma maneje la siguiente lógica. Ellos esperan agregar más comportamiento en el futuro, entonces se necesita mantener el diseño lo más flexible y mantenible posible.



La primera versión de la aplicación


```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }

    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println("Please wait, we're already giving you a gumball");
        }
    }
```

```
public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}
```

Now, if the customer tries to remove the quarter...

If there is a quarter, we return it and go back to the NO_QUARTER state.

Otherwise, if there isn't one we can't give it back.

You can't eject if the machine is sold out, it doesn't accept quarters!

If the customer just turned the crank, we can't give a refund; he already has the gumball!

The customer tries to turn the crank...

```
public void turnCrank() {
    if (state == SOLD) {
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    }
}
```

Someone's trying to cheat the machine.

We need a quarter first.

We can't deliver gumballs; there are none.

Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.

Called to dispense a gumball.

```
public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("No gumball dispensed");
    }
}

// other methods here like toString() and refill()
}
```

We're in the SOLD state; give 'em a gumball!

Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.

None of these should ever happen, but if they do, we give 'em an error, not a gumball.

Sabíamos que pasaría…un requerimiento de cambio

El CEO de Mighty Gumball, Inc. piensa que convertir "comprar un goma" en un juego incrementaría significativamente las ventas. Para esto, desea que agreguemos la siguiente lógica: "el 10% de las veces que se gira la manivela, el cliente recibe 2 gomas en vez de 1".

Analicemos cómo implementar este cambio…….

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;


public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}
```

First, you'd have to add a new WINNER state here. That isn't too bad...

... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

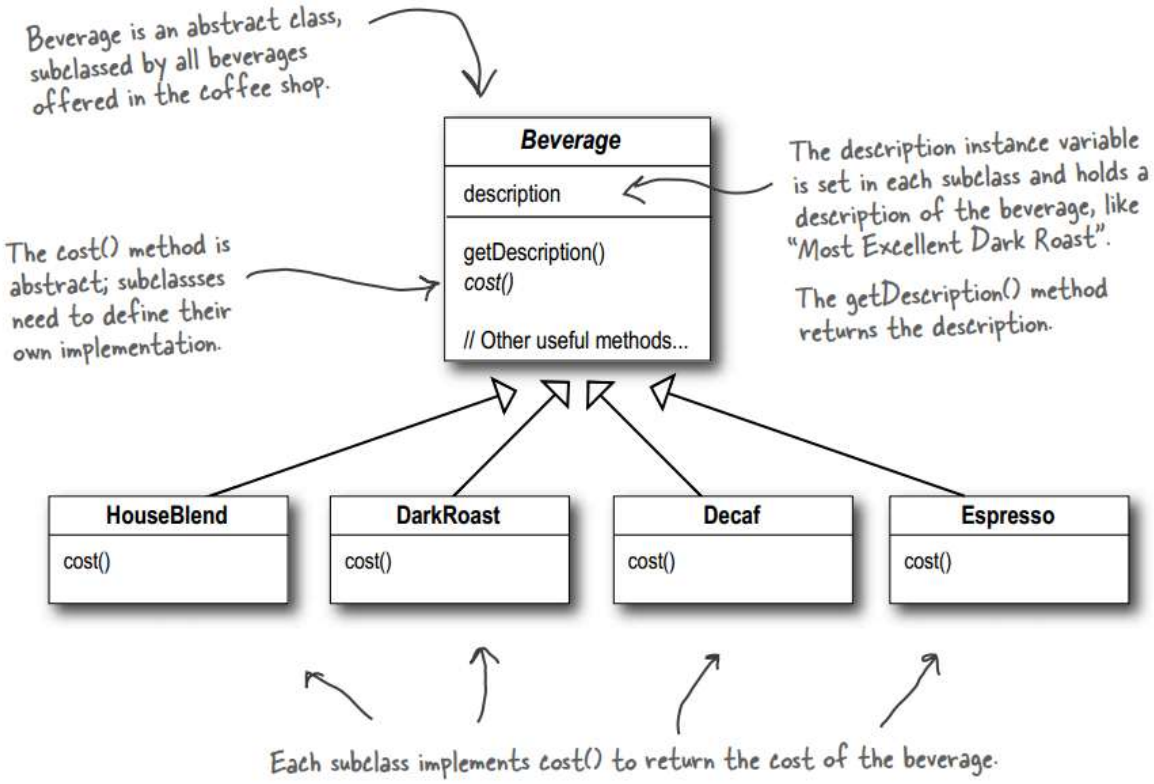
turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

¿Cuáles son los problemas de nuestra implementación?

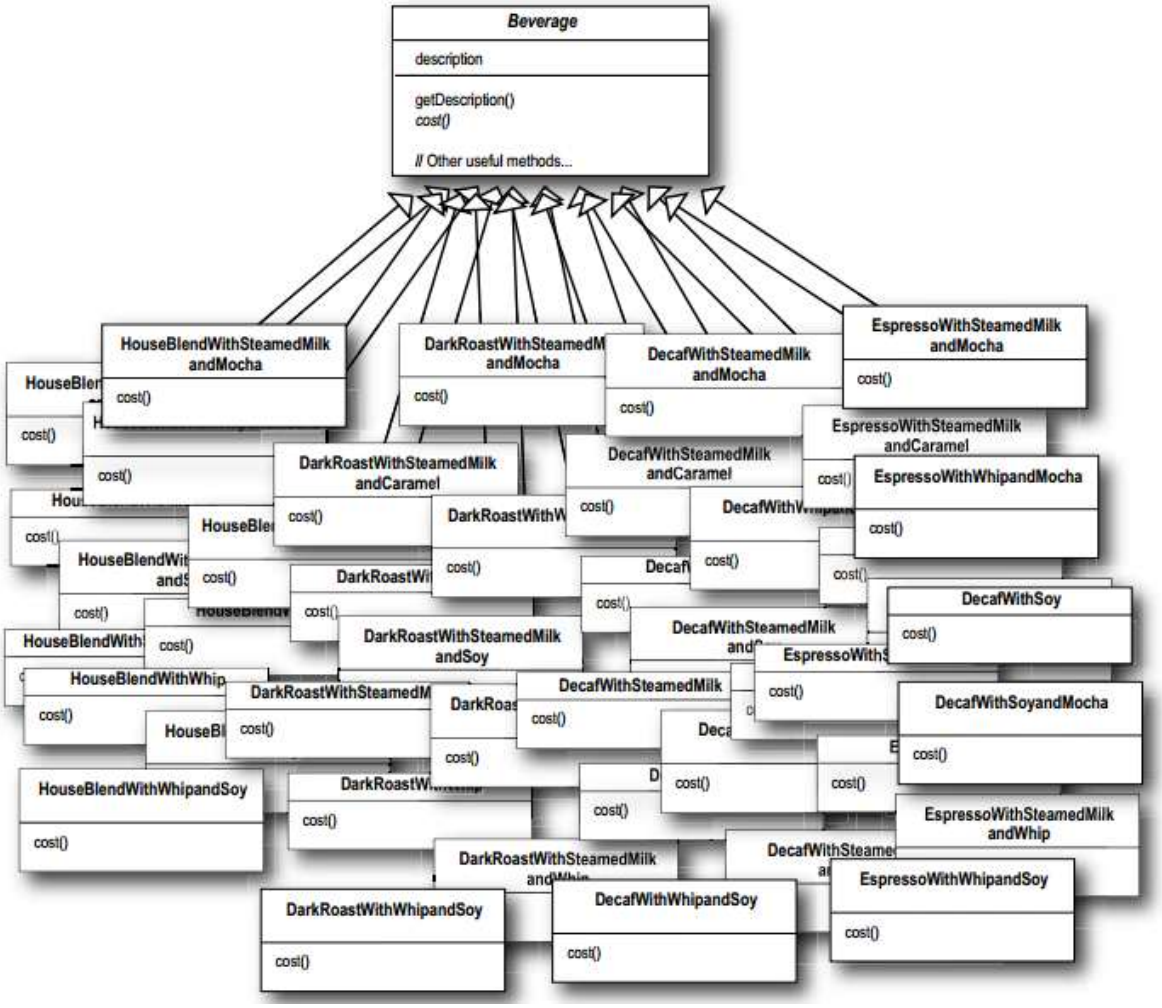
Starbuzz Coffee

Starbukzz Coffe es el coffee shop con mayor crecimiento del país, debido a esto tienen problemas actualizando su sistema de órdenes para que concuerde con todas las bebidas que ofrecen.

La aplicación actual está diseñada de la siguiente manera:



Adicionalmente al café, los clientes también pueden solicitar condimentos adicionales como: mocha, crema, leche al vapor, soya, etc. Starbuzz carga un costo adicional por cada uno de estos por lo tanto también deben considerarse dentro del sistema de órdenes.

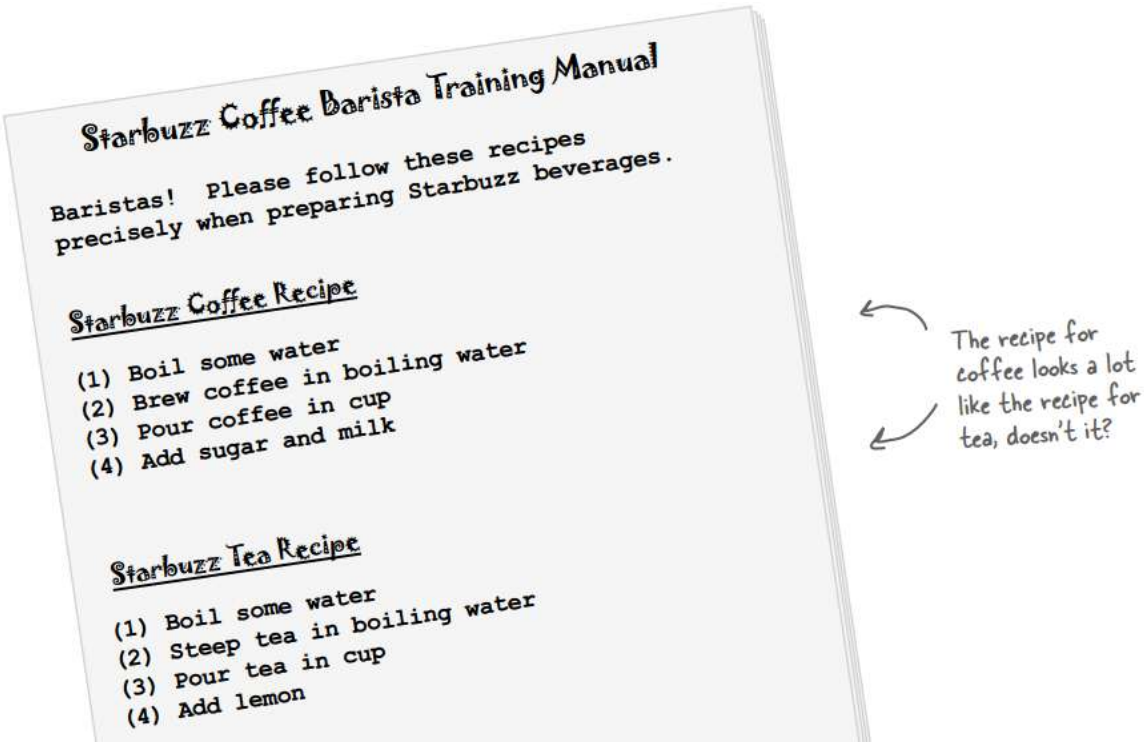


¿Cuáles son los problemas de la implementación anterior?

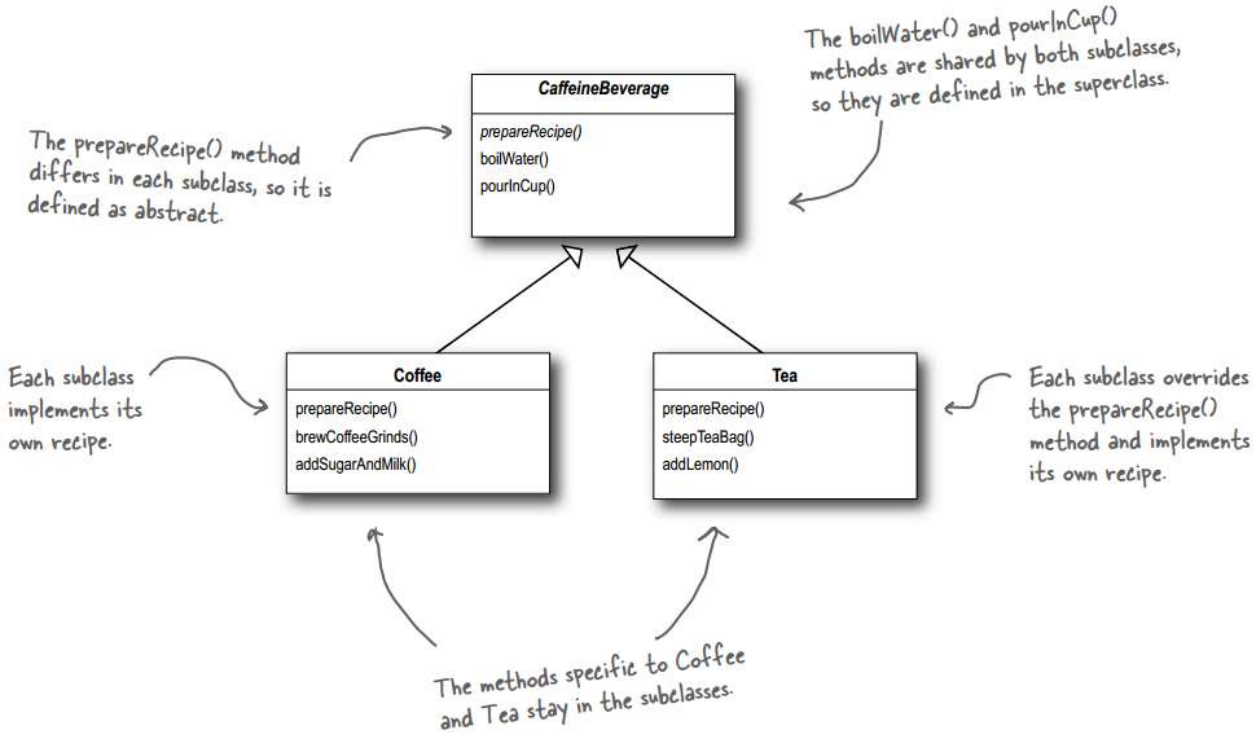
Starbuzz Coffee Recipes

Para poder atender a más clientes, Starbuzz Coffee piensa automatizar la preparación de todas sus bebidas. Tu equipo ha sido contratado para implementar la aplicación que realice esta tarea.

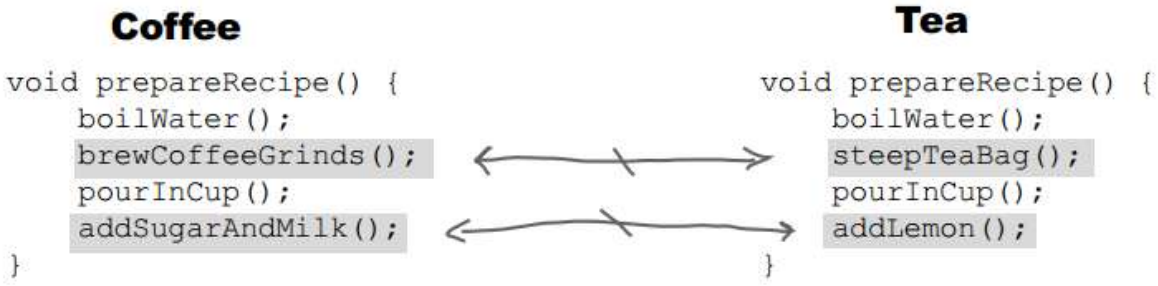
Se debe considerar las siguientes tareas que realizan los baristas de forma manual:



Otros miembros del equipo diseñaron el siguiente modelo:



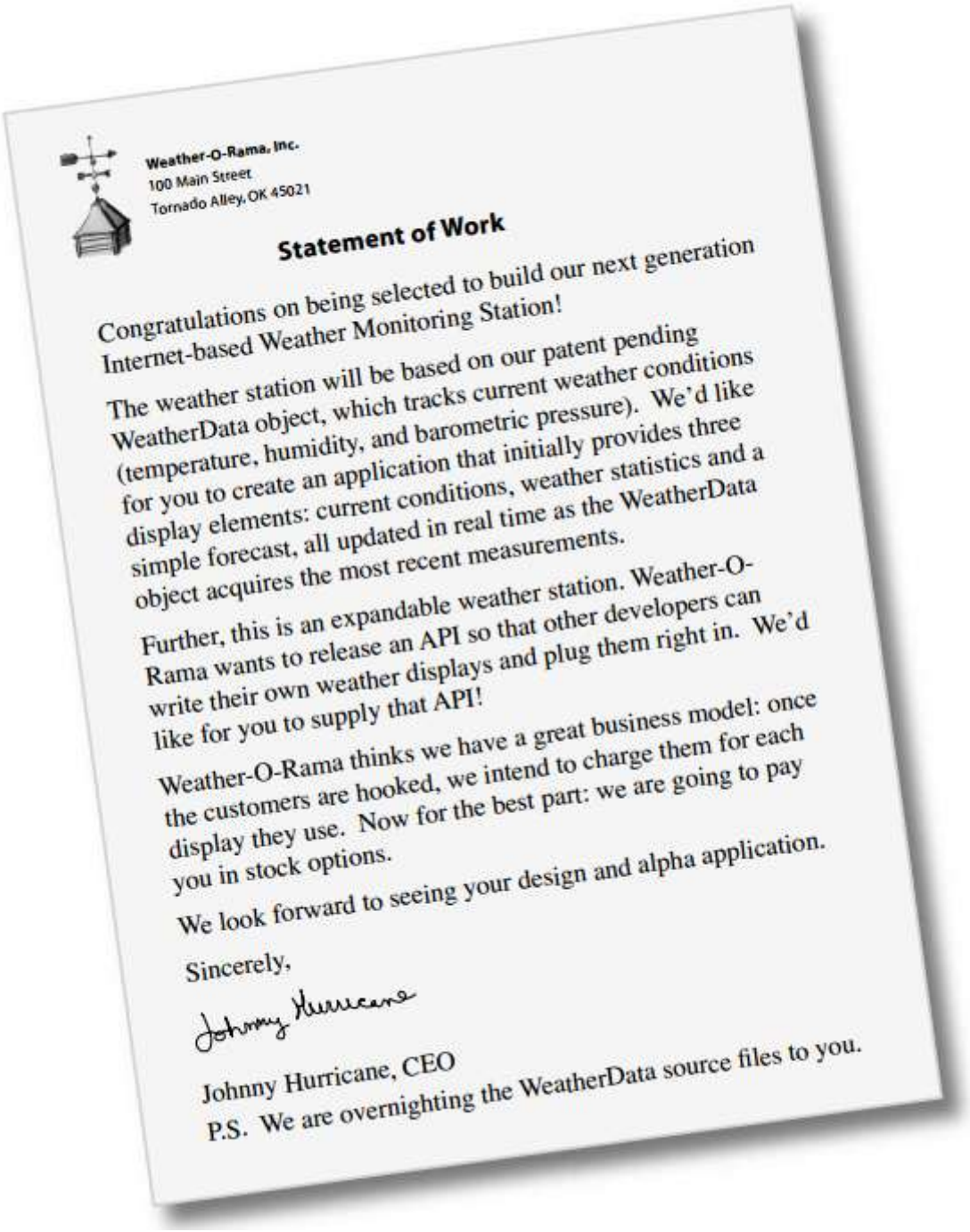
Examinando la lógica de la aplicación, te diste cuenta que existen ciertas similitudes en la preparación de ambas bebidas. Crees que el diseño actual es aceptable para una versión inicial pero que es necesario sacar una nueva versión considerando estas similitudes.



¿Cuáles son los problemas de la implementación anterior?

Internet-based Weather Monitoring Station

Tu equipo ha sido seleccionado para construir la siguiente generación del famoso “Internet-based Weather Monitoring Station” para la empresa “Weather-O-Rama, Inc.”.



Weather-O-Rama, Inc.
100 Main Street
Tornado Alley, OK 45021

Statement of Work

Congratulations on being selected to build our next generation Internet-based Weather Monitoring Station!

The weather station will be based on our patent pending WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We'd like for you to create an application that initially provides three display elements: current conditions, weather statistics and a simple forecast, all updated in real time as the WeatherData object acquires the most recent measurements.

Further, this is an expandable weather station. Weather-O-Rama wants to release an API so that other developers can write their own weather displays and plug them right in. We'd like for you to supply that API!

Weather-O-Rama thinks we have a great business model: once the customers are hooked, we intend to charge them for each display they use. Now for the best part: we are going to pay you in stock options.

We look forward to seeing your design and alpha application.

Sincerely,

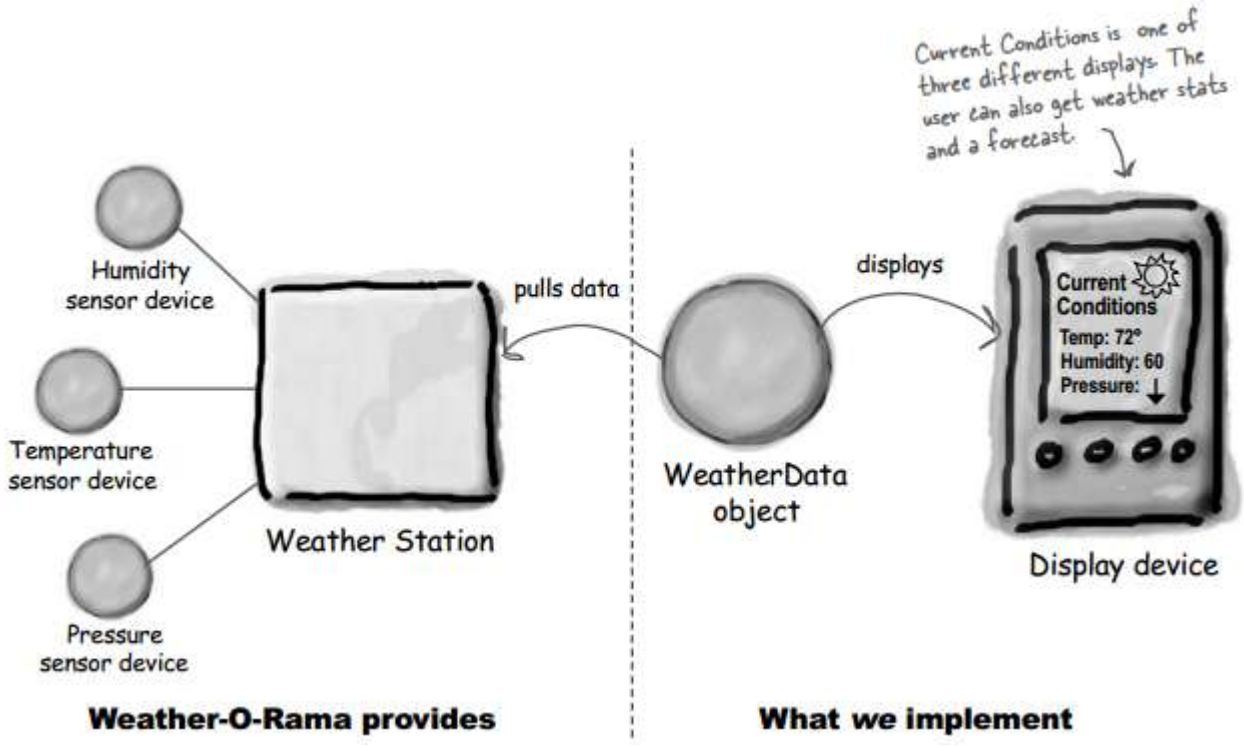
Johnny Hurricane

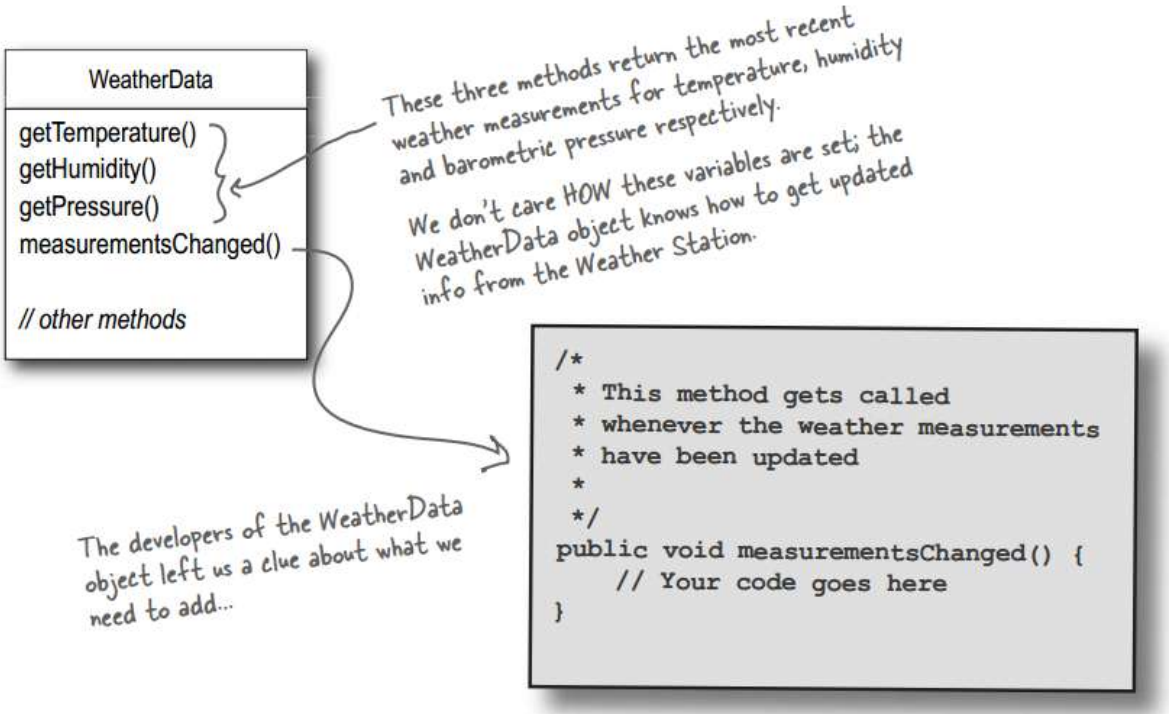
Johnny Hurricane, CEO

P.S. We are overnighting the WeatherData source files to you.

Arquitectura

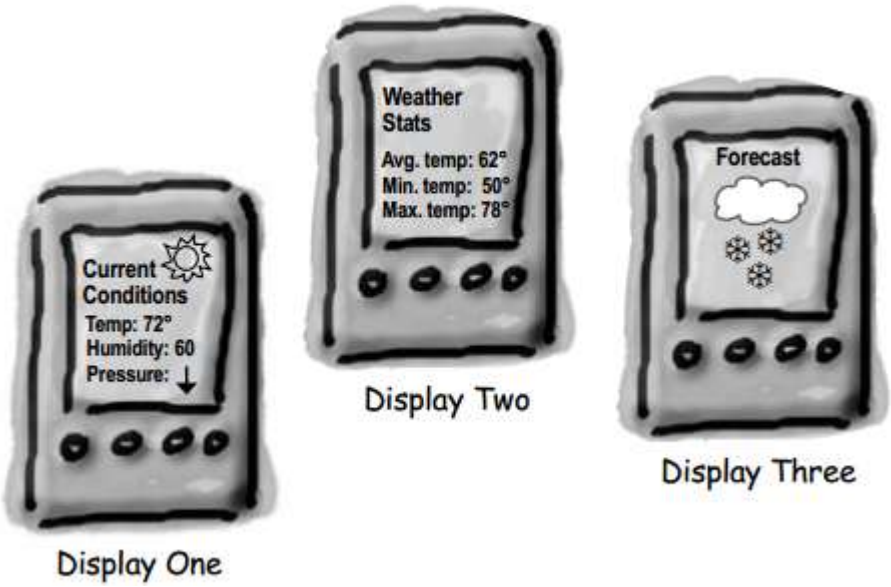
- Weather station: Dispositivo Físico que obtiene los datos del clima.
- WeatherData Objetct: Rastrea los datos provenientes del Weather Station y actualiza las pantallas.
- Displays: Pantallas que muestran a los usuarios las condiciones actuales del clima.





Nuestro Trabajo

Nuestro trabajo es crear una aplicación que use el WeatherData Object y actualice 3 diferentes tipos de pantallas (Current conditions, Weather Stats, Forecast).



El sistema debe ser fácilmente expandible: en el futuro, otros desarrolladores podrán crear nuevas pantallas y agregarlas fácilmente a la aplicación; los usuarios podrán agregar o remover de la aplicación, tantas pantallas como deseen.

El Diseño Inicial

Esta es nuestra primera implementación, hemos tomado la idea de los desarrolladores de “Weather-O-Rama” y agregamos nuestro código en el método measurementsChanged():

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

¿Cuáles son los problemas de esta primera implementación?