Qué Me Pongo

Alternativas de diseño a la cuarta Iteración

Franco Bulgarelli
Versión 1.1
Mayo 2021

¡Llegamos a la cuarta iteración! En esta etapa se incorpora el concepto de clima, que va a hacer que todo sea mucho más interesante (y un poco más complejo). ¿Cómo podemos resolver los nuevos requerimiento? ¡Acompáñenos!

Primer punto

> *Como usuarie de QuéMePongo, quiero poder conocer las condiciones climáticas de Buenos Aires en un momento dado para obtener sugerencias acordes.*

Pensemos una primera colaboración:

```
AccuWeatherAPI apiClima = new AccuWeatherAPI();
apiClima.getWeather("Buenos Aires, Argentina");
estadoDelTiempo.get(0).get("Temperature");
```

Listo, ¿no? El código que acompaña el enunciado nos habla de un componente externo y nos da suficientes indicios de cómo usarlo. Si ya alguien nos dejó una solución al problema, aprovechémosla sin más, ¿no?

Segundo punto

> *Como stakeholder de QuéMePongo, quiero poder asegurar la calidad de mi aplicación sin incurrir en costos innecesarios.*

Pero no todo es felicidad en el bello mundo de AccuWeather y los componentes externos: según la documentación nos *cobra 0,05 USD por cada vez que la llamamos a partir del décimo llamado diario*. ¡Esto sí es un problema!

Entonces quizás nos convenga *envolver* a `AccuWeatherAPI` para que sea un poco más inteligente y cachee[1]. Por ejemplo podríamos hacer que los resultados vivan en caché durante dos horas y media, así aseguramos un máximo de 10 pedidos al día por cada ciudad.

Podríamos usar a este *envoltorio* así:

```
// van a ser objetos propios
ServicioMeteorologico servicioMeteorologico = new ServicioMeteorologico();
estadoDelTiempo = servicioMeteorologico
                      .obtenerCondicionesClimaticas("Buenos Aires, Argentina");
estadoDelTiempo.get(0).get("Temperature");
```

¿Y cómo lo implementamos?

```
public class ServicioMeteorologico {
  private Map<String, RespuestaAccuWeather> ultimasRespuestas;

  // ¿es el API un componente pesado? ¿es un componente que podemos compartir?
  // si instanciamos el objeto acá mismo, estamos asumiendo un montón de cosas
  // (que es liviano, que no parece ser compartible),
  // pero al usar inyección de dependencias evitamos
  // hacer esas suposiciones: "pateamos el problema para adelante"
  public ServicioMeteorologico(AccuWeatherAPI api, Duration periodoDeValidez) {
     this.api = api;
     this.periodoDeValidez = periodoDeValidez;
     this.ultimasRespuestas = new HashMap<String, Object>();
   }

  // esto es un algoritmo clásico de caché:
  //     * si no está en caché o está expirada, recalcular y guardar.
  //     * en cualquier caso: devolver el resultado de la caché
  public Map<String, Object> obtenerCondicionesClimaticas(String direccion) {
    if (
         !this.ultimasRespuestas.contains(direccion)
         || this.ultimasRespuestas.get(direccion).expiro()) {
      ultimasRespuestas.put(
            new RespuestaAccuWeather(this.consultarApi(direccion), proximaExpiracion()));
    }
    return this.ultimasRespuestas.get(direccion).getEstadoDelTiempo();
  }
```

[1] Cachear, o más específicamente, memorizar (**memoize**), es el acto de almacenar el resultado de una operación costa (en términos de tiempo, capacidad de cómputo, dinero u otras variables) para que más tarde podamos reutilizarlo sin pagar dicho costo, aún a expensas de recibir respuestas desactualizadas.

```
  private LocalDateTime proximaExpiracion() {
    return LocalDateTime.now().plus(this.periodoDeValidez);
  }

  // optimización: acá podríamos guardar directamente las próxima predicciones y
  // actualizar las cacheadas si las nuevas son diferentes
  private Map<String, Object> consultarApi(String direccion) {
    return this.api.getWeather(direccion).get(0);
  }
}


// Esto es necesario solamente para "agregarle" a las condiciones
// climáticas una expiración
public class RespuestaAccuWeather {
  public RespuestaAccuWeather(Map<String, Object> estadoDelTiempo, DateTime expiracion) {
    // ...etc...
  }

  // getters

  // si no fuera por este método en algunos lenguajes
  // bien podría haber sido una tupla
  public boolean expiro() {
    return this.expiracion.isAfter(DateTime.now);
  }
}
```

Listo, ¿no?

Alternativa minimalista

Otra alternativa minimalista consistirá en tener un servicio meteorológico por ciudad, y guardar el resultado de la última operación en una variable de instancia:

```
public class ServicioMeteorologico {
  private Map<String, Object> ultimaRespuesta;
  private LocalDateTime proximaExpiracion;

public ServicioMeteorologico(AccuWeatherAPI api, Duration periodoDeValidez, String
direccion) {
    this.api = api;
    this.expiracion = expiracion;
    this.direccion = direccion;
  }

  public Map<String, Object> obtenerCondicionesClimaticas() {
    if (this.ultimaRespuesta == null || this.expiro()) {
      this.ultimaRespuesta = consultarApi();
```

```
      this.proximaExpiracio = LocalDateTime.now().plus(this.expiracion);
    }
    return this.ultimaRespuesta;
  }

  private boolean expiro() {
    return proximaExpiracion.isAfter(DateTime.now);
  }
}
```

Esta alternativa tiene la ventaja de ser más sencilla que la anterior, y posibilitar tener diferentes servicios meteorológicos por ciudad (incluso podríamos asociar una ciudad a su servicio meteorológico), a expensas de tener que crear más objetos. Como el enunciado no dice nada sobre si esto es posible o deseable, cualquiera de las dos alternativas son igualmente válidas.

Tercer Punto

> *Como usuarie de QuéMePongo, quiero que al generar una sugerencia las prendas sean acordes a la temperatura actual sabiendo que para cada prenda habrá una temperatura hasta la cual es adecuada. (Ej.: "Remera de mangas largas" no es apta a más de 20°C)*

Acá es donde la cosa se pone interesante, porque tenemos que relacionar nuestro componente de clima con el resto del sistema. En otras palabras, ¡llegó el momento de usarlo!

Para resolver este punto, asumiremos que el `Guardarropas` ya es capaz de calcular la combinatoria de prendas y producir todos los atuendos posibles (sin considerar si son útiles o no). También asumiremos que el existe un objeto *seleccionador de atuendos*, que llamaremos `AsesorDeImagen`, que es responsable justamente de elegir entre los atuendos aquellos que son aptos para el clima (y quizás, nuestras preferencias de moda en el futuro)

```
public class AsesorDeImagen {
  private ServicioMeteorologico servicioMeteorologico;

  // .... constructor que inyecta al servicio meteorologico....

  public Atuendo sugerirAtuendo(String direccion, Guardarropas guardarrpas) {
    Map estadoDelTiempo = this.servicioMeteorologico()
                              .obtenerCondicionesClimaticas(direccion);

    int temperatura = (int) prediccion.get(¿¿??);
    Humedad humedad = (double) prediccion.get(¿¿???) > 0.8 ? LLUVIOSO : SECO;
    // El enunciado no lo pide, pero a modo de ejemplo
    // de cómo fácilmente se podría complicar más
```

```
    List<Atuendo> combinaciones = guardarropas.todasLasPosiblesCombinaciones()

      return combinaciones
            .filter( combinacion -> combinacion.aptaParaTemperatura(temperatura) )
            .filter( combinacion -> combinacion.aptaParaHumedad(humedad) )
            .first();
  }
}
```

Además necesitaremos modificar las clases Prenda y Atuendo:

```
public class Prenda {
  // ... agregamos temperaturaMaxima y un
  // método aptaParaTemperatura para comprobar si la prenda es apta a cierta
temperatura....

}

public class Atuendo {
    // ...agregamos aptaParaTemperatura, que verifica que todas
    // las prendas sean aptaParaTemperatura....
}
```

¡Complicado! Si bien este planteo funciona, de forma **implícita** estamos acoplando al AsesorDeImagen a AccuWeather. Volamos por un momento a esta parte:

```
int temperatura = (int) prediccion.get(¿¿??);
// Momento, ¿estará en farenheit? ¿Se llamará Temperatura, temperature, temp? ¿es int o
long?

Humedad humedad = (double) prediccion.get(¿¿???) > 0.8 ? LLUVIOSO : SECO;
// ¿A partir de qué punto lo consideramos húmedo? ¿No podría ya el API resolverme eso?
```

Como tenemos que saber como se llaman sus keys y tipos de datos siempre que lo usemos, y convertirlos a lo que necesitamos.

Entonces una posibilidad es justamente llevar el problema del manejo de claves, de las conversiones de tipos de datos y otras reglas asociadas a la respuesta de AccuWeather a... ¡el componente que lidia con AccuWeather!

```
public class ServicioMeteorologico  {
  private Map<String, RespuestaMeteorologica> ultimasRespuestas;
      public ServicioMeteorologico(AccuWeatherAPI api, Duration periodoDeValidez) {
    this.api = api;
    this.periodoDeValidez = periodoDeValidez;
    this.ultimasRespuestas = new HashMap<String, EstadoDelTiempo>();
  }
```

```
  // notar que cambió el retorno del método
  public EstadoDelTiempo obtenerCondicionesClimaticas(String direccion) {
    // ... idem versión anterior....
  }

  // ... idem....

  private EstadoDelTiempo consultarApi(String direccion) {
    // Encapsulamos el problema
    // ¿Podríamos haber usado algún patron creacional?
    // Quizás, pero no sentí que aportara nada acá.
    Map<String, Object> respuesta = this.api.getWeather(direccion).get(0);
    int temperatura = (int) respuesta.get(...);
    Humedad humedad = (double) respuesta.get(...) > 0.8 ? LLUVIOSO : SECO;
    return new EstadoDelTiempo(temperatura, humedad);
  }
}

// ya que estaos, renombramos esta clase
public class RespuestaMeteorologica {
  public RespuestaMeteorologica(EstadoDelTiempo estadoDelTiempo, DateTime expiracion) {
    // ...etc...
  }

  // ...el resto queda igual...
}
```

Ahora podemos usar a este componente de una forma mucho más cómoda y menos (implícitamente) acoplada:

```
public class AsesorDeImagen {
  private ServicioMeteorologico servicioMeteorologico;

  public Atuendo sugerirAtuendo(String direccion, Guardarropas guardarrpas) {
    EstadoDelTiempo estadoDelTiempo = this.servicioMeteorologico()
                                          .obtenerCondicionesClimaticas(direccion);
    List<Atuendo> combinaciones = guardarropas.todasLasPosiblesCombinaciones()
      return combinaciones
        .filter(combinacion ->
combinacion.aptaParaTemperatura(estadoDelTiempo.temperatura))
        .filter(combinacion -> combinacion.aptaParaHumedad(estadoDelTiempo.humedad))
        .first();

  }
}
```

Pero pero, ¿ya llegamos al adapter? No. ¿Ya llegamos al adapter? No. Pero definitivamente hasta acá hemos hecho un gran trabajo de adaptar las interfaces externas a nuestro sistema.

Cuarto Punto

Como administraderes de QuéMePongo, quiero poder configurar fácilmente diferentes servicios de obtención del clima para ajustarme a las cambiantes condiciones económicas.

¡A generalizar más aún! Recién es acá donde entra el *adapter.* Para eso vamos a hacer un refactor:

1. renombrar `ServicioMeteorologico` -> `ServicioMeteorologicoAccuWeather`
2. introducir la interface `ServicioMeteorologico`, que tiene la misma semántica e interfaz del viejo `ServicioMeteorologico`

Otro refactor que no hubiera estado conceptualmente bien hubiera sido el siguiente:

1. dejar ServicioMeteorologico como está
2. introducir la interface IServicioMeteorologico, que tiene la misma semántica e interfaz del viejo ServicioMeteorologico

De esta manera, nos queda:

```
public interface ServicioMeteorologico {
   EstadoDelTiempo obtenerCondicionesClimaticas(String direccion);
}
```

O en una alternativa minimalista:

```
public interface ServicioMeteorologico {
   int consultarTemperatura(String direccion);
   // opcional
  // Humedad consultarHumedad(String direccion);
}
```

Con estos cambios, la implementación del ahora devenido `ServicioMeteorologicoAccuWeather` queda así:

```
public class ServicioMeteorologicoAccuWeather implements ServicioMeteorologico {
  // .... idem punto anterior....

  private Map<String, RespuestaMeteorologica> ultimasRespuestas;
      public ServicioMeteorologico(AccuWeatherAPI api, Duration periodoDeValidez) { //
    this.api = api;
    this.periodoDeValidez = periodoDeValidez;
    this.ultimasRespuestas = new HashMap<String, Object>();
```

```
  }

  public EstadoDelTiempo obtenerCondicionesClimaticas(String direccion) {
    if (!this.ultimasRespuestas.contains(direccion) ||
this.ultimasRespuestas.get(direccion).expiro()) {
      ultimasRespuestas.put(new RespuestaMeteorologica(consultarApi(direccion),
proximaExpiracion()));
    }
    return this.ultimasRespuestas.get(direccion).getEstadoDelTiempo();
  }

  // etc...
}
```

Algunas aclaraciones importantes:

1. Las interfaces externas no implementan la interfaz del adapter:
2. Aunque NO hubiéramos implementado el caché, igual esta clase hubiera sido necesaria

Diagrama de clases final