PRUEBA TÉCNICA PARA AXPE CONSULTING - ANTONIO F GASCO

Ejercicio 1

Tras analizar el pseudocódigo, he podido apreciar varios errores tanto de diseño como errores en runtime y en compilación. A continuación, los expongo y explico cada uno de los errores:

ERRORES DE COMPILACIÓN

1.-

```
    this.services.forEach(service, index => {...});
```

Esta sintaxis es incorrecta, debido a que forEach acepta una callback function como primer parámetro, y en cambio, se le está pasando la instancia del servicio como primer parámetro, y una callback function como segundo parámetro. Esta segunda callback function tiene index como su único parámetro.

SOLUCION:

```
    this.services.forEach((service, index) => {...});
```

Aplicando este cambio, ahora sí se le está pasando la callback function como primer parámetro de forEach correctamente.

Por otra parte, index es un parámetro que no se está usando en el pseudocódigo original, por lo que habría que borrarlo, ya que un código limpio no debería estar declarando o importando variables que no se usan. Por tanto quedaría así:

```
    this.services.forEach(service => {...});
```

ERRORES EN RUNTIME

1.-

```
    if (typeof service == StreamingService) {...}
    // AND
     else if (typeof service == DownloadService) {...}
    // AND
    if (typeof multimediaContent == PremiumContent) {...}
```

Estos checks nunca darán el resultado deseado ya que, en Javascript, cuando chequeamos usando typeof <instancia de clase> siempre obtendremos “object” como resultado del

check. En Javascript, no existen las clases como tal, sino que son funciones de tipo constructor “maquilladas” para parecer una clase.Y sus instancias siempre son de tipo “object” ya que todo en Javascript es de tipo “object”, excepto los tipos primitivos, es decir “string”, “number”, “boolean”, etc.

Una solución sería añadir el atributo ‘type’ a Service y guardar los tipos de todos los servicios en constantes:

```
const STREAMING_SERVICE_TYPE = 'streaming';
const DOWNLOAD_SERVICE_TYPE = 'download';
```

Entonces, simplemente a la hora de chequear qué servicio ha usado el usuario, hacemos lo siguiente:

```
      if (service.type === STREAMING_SERVICE_TYPE) {
        total += multimediaContent.streamingPrice;
      } else if (service.type === DOWNLOAD_SERVICE_TYPE) {
        total += multimediaContent.downloadPrice;
      }
```

ERRORES DE DISEÑO

Esta sección es donde he encontrado el mayor número de errores o problemas críticos a la hora de escalar la app. La implementación actual viola varios de los principios SOLID que fundamentan la base para diseñar software escalable, flexible y mantenible.

1.- S**O**LID -> **OPEN/CLOSED PRINCIPLE**

```
    this.services.forEach((service, index) => {
      let multimediaContent = service.getMultimediaContent();

      if (typeof service == StreamingService) {
        total += multimediaContent.streamingPrice;
      } else if (typeof service == DownloadService) {
        total += multimediaContent.downloadPrice;
      }

      if (typeof multimediaContent == PremiumContent) {
        total += multimediaContent.additionalFee;
      }
    });
```

En este código de getTotal, se chequean instancias concretas de clases a la hora de calcular el total. Esto crea un problema a la hora de añadir nueva funcionalidad o nuevos servicios en el futuro. Ya que OPEN/CLOSED PRINCIPLE indica que una clase debe estar **abierta** para extenderse pero **cerrada** para modificarse. Por tanto, cuando se añada un nuevo servicio en el futuro, tendremos que tocar (modificar) la clase RegisteredUser, lo cual no tiene ningún sentido, ya que ¿por qué deberíamos tocar la clase que gestiona los usuarios registrados cuando estamos añadiendo un nuevo servicio de multimedia cuando son dos secciones de la app que no tienen nada que ver?

Por tanto, una solución óptima sería delegar la lógica de saber qué tipo de servicio está usando el usuario a otra clase, y no incluir esta lógica en la clase de RegisteredUser, la cual no debería ser consciente de cómo se calcula el precio, sino simplemente del precio a pagar. El método getTotal quedaría así:

```
getTotal() {
  let total = 0;

  this.services.forEach((service) => {
    total += service.getPrice();
  });

  return total;
}
```

2.- SOLI**D** -> **DEPENDENCY INVERSION**

```
  this.services.forEach((service, index) => {
    let multimediaContent = service.getMultimediaContent();

    if (typeof service == StreamingService) {
      total += multimediaContent.streamingPrice;
    } else if (typeof service == DownloadService) {
      total += multimediaContent.downloadPrice;
    }

    if (typeof multimediaContent == PremiumContent) {
      total += multimediaContent.additionalFee;
    }
  });
```

Esta pieza de código vuelve a ser la culpable de cometer otro error de diseño si nos basamos en las “best practices” y los principios SOLID. En este caso, se viola el principio de DEPENDENCY INVERSION, que indica que los módulos de alto nivel no deberían

depender de módulos de bajo nivel. Esto crea un problema de clases que están “tightly coupled” lo cual hace que el sistema sea menos flexible y más difícil de mantener.

En este caso, la clase RegisteredUser, un módulo de alto nivel, depende de implementaciones concretas de subclases de Service, además de PremiumContent, todas ellas clases o módulos de bajo nivel. RegisteredUser es un módulo de alto nivel ya que gestiona la lógica de negocio, en este caso calcular el total del coste a cobrar a un cierto usuario. El resto de módulos son de bajo nivel ya que consisten en gestionar operaciones específicas, detalles de implementación, etc.

Para prevenir esto, el principio de DEPENDENCY INVERSION recomienda el uso de clases abstractas (relevante en otros lenguajes como C#, el cual conozco ya que me encanta el gamedev en mi tiempo libre), interfaces, etc. y que RegisteredUser dependa de éstas, en vez de implementaciones concretas. Mi solución para prevenir este problema sería:

Implementar una superclase con getPrice como método “abstracto”. Por tanto, la lógica de getPrice la gestionará cada subclase, o cada implementación concreta de la clase Service

```
// Clase “abstracta” o interface
class Service {
  getMultimediaContent() {
     // Lógica para retornar el contenido multimedia asociado al servicio
  }

  // Este método es “abstracto”. A implementar en subclases
  getPrice() {}
}
```

```
class StreamingService extends Service {
  getPrice() {
    let content = this.getMultimediaContent();
    return content.getPrice('streaming');
  }
}
```

```
  getTotal() {
    let total = 0;

    this.services.forEach((service) => {
      total += service.getPrice();
    });

    return total;
  }
```

Como ven, ahora el módulo de alto nivel RegisteredUser depende de una clase “abstracta” o una interfaz, en vez de depender de módulos de alto nivel. A su vez, el módulo de bajo nivel StreamingService depende de una clase “abstracta” o interfaz, las cuales no existen específicamente en Javascript, pero implementando una solución parecida a la mía se puede resolver el problema mencionado anteriormente.

Además, pueden observar un spoiler del siguiente punto, que es la falta de polimorfismo y “encapsulation”, y vemos como el contenido multimedia es el responsable de calcular el precio. Tiene sentido ya que cada película o stream o serie tendrá un precio diferente, por lo tanto que esa lógica esté presente en la clase MultimediaContent es preferible.

3.- **S**OLID -> **SINGLE RESPONSIBILITY**

```
this.services.forEach((service, index) => {
  let multimediaContent = service.getMultimediaContent();

  if (typeof service == StreamingService) {
    total += multimediaContent.streamingPrice;
  } else if (typeof service == DownloadService) {
    total += multimediaContent.downloadPrice;
  }

  if (typeof multimediaContent == PremiumContent) {
    total += multimediaContent.additionalFee;
  }
});
```

De nuevo, esta pieza de código viola otro principio SOLID, en este caso el de SINGLE RESPONSIBILITY, el cual dicta que una clase sólo debería tener una responsabilidad, en otras palabras, sólo debería dedicarse a hacer una cosa. En este caso, RegisteredUser tiene demasiadas responsabilidades: gestionar la data relacionada con el usuario en específico (o así debería aunque no esté presente en el pseudocódigo), chequear qué tipo de contenido ha consumido el usuario, además de calcular el coste final que tiene que pagar el usuario.

Esto puede crear varios problemas en el futuro, por ejemplo, a la hora de cambiar la lógica relacionada con calcular el precio tendríamos que modificar RegisteredUser, lo cual no tiene sentido. Además, reduce la modularidad de la app ya que los servicios y el usuario están “tightly coupled”, y por tanto, los servicios no son reusables, ya que necesitan de RegisteredUser para “completar” su responsabilidad.

Para solucionar este problema, podemos utilizar “encapsulation” y polimorfismo, permitiendo así que cada clase gestiona su propia lógica para calcular el precio. Como la solución requiere mostrar todo el código, y la segunda parte de este ejercicio es mostrar una solución mejor, a continuación mostraré mi solución alternativa en pseudocódigo.

MI SOLUCIÓN PROPUESTA

1.- Primero, creamos constantes que van a ser reutilizadas en toda la app. Esto reduce la posibilidad de errores, bugs, además de que si tenemos que cambiar la string en el futuro, ahora sólo la tenemos que cambiar en un sitio, en vez de cambiarla en todas las instancias en las que se use “streaming” o “download”:

```
const STREAMING_SERVICE_TYPE = 'streaming';
const DOWNLOAD_SERVICE_TYPE = 'download';
```

2.- Refactorizamos para que MultimediaContent y PremiumContent sean los responsables de calcular el precio a pagar por el contenido concreto que representan:

```
// MultimediaContent es ahora responsable de calcular el precio del
contenido multimedia que representa
class MultimediaContent {
  getPrice(serviceType) {
    if (serviceType == STREAMING_SERVICE_TYPE) {
      return this.streamingPrice;
    } else if (serviceType == DOWNLOAD_SERVICE_TYPE) {
      return this.downloadPrice;
    } else {
      return -1; // Por simplicidad, retornamos -1 como señal de error.
    }
  }
}

// PremiumContent es responsable de añadir additionalFee, en vez de ser
RegisteredUser como antes
class PremiumContent extends MultimediaContent {
  getPrice(serviceType) {
    let basePrice = super.getPrice(serviceType);
    return basePrice + this.additionalFee;
  }
}
```

3.- Modificamos Service como “abstracta” y hacemos que sus subclases sean las responsables de calcular el precio basado en el tipo de contenido que gestiona el servicio.

```
// Clase "abstracta" o interface
class Service {
  getMultimediaContent() {
    // Lógica para retornar el contenido multimedia asociado al servicio
  }

  // Este método es "abstracto". A implementar en subclases
  getPrice() {}
}

class StreamingService extends Service {
  getPrice() {
    let content = this.getMultimediaContent();
    return content.getPrice(STREAMING_SERVICE_TYPE);
  }
}

class DownloadService extends Service {
  getPrice() {
    let content = this.getMultimediaContent();
    return content.getPrice(DOWNLOAD_SERVICE_TYPE);
  }
}
```

4.- Integramos todo con RegisteredUser y nos queda así:

```
class RegisteredUser {
  constructor(services = []) {
    this.services = services;
  }

  getTotal() {
    let total = 0;

    this.services.forEach((service) => {
      total += service.getPrice();
    });

    return total;
  }
}
```

En conclusión, antes la aplicación era muy inflexible, además de que las clases estaban “tightly coupled”, lo cual hacía que cualquier cambio en los servicios requiriera cambios en RegisteredUser. Además de que el código no era nada modular, lo cual impide reutilizar los servicios, ya que éstos dependen de que RegisteredUser “complete” la responsabilidad de éstos.

Con la nueva versión, podemos añadir nuevos servicios sin que influyan para nada en la clase RegisteredUser. Simplemente tendremos que asegurarnos que el servicio gestiona el precio adecuadamente a través de el MultimediaContent que esté sirviendo.