Práctica 1: Patrones de Diseño

Sesión 1. Patrones creacionales

Ejercicio 1 (Java): Patrón factoría abstracta y método factoría.

Vamos a implementar un programa que simule una sesión de pesca. Para ello tenemos un número N de peces el cual se desconoce hasta que comience la sesión. También existen 2 tipos de barcos: Pesquero Grande y Pesquero de redes, los cuales pescan un tipo de pez: Tiburones y Pez Pequeño respectivamente. Cada tipo de barco tiene una probabilidad de pescar un pez de su tipo: 60% y 90% respectivamente.

Deben seguirse las siguientes especificaciones:

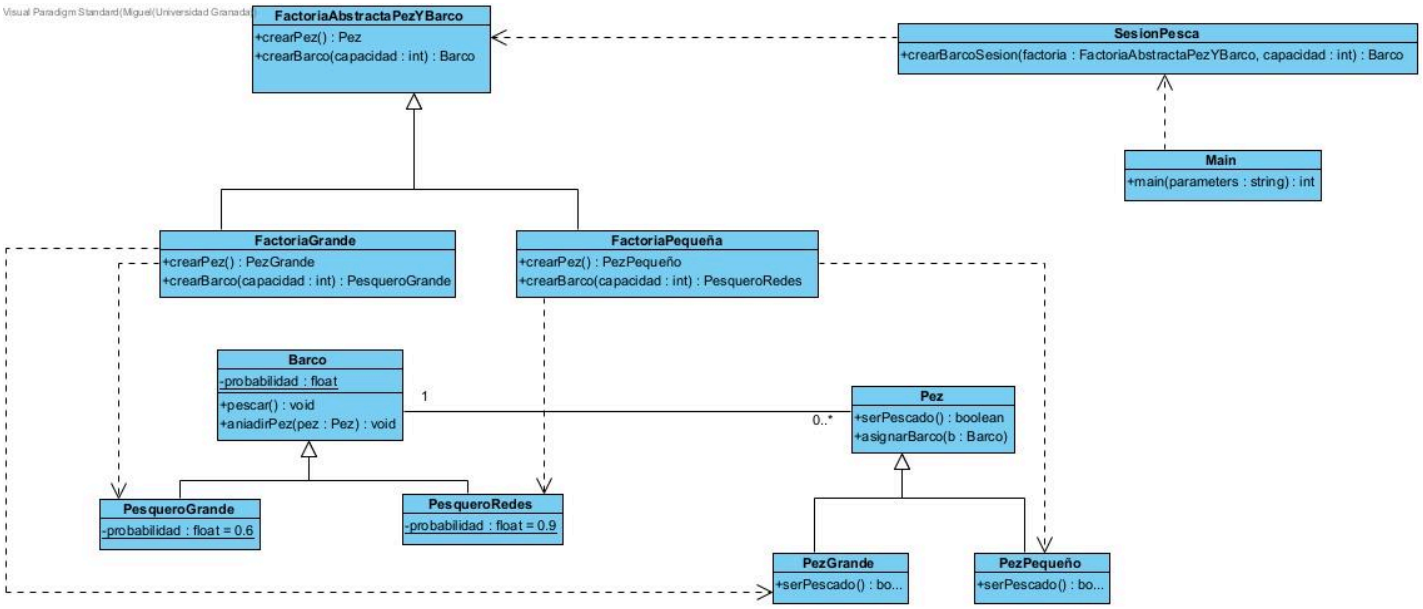
- Se implementará el patrón de diseño Factoría Abstracta junto con el patrón de diseño Método Factoría.
- Se usará Java como lenguaje de programación.
- Se usará un interfaz de usuario de texto.
- Se utilizarán hebras tanto para los barcos como para los peces.
- Se incluyen los dos tipos de barco PesqueroGrande y PesqueroRedes así como los dos tipos de peces.
- Se definirá la interfaz Java FactoriaPezYBarco para declarar los métodos de fabricación públicos:
    - crearPez que devuelve un objeto de alguna subclase de la clase abstracta Pesca y
    - crearBarco que devuelve un objeto de alguna subclase de la clase abstracta Barco

* La clase Barco tiene un atributo ArrayList<Pez>, con los peces que pesca en el barco.

Las clases factoría específicas heredarán de FactoriaPezYBarco y cada una de ellas se especializa en un tipo de barco y de peces: PesqueroGrande y PesqueroRedes y PezGrande y PezPequeño. Por consiguiente, tendremos dos clases factoría específicas: FactoriaGrande y FactoriaPequenios. Cada una implementa sus métodos crearBarco y crearPez.

Se definen las clases abstractas Barco y Pez las cuales se concretan en las clases PesqueroGrande y en PesqueroRedes para la clase Barco y PezGrande y PezPequeño para la clase Pez.

Diagrama de clases : Factoría abstracta con método factoría



Ejercicio 2 (C++): Patrón prototipo.

Vamos a implementar un programa que simule una sesión de pesca. Para ello tenemos un número N de peces el cual se desconoce hasta que comience la sesión. También existen 2 tipos de barcos: Pesquero Grande y Pesquero de redes, los cuales pescan un tipo de pez: Tiburones y Pez Pequeño respectivamente. Cada tipo de barco tiene una probabilidad de pescar un pez de su tipo: 60% y 90% respectivamente.
Deben seguirse las siguientes especificaciones:

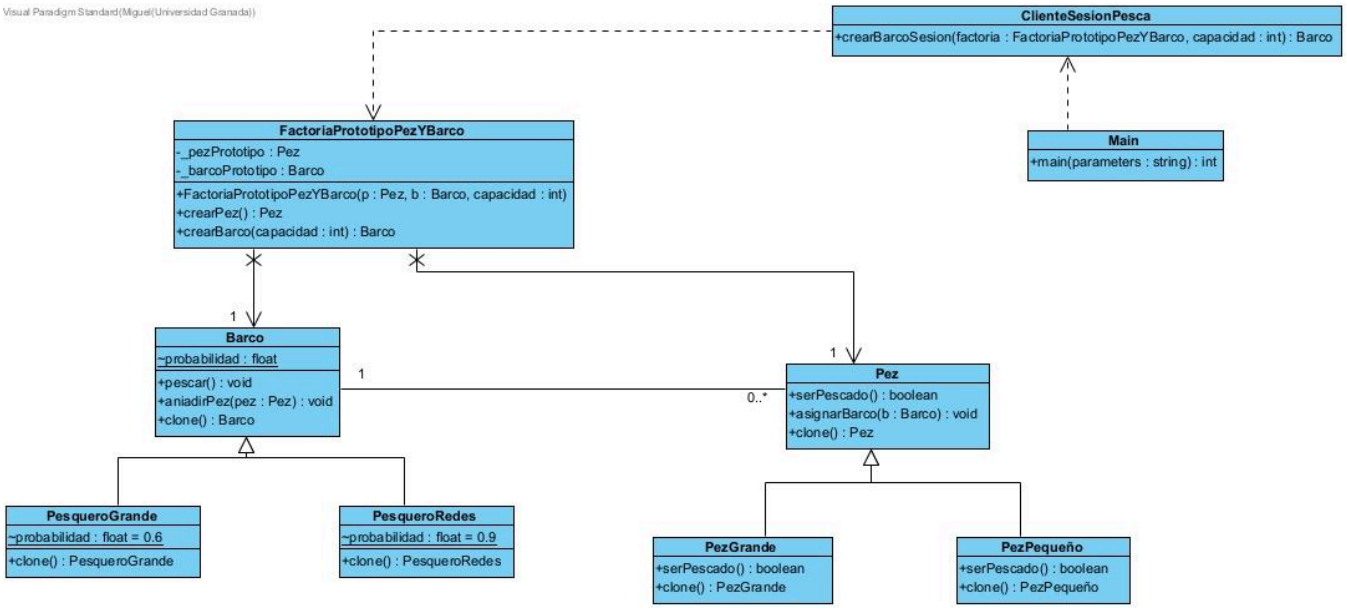
- Se implementará el patrón de diseño prototipo.
- Se usará c++como lenguaje de programación.
- Se usará un interfaz de usuario de texto.
- Se incluyen los dos tipos de barco PesqueroGrande y PesqueroRedes así como los dos tipos de peces.
- Se definirá la interfaz FactoriaPrototipoPezYBarco para declarar los métodos de fabricación públicos:
    - crearPez que devuelve un objeto de alguna subclase de la clase abstracta Pez
    - crearBarco que devuelve un objeto de alguna subclase de la clase abstracta Barco

* La clase Barco tiene un atributo ArrayList<Pez>, con los peces que pesca en el barco.
La clase FactoriaPrototipoPezYBarco le deben llegar una instancia de Pez y de Barco las cuales guardarán en las variables de instancia de la clase FactoriaPrototipoPezYBarco. En sus métodos crearPez() y crearBarco() se debe devolver una copia del Pez o del Barco correspondiente

Se definen las clases abstractas Barco y Pez las cuales se concretan en las clases PesqueroGrande y en PesqueroRedes para la clase Barco y PezGrande y PezPequeño para la clase Pez.

Posteriormente en el main se deben crear ambas factorias prototipo con las diferentes líneas de objetos (Grandes y Pequeños) y un objeto del tipo ClienteSesionPesca, el cual implementa el método crearBarcoSesion(<parametros>) simplemente creando el barco y añadiendo los peces que va obteniendo de la factoria que le llega por parámetro.Por ultimo devuelve el barco creado. Dentro del propio main se crearán ambos tipos de barcos (grandes y pequeños) y se llama al método pescar de cada uno.

Diagrama de clases patrón prototipo



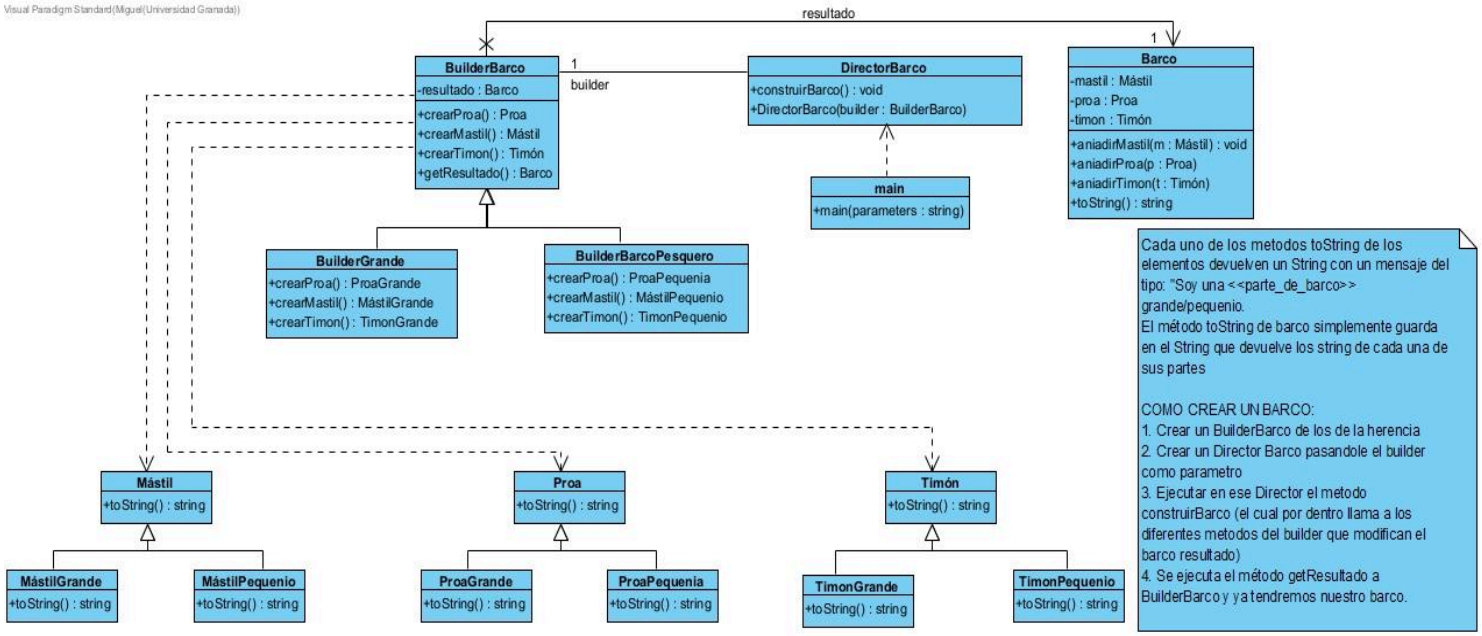
Ejercicio 3 (Ruby): Patrón builder.

Vamos a implementar un programa que simule una construcción de un barco, de parte del director de barco.

Deben seguirse las siguientes especificaciones:

- Se implementará el patrón de diseño builder.
- Se usará Ruby como lenguaje de programación.
- Se usará un interfaz de usuario de texto.
- Se incluyen el tipo barco, con sus partes correspondientes: Mástil, Proa y Timón.
- Se implementarán dos tipos de barcos que se construirán con distintas piezas utilizando herencia.

Diagrama de clases : Patrón Builder.



Sesión 2. Patrones conductuales y estructurales

Ejercicio 1 (Java): Patrón conductual observador y composite para la monitorización de datos F1

Programa en Java, utilizando el patrón de diseño observador, una aplicación con una GUI que simule los paneles de tiempos de una carrera de F1.

El programa debe crear un sujeto-observable con un coche de F1 y 2 paneles observables. Cada vez que un coche actualice los tiempos, lo hace de forma regular (mediante una hebra), deberá notificar el cambio a los observadores que tenga suscritos (comunicación push).

Deberán cumplirse las siguientes especificaciones:

- Se usará Java como lenguaje de programación, utilizándose la clase abstracta Observable de Java.util y la interfaz Observer de Java.
- Debe tenerse en cuenta que antes de llamar al método notifyObservers hay que invocar al método setChanged para dejar constancia de que se ha producido un cambio (si no se hace, el método notifyObservers no hará nada).

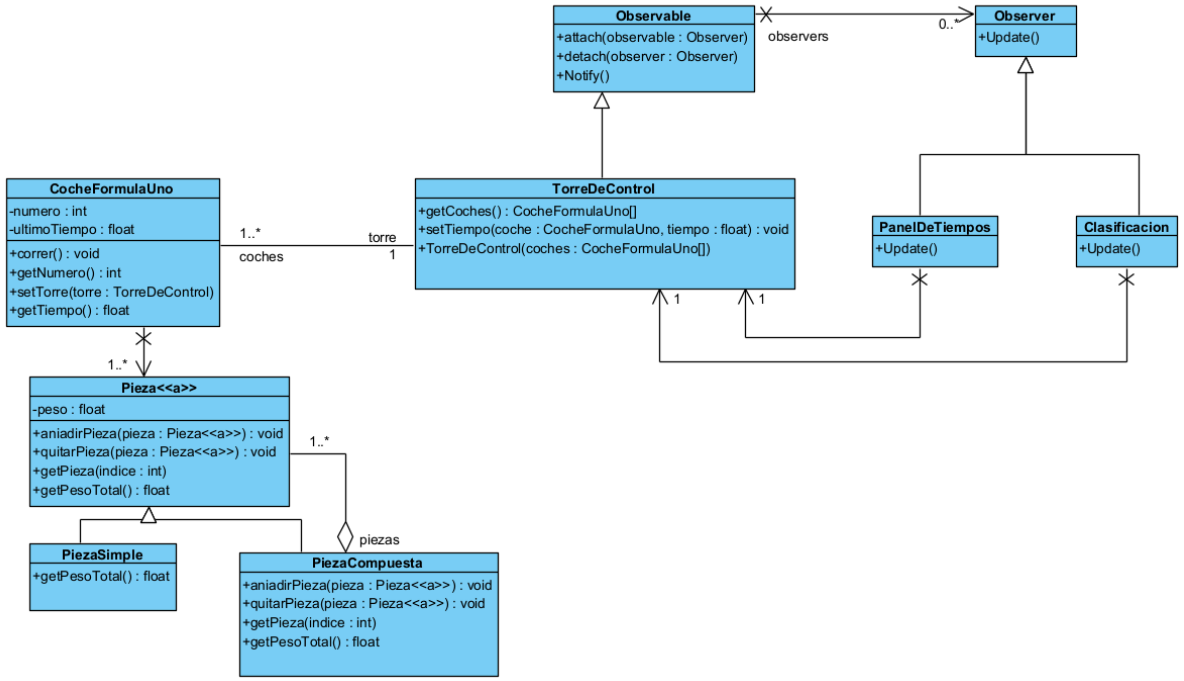
- Se creará una GUI donde cada observador aparezca en una ventana distinta.Se definirán 2 observadores, de la siguiente forma:

    - Observador PanelDeTiempos, debe mostrar los tiempos de cada uno de los coches que participen en la carrera. Se trata de un observador suscrito convencional (comunicación push) mediante método notifyObservers.
    - Observador Clasificacion, debe mostrar la posición en la que se encuentra cada uno de los coches, según los tiempos registrados por cada coche. Se trata de un observador suscrito convencional (comunicación push) mediante método notifyObservers.
    - También tendrá un botón para que cada vez que se le haga click haga una nueva ronda de vueltas y posteriormente se actualice.

Primero necesitaremos de la creación de un Arraylist de coches, que posteriormente utilizaremos para la creación de la torre de control. Posterior a esto haremos el set a cada uno de los coches con esa misma torre de control.

Para programar la simulación de la actualización de datos por parte del modelo y la petición de datos por parte del observador no suscrito, crearemos hebras en Java.

Diagrama de clases patrón conductual observador y composite



Ejercicio 2 (Java): Patrón estructural Filtros de intercepción para la construcción de comentarios.

Queremos representar un sistema de comentarios, al que le aplicaremos distintos filtros:

- FiltroContenido: Eliminamos las posibles palabras prohibidas que contenta la cadena (cabe aclarar que las palabras prohibidas estaran contenidas en un array de String en la propia clase)
- FiltroHora: Añadirá al comentario la hora en la que se realizó.

A continuación se explican las entidades de modelado necesarias para programar el estilo filtros de intercepción para este ejercicio.

- Objetivo (target): Representa el comentario al que se le aplicarán los filtros como servidorTarget.

- Filtro: Esta interfaz está implementada por las clases FiltroContenido y FiltroHora arriba comentadas. Estos filtros se aplicarán antes de que el mensaje se imprime por pantalla.

- Cliente: Es el objeto que envía la petición a la instancia de Objetivo. Como estamos usando el patrón Filtros de intercepción, la petición no se hace directamente, sino a través de un gestor de filtros (GestorFiltros) que enviará a su vez la petición a un objeto de la clase CadenaFiltros.

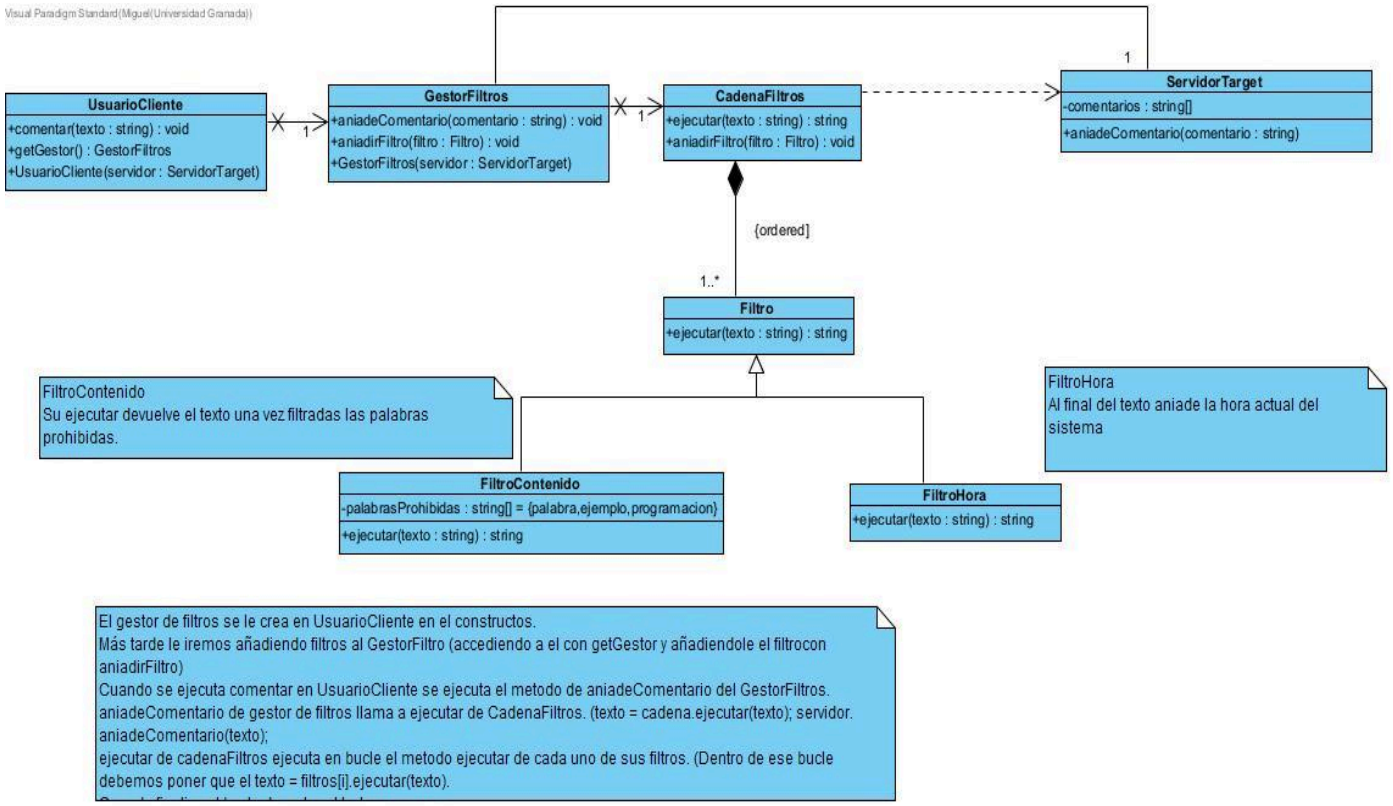
- CadenaFiltros:Tendrá una lista con los filtros que se aplicarán y los ejecutará en el orden en que fueron introducidos en la aplicación. Tras ejecutar estos filtros, se jecutará la tarea propia (método ejecutar de la clase Objetivo), todo dentro del método ejecutar de CadenaFiltros.

- GestorFiltros: Se encarga de gestionar los filtros: crea la cadena de filtros y tiene métodos para añadir filtros concretos y que se ejecute la petición por los filtros y el “objetivo” (método peticionFiltros).

Este ejercicio contará con interfaz gráfica, de tal manera que se introducirán los distintos comentarios a través de esta a los que se les pasará los filtros, contendrá dos partes:

- Cuadro de texto, donde se introducirá el comentario.
- Botón de enviar, que lanzará todo el proceso de filtrado, tras lo que se imprimirá por la pantalla.

Diagrama de clases Patrón estructural Filtros de Intercepción



El gestor de filtros se le crea en UsuarioCliente en el constructos.
Más tarde le iremos añadiendo filtros al GestorFiltro (accediendo a el con getGestor y añadiendole el filtrocon aniadirFiltro)
Cuando se ejecuta comentar en UsuarioCliente se ejecuta el metodo de aniadeComentario del GestorFiltros.
aniadeComentario de gestor de filtros llama a ejecutar de CadenaFiltros. (texto = cadena.ejecutar(texto); servidor.
aniadeComentario(texto);
ejecutar de cadenaFiltros ejecuta en bucle el metodo ejecutar de cada uno de sus filtros. (Dentro de ese bucle debemos poner que el texto = filtros[i].ejecutar(texto).

Ejercicio 3 (Ruby): Aplicación del patrón de comportamiento visitante

Utilizando este patrón se pretende recorrer una estructura de componentes que forman un equipo de cómputo (clase Barco), y construir un barco con los siguientes componentes: Mástil, Timón y Proa.

El programa mostrará una funcionalidad distinta en función del tipo de visitante para cada una de las partes del barco en función de qué tipo de visitante sea:

- VisitanteAreaBarcos: Utilizara las funcionalidades de los getMetros()
- VisitanteDistanciaBarcos: Utiliza la funcionalidad de getDistancia()

Cómo vendrán explicadas más tarde.

Debe adaptarse este código a Ruby, teniendo en cuenta que en Ruby no existe una palabra reservada para declarar una clase como abstracta. En Ruby una clase es abstracta cuando, o bien tiene declarado un método de ligadura dinámica (virtual) y no se implementa, o bien hereda un método de este tipo y no lo implementa, o ambas condiciones.

Las subclases de VisitanteBarco definirán algoritmos concretos que se aplican sobre la estructura de objetos que se obtiene de instanciar las subclases de Barco.

Así, se definirán las siguientes metodos de VisitanteAreaBarcos:

- VisitarMastil, VisitarProa y VisitarTimon: mostrará los nombres de las partes que componen un barco llamando al método getMetros().

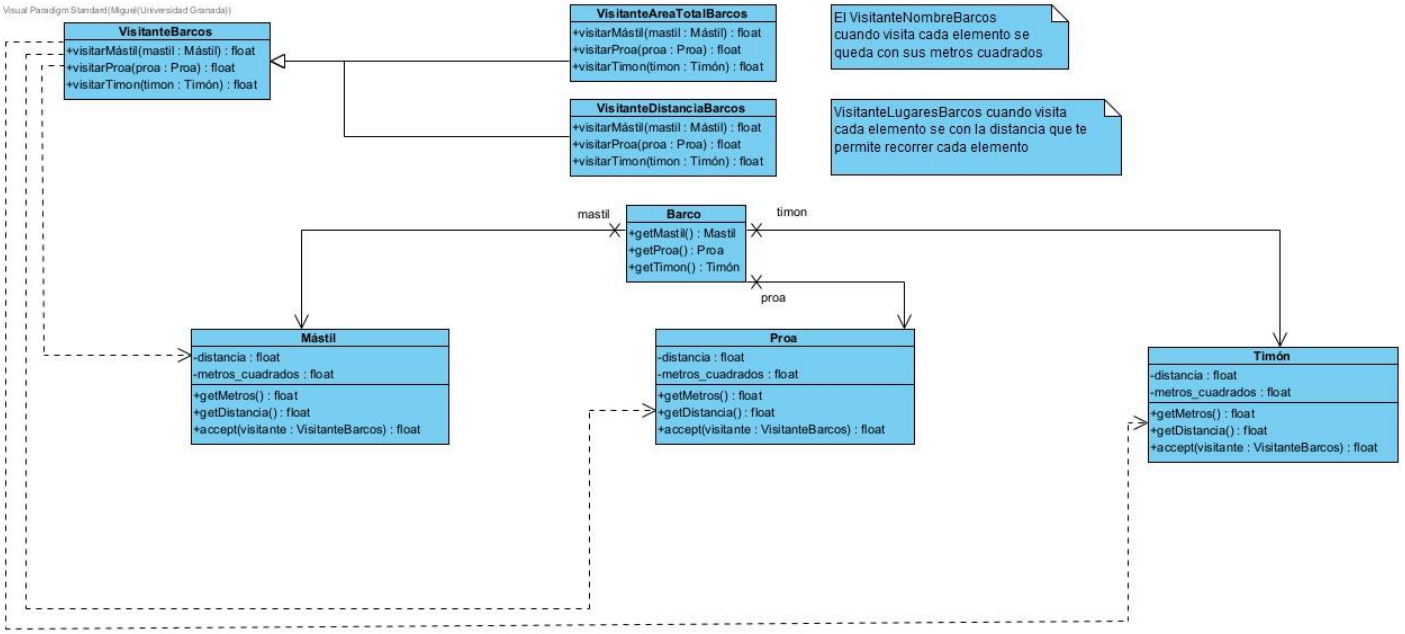
Así, se definirán las siguientes metodos de VisitanteDistanciaBarcos:

- VisitarMastil, VisitarProa y VisitarTimon: devuelve un float que indica la distancia que se es capaz de recorrer por tener cada parte del barco. Método getDistancia()

Cuando se definan las instancias de cada objeto, se deberá especificar atributos de metros cuadrados y distancia.

El programa principal (main) se encargará de crear varios Visitantes y hacer uso de su funcionalidad concreta para cada uno de ellos, de esta forma poder distinguir y hacer uso de este patrón de comportamiento.

Diagrama de clases : Patron de comportamiento visitante



Aclaraciones sobre ciertos métodos:

crearBarco(int cantidad): En un bucle que se ejecuta la cantidad pasada por parámetro de veces se le añade un pez creado anteriormente con crearPez(). En cada una de esas iteraciones se meten en el vector de peces de cada barco y se le asigna el barco al propio pez.

Método pescar(): Queremos que la funcionalidad principal del barco vaya aquí implementada. Se crea la hebra y se ejecuta un bucle que pase por todos los peces y ejecute el método serPescado

Método serPescado(): Devuelve bool, true si es pescado y false si no es pescado.. El como decidir si es pescado se hace de la siguiente manera: creamos un float entre 0 y 1 y si es menor que el float de probabilidad asociada a cada Barco (podemos acceder a esta probabilidad con el método getProbabilidad() de barco). Al final del método debe aparecer por pantalla “Soy un pez grande/pequeño y he sido/no he sido pescado”.

Para el patrón Factoria Abstracta el main quedaría tal que se crean 2 factorías con las que se crean 2 barcos (uno de cada tipo). A continuación se llamaría al método pescar en cada instancia de Barco.

Para el patrón prototipo esto variaría ligeramente, simplemente creando las factorías prototipos y obteniendo de ellas cada barco en los que llamamos al método pescar().

Informe sobre el trabajo realizado por cada miembro del Equipo de Desarrollo

Responsables

En general, además de los respectivos roles los responsables hemos realizado un trabajo de apoyo en las tareas de los demás. Si veíamos que un responsable estaba sobrepasado porque había varias dudas al mismo tiempo o surgía algún problema los otros dos responsables se han encargado de ayudarle, pero siempre considerando una equidad en la carga de trabajo.

Jose Luis Rico Ramos - Director de Proyecto

Se ha encargado de gestionar los distintos ejercicios a nivel de reparto así como de resolver dudas con la profesora sobre trámites de estos (qué nivel de especificación hace falta en los enunciados, saber si se debían hacer los ejercicios siguiendo los mismos patrones que están en los ejemplos o según nosotros quisiéramos…)

Además de realizar tareas de apoyo en la resolución de dudas.

Miguel Tirado Guzmán - Arquitecto Software

Ha sido el responsable de la creación de los ejercicios así como de la elaboración de los distintos diagramas de clases. También se ha encargado de ir resolviendo las dudas que podían ir surgiendo respecto al concepto del ejercicio en sí por parte de los programadores.

David Martínez Díaz - Experto

Se ha encargado de la resolución de dudas en lo que respecta a la teoría, tanto de cara a los programadores como a los propios responsables de la unidad.

Programadores

Para el reparto de los ejercicios decidimos que haríamos 3 parejas, las cuales harían un ejercicio por sesión. Además, también tuvimos en cuenta para la carga de trabajo la dificultad respecto al propio lenguaje de programación y al ejercicio (De esta forma, quienes tuviesen lenguajes de programación más difíciles tendrán ejercicios más simples y viceversa).
A su vez, antes de comenzar hicimos ciertas preguntas sobre el nivel de destreza de cada componente respecto a la programación, logrando así parejas equilibradas en ese sentido.

En general consideramos que los programadores han tenido un nivel de entrega excelente, obteniendo todos los conocimientos sobre patrones necesarios para la realización de las prácticas y han llevado los ejercicios al día, permitiéndonos corregirlos a la siguiente sesión.

Pareja 1 - { Gador Romero Prieto - Sergio Muñoz Gomez }

Ellos han sido los responsables de la realización de los ejercicios **Ejercicio 3 (Ruby): Patrón builder** y **Ejercicio 3 (Ruby): Aplicación del patrón de comportamiento visitante.** Su mayor fortaleza era la programación en Ruby (el cual era el lenguaje menos preferido por los demás integrantes), por lo que en concordancia los ejercicios fueron de un nivel promedio. En general no tuvieron muchas dudas en la realización de los ejercicios y comprendieron los patrones perfectamente, por lo que apenas tuvimos que hacer correcciones en su código.

Pareja 2 - { Raúl Morgado Saravia - Elena Ortega Contreras }

Ellos han sido los responsables de la realización de los ejercicios **Ejercicio 1 (Java): Patrón factoría abstracta y método factoría** y **Ejercicio 1 (Java): Patrón conductual observador y composite para la monitorización de datos F1.** Como Java era el lenguaje de programación preferido por los demás integrantes, tuvieron los ejercicios de mayor nivel (además tuvieron que realizar programación con hebras, algo que ninguno de los integrantes había realizado anteriormente). Plantearon dudas interesantes sobre cada patrón (para las cuales algunas tuvimos que preguntar a la profesora o buscar por internet). También consideramos que comprendieron perfectamente los patrones sobre los que trataron sus ejercicios.

Pareja 3 - { Ruben Rosales Tapia - Diego Velázquez Ortuño}

Ellos han sido los responsables de la realización de los ejercicios **Ejercicio 2: Patrón prototipo** y **Ejercicio 2 (Java): Patrón estructural Filtros de intercepción para la construcción de comentarios.** Tuvieron, al igual que sus compañeros, un buen nivel de desempeño, planteando muy buenas dudas y haciendo que todos comprendamos mejor los patrones de diseño. Hicieron ciertas correcciones muy oportunas sobre los diseños propuestos por los responsables.

Conclusiones

La realización de las prácticas de esta unidad 1 ha sido muy colaborativa entre los participantes, tanto a nivel de responsables como de programadores. Consideramos por tanto que el trabajo ha sido repartido de una manera equitativa y se han producido numerosas sinergias entre integrantes durante la realización y preparación de los ejercicios así como de las clases teóricas, logrando completar un muy buen trabajo.