INFORME EJERCICIO 2 PRÁCTICA PD: INCURSIONES NAVALES

En este ejercicio tenemos que crear un software para realizar incursiones navales con flotas y determinados nodos. Por lo tanto, crearemos flotas con determinados valores y que irán recorriendo una serie de nodos donde, dependiendo del tipo que tengan, sucederá una cosa u otra o incluso nada (caso del nodo Fin), es decir, que tendremos como una especie de mapa o de camino para seguir. En este problema tenemos tres tipos de nodos: Nodo de Fin (no tiene hijos), de Ruta Fija (que tiene a su vez nodos de Ataque Aéreo y de Tormenta Marina. Pueden tener solo un hijo) y de Bifurcación (que tiene a su vez nodos de Batalla y de Avistamiento. Pueden tener hasta 2 hijos). Dejaremos el código abierto para dejar opción a incorporar distintos tipos de nodos en un futuro.

Para dicho ejercicio emplearemos principios y patrones de diseño que nos permitan hacer el código lo más legible posible, facilitando la incorporación de nuevos tipos de nodos u otro tipo de operaciones en un futuro.

Tras esta pequeña introducción del problema, nos disponemos a explicar los principios y patrones de diseño elegidos:

PRINCIPIOS DE DISEÑO:

- **PRINCIPIO ABIERTO-CERRADO:** Este principio se utiliza en los nodos para permitir su extensión, pero no permitir la modificación de la clase. Por ejemplo, contamos con una clase abstracta nodo que permite que otras subclases hereden de ella, pero hacemos que estas no modifiquen su código. Insistimos, esto se puede observar en la clase abstracta Nodo y en sus subclases, como NodoBifurcación y NodoRutaFija y, además, también en las subclases de estas.
- **PRINCIPIO DE RESPONSABILIDAD ÚNICA:** Con este principio lo que buscamos es que cada objeto tenga una única responsabilidad. Esto en nuestro código es evidente en clases como Flota o Nodo. La responsabilidad de la clase Flota es la de crear una “flota” con unos determinados valores que el usuario deberá introducir, a su gusto. Con respecto a la clase Nodo y sus subclases pasa lo mismo. El usuario deberá escoger el tipo de Nodo que más le convenga (NodoAvistamiento, NodoBatalla, NodoAtaqueAéreo, NodoTormentaMarina o NodoFin) y darle unos valores. Con los nodos lo que nos permite es crear una especie de mapa o ruta que la flota va ha seguir hasta llegar a un fin o hasta acabar su hp.

- **PRINCIPIO DE SUSTITUCIÓN DE LISKOV:** Este principio lo usamos en los nodos. El concepto es que los objetos de una clase base deben poder ser reemplazados por objetos de sus subclases sin afectar al funcionamiento del programa. Esto ocurre en la clase abstracta nodo con el método abstracto operation. Cada subclase de nodo (como puede ser Nodo Fin, NodoRutaFija, NodoBifurcacion; y las subclases de estas, como NodoBatalla, etc.) implementa el método de una forma diferente pero coherente con el comportamiento definido por Nodo. Este principio es crucial en nuestro diseño pues garantiza que la arquitectura de nuestro programa sea robusta y flexible.

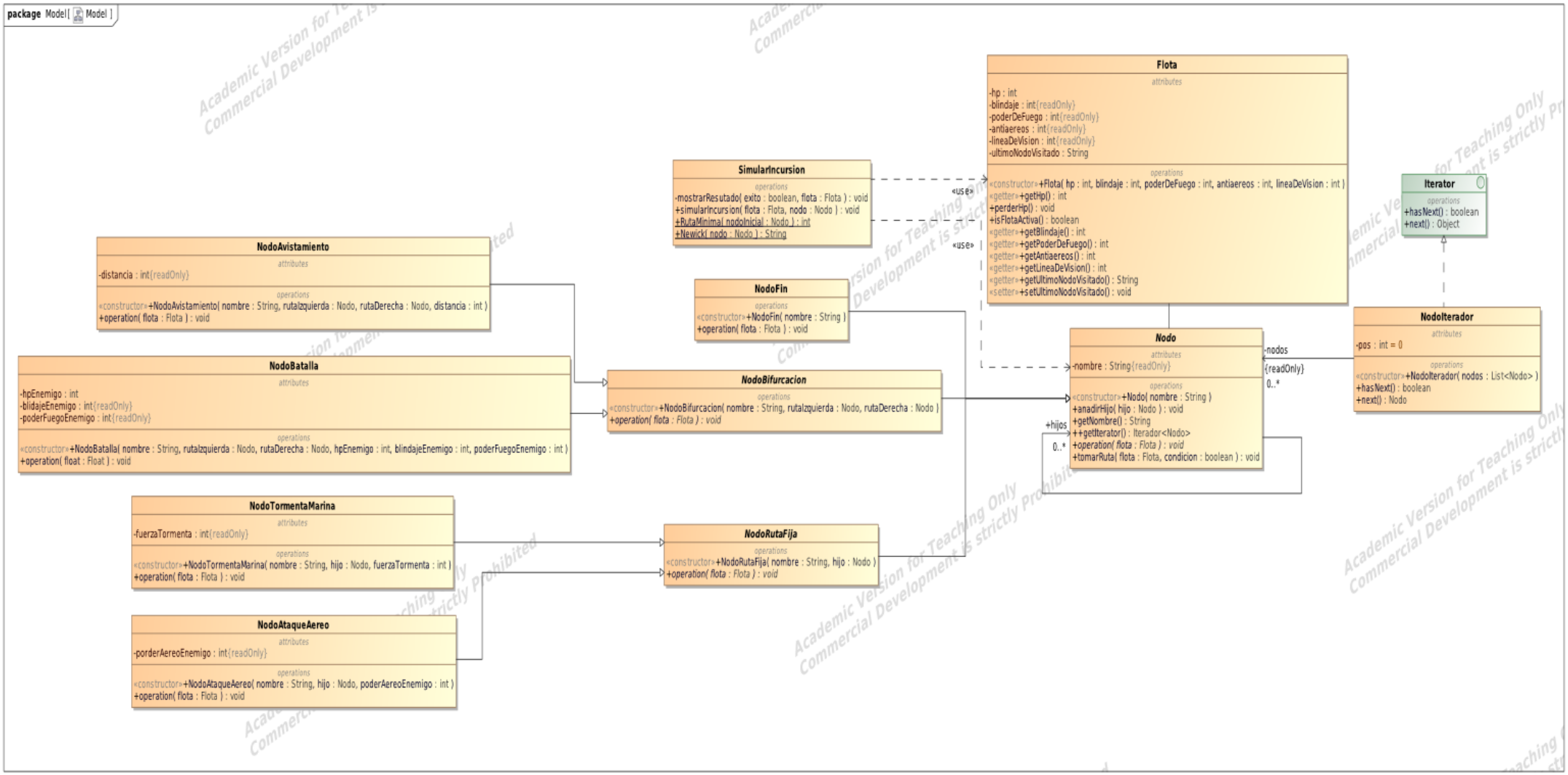
PATRONES DE DISEÑO:

- **PATRON COMPOSICIÓN:** Usado claramente en nuestro código, el patrón composición nos permite representar el mapa de la incursión naval pedida en el ejercicio.

La clase abstracta Nodo actúa como el componente base en la estructura de nuestro árbol. Cada nodo puede tener 0 o más nodos hijos, que se almacenan en la lista hijos presente en dicha clase. Sus subclases heredan esta capacidad y pueden tener sus propios hijos, pero vamos a dejar en cuenta varias cosas: un nodoFin no puede tener hijos, un nodoRutaFija (como podría ser NodoAtaqueAéreo o NodoTormentaMarina) pueden tener como máximo un hijo y nodoBifurcación (como nodoBatalla o nodoAvistamiento) pueden tener hasta un máximo de 2 hijos.

Con esta implementación de Nodos, dejamos abierto el código para que en un futuro se pueden añadir nuevos tipos de nodos que pueden tener los hijos deseados por el usuario. El hecho de poder tener hijos (que también podemos llamar rutas porque un hijo también puede tener un hijo o más) nos permite poder crear un Mapa donde, dependiendo del tipo de nodo, accederemos a un camino u otro. De todas destacar que los nodos, independientemente de su tipo, se manejan de manera similar como parte de la estructura del árbol lo que simplifica el código y hace que sea más manejable.

Para dejar más o menos detallada las estructuras de la clase abstracta Nodo y de sus subclases, os vamos a mostrar el diagrama de clases: **Comentar de todas formas que si no se puede visualizar de forma adecuada las imágenes de los diagramas en este informe, dichas imágenes están subidas en el repositorio de github en formato vectorial.**

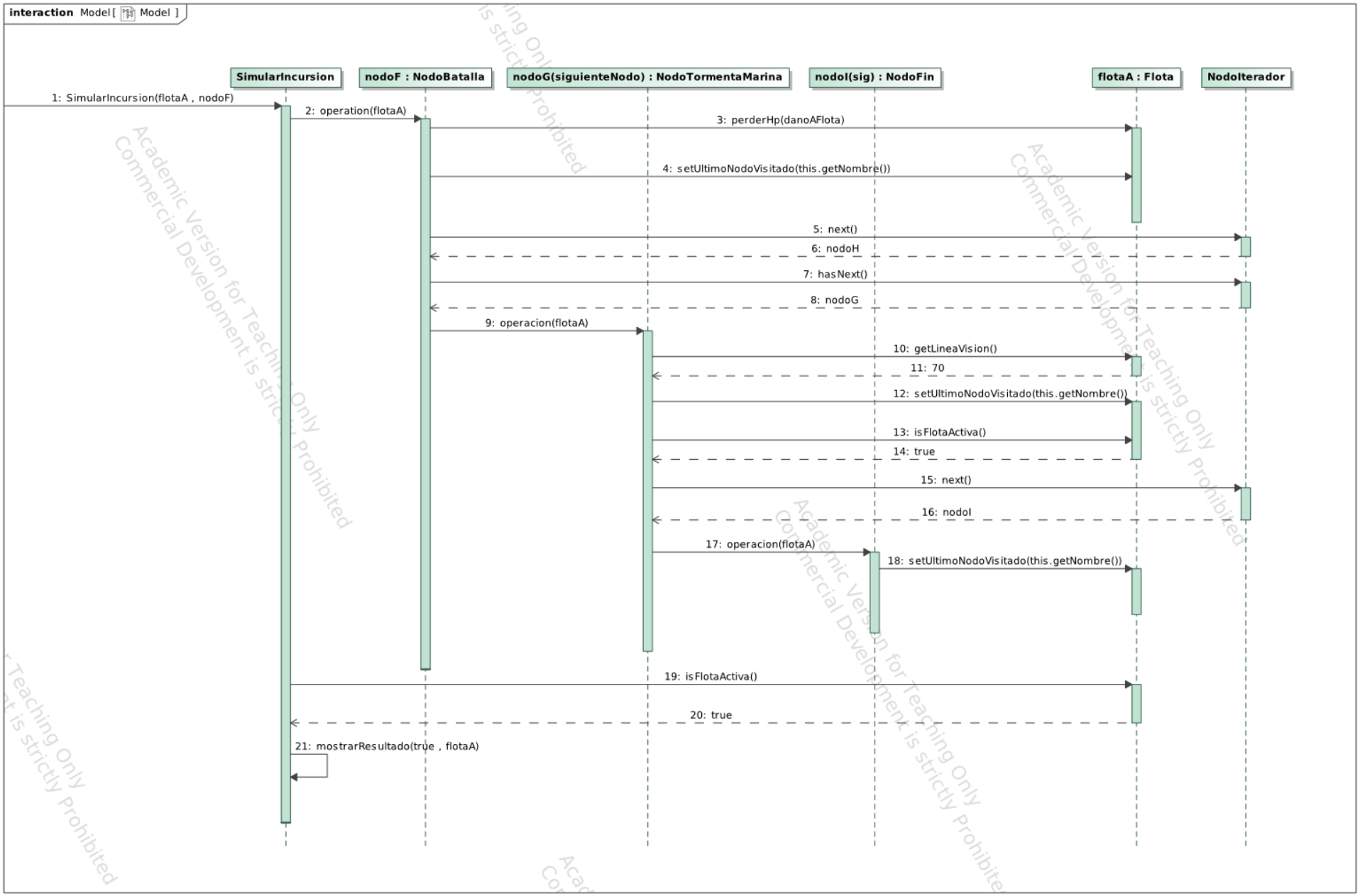

package Model [Model]
Flota
attributes
-hp : int
-blindaje : int{readOnly}
-poderDeFuego : int{readOnly}
-antiaereos : int{readOnly}
-lineaDeVision : int{readOnly}
-ultimoNodoVisitado : String
operations
«constructor»+Flota(hp : int, blindaje : int, poderDeFuego : int, antiaereos : int, lineaDeVision : int)
«getter»+getHp() : int
+perderHp() : void
+isFlotaActiva() : boolean
«getter»+getBlindaje() : int
«getter»+getPoderDeFuego() : int
«getter»+getAntiaereos() : int
«getter»+getLineaDeVision() : int
«getter»+getUltimoNodoVisitado() : String
«setter»+setUltimoNodoVisitado() : void
SimularIncursion
operations
-mostrarResutado(exito : boolean, flota : Flota) : void
+simularIncursion(flota : Flota, nodo : Nodo) : void
+RutaMinima(nodoInicial : Nodo) : int
+Newick(nodo : Nodo) : String
«use»
«use»
Iterator
operations
+hasNext() : boolean
+next() : Object
NodoAvistamiento
attributes
-distancia : int{readOnly}
operations
«constructor»+NodoAvistamiento(nombre : String, rutaIzquierda : Nodo, rutaDerecha : Nodo, distancia : int)
+operation(flota : Flota) : void
NodoFin
operations
«constructor»+NodoFin(nombre : String)
+operation(flota : Flota) : void
Nodo
attributes
-nombre : String{readOnly}
operations
«constructor»+Nodo(nombre : String)
+anadirHijo(hijo : Nodo) : void
+getNombre() : String
++getIterator() : Iterador<Nodo>
+operation(flota : Flota) : void
+tomarRuta(flota : Flota, condicion : boolean) : void
-nodos
{readOnly}
0..*
+hijos
0..*
NodoIterador
attributes
-pos : int = 0
operations
«constructor»+NodoIterador(nodos : List<Nodo>)
+hasNext() : boolean
+next() : Nodo
NodoBatalla
attributes
-hpEnemigo : int
-blidajeEnemigo : int{readOnly}
-poderFuegoEnemigo : int{readOnly}
operations
«constructor»+NodoBatalla(nombre : String, rutaIzquierda : Nodo, rutaDerecha : Nodo, hpEnemigo : int, blindajeEnemigo : int, poderFuegoEnemigo : int)
+operation(float : Float) : void
NodoBifurcacion
operations
«constructor»+NodoBifurcacion(nombre : String, rutaIzquierda : Nodo, rutaDerecha : Nodo)
+operation(flota : Flota) : void
NodoTormentaMarina
attributes
-fuerzaTormenta : int{readOnly}
operations
«constructor»+NodoTormentaMarina(nombre : String, hijo : Nodo, fuerzaTormenta : int)
+operation(flota : Flota) : void
NodoRutaFija
operations
«constructor»+NodoRutaFija(nombre : String, hijo : Nodo)
+operation(flota : Flota) : void
NodoAtaqueAereo
attributes
-porderAereoEnemigo : int{readOnly}
operations
«constructor»+NodoAtaqueAereo(nombre : String, hijo : Nodo, poderAereoEnemigo : int)
+operation(flota : Flota) : void

- **PATRÓN ITERATOR:** Es el otro patrón de diseño que usamos en este ejercicio y, aunque de primeras no parezca de gran utilizad ya que los hijos que pueden tener alguno de nuestros nodos es dos, el uso de este patrón está pensado por si en un futuro queremos añadir nuevos tipos de nodos.

Hemos implementado NodoIterator, que es una clase que implementa la interfaz Iterator<Nodo>. El hecho de implementar esta clase nos permite abstraer la lógica de la iteración ya que facilita cambios futuros en la estructura de datos sin afectar a las partes del código que usan el iterador. Por ejemplo, si queremos cambiar la estructura interna de almacenamiento de los nodos (estamos usando un ArrayList actualmente), a LinkedList por ejemplo, solo necesitaríamos ajustar NodoIterator, mientras que el resto del código que utiliza este iterador permanecerá sin cambios. Además, podemos añadir funcionalidades adicionales.

Con este patrón podemos recorrer la ruta de una forma sencilla. Para ellos recorremos secuencialmente la lista de nodos hijos almacenada en cada objeto Nodo hasta acabar con un hp 0 o inferior y fracasar en la incursión o llegar a un nodo fin y tener éxito.

Para finalizar, un ejemplo claro de su funcionamiento podemos verlo con el funcionamiento de simulación de ruta de la flota A (con hp = 9 y línea de visión = 70) del enunciado desde el nodo F. El resultado se puede verificar en el último test de nuestro ejercicio 2. El diagrama de secuencia es el siguiente:


interaction Model [Model]
SimularIncursion
nodoF : NodoBatalla
nodoG(siguienteNodo) : NodoTormentaMarina
nodoI(sig) : NodoFin
flotaA : Flota
NodoIterador
1: SimularIncursion(flotaA , nodoF)
2: operation(flotaA)
3: perderHp(danoAFlota)
4: setUltimoNodoVisitado(this.getNombre())
5: next()
6: nodoH
7: hasNext()
8: nodoG
9: operacion(flotaA)
10: getLineaVision()
11: 70
12: setUltimoNodoVisitado(this.getNombre())
13: isFlotaActiva()
14: true
15: next()
16: nodoI
17: operacion(flotaA)
18: setUltimoNodoVisitado(this.getNombre())
19: isFlotaActiva()
20: true
21: mostrarResultado(true , flotaA)