Planet Wars

Vamos a crear una aplicación basada en un popular juego de navegador, ogame. Será una versión bastante simplificada.
Vivimos en un planeta, que contiene recursos, flota de naves, defensas y dos tipos de tecnología.
En el universo que vivimos, existen otros planetas que quieren conquistarnos, y tenemos que hacer lo posible por defendernos.
Para defendernos tendremos que crear unas defensas y una flota capaz de contener y derribar la llegada de flotas enemigas.
Para crear estas defensas, necesitaremos recursos. Cada unidad, ya sea flota o defensa, tendrá un coste (Metal y Deuterio).
Para conseguir estos recursos. Nuestro planeta generará una cierta cantidad de recursos cada cierto tiempo, aparte, en cada batalla se generará una cantidad de escombros resultante de las naves derrotadas.
Si ganamos la batalla, podremos recoger esos escombros. En caso de perder la batalla, perderemos los escombros resultantes de la batalla.
Las tecnologías nos servirán para crear mejores unidades, y así tener una ventaja sobre la flota enemiga. Podemos avanzar en las tecnologías de defensa, que nos permitirá crear naves con un blindaje mejor, y la tecnología de ataque, que nos permitirá crear unidades con más poder de ataque.
Los ataques de flotas enemigos son constantes, por lo que no podemos dejar desatendido el planeta, tenemos que reemplazar lo antes posible las unidades que perdemos, así como mejorar en lo posible todas las características de nuestro planeta.

También tendremos que elegir convenientemente qué unidades formarán nuestro ejército, dado que cada unidad tendrá unas características diferentes que nos ayudará a que nuestras batallas sean más efectivas.

La siguiente tabla recoge los costes y los stats de las unidades militares que podemos crear:

		Cost		Stats			
		Metal	Deuterium	Armor	Atack power	Chance of Attack Again	Chance of Generate Wastings
Fleet	Ligth Hunter	3000	50	400	80	3	45
	Heavy Hunter	6500	50	1000	150	7	55
	Battleship	45000	7000	6000	1000	45	70
	Armored Ship	30000	15000	8000	700	70	85
Defenses	Missile Launcher	2000	0	200	80	5	45
	Ion Cannon	4000	500	1200	250	12	55
	Plasma Cannon	50000	5000	7000	2000	30	65

Vamos a describir cómo podemos implementar una aplicación que describa lo anterior.

Classe Planet

Necesitaremos una clase **Planet**, que representará nuestro planeta. Que tendrá las siguientes características:

```
int technologyDefense;
int technologyAtack;
int metal;
```

```
int deuterium;
int upgradeDefenseTechnologyDeuteriumCost;
int upgradeAttackTechnologyDeuteriumCost;
ArrayList<MilitaryUnit>[] army = new ArrayList[7];
```

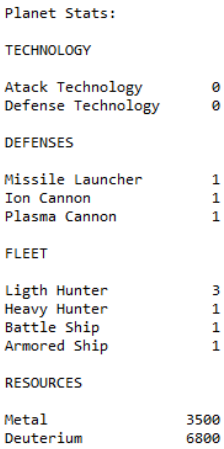
Las características de nuestro planeta son bastante intuitivas.
La característica más compleja es army, que representa nuestro ejército. Es un array de arrayLists, donde cada posición del array representa el conjunto de unidades del mismo tipo.

Army[0] → arrayList de Ligth Hunter
Army[1] → arrayList de Heavy Hunter
Army[2] → arrayList de Battle Ship
Army[3] → arrayList de Armored Ship
Army[4] → arrayList de Missile Launcher
Army[5] → arrayList de Ion Cannon
Army[6] → arrayList de Plasma Cannon

Los métodos necesarios en la clase Planet, además de los setters, getters y constructores, serán:

- **void upgradeTechnologyDefense, void upgradeTechnologyAttack**:
  Con estos métodos, pretendemos actualizar nuestras tecnologías de ataque/defensa, pero antes tendremos que comprobar que tenemos recursos suficientes para actualizar dicha tecnología. En caso de no tener recursos suficientes, lanzaremos una excepción del tipo **ResourceException**, que comentaremos más adelante.
  Tenemos que tener en cuenta, que cada vez que subimos un nivel de tecnología, la siguiente actualización será un porcentaje establecido más caro.
  Por ejemplo, si pasar del nivel 1 de defensa al nivel 2 de defensa costase 100 de deuterio, y el porcentaje establecido de incremento de precio fuese un 10%, pasar del nivel 2 al nivel 3 nos costaría 110 de deuterio. Este valor estaría previamente indicado en la característica upgradeDefenseTechnologyDeuteriumCost.
- **void newLigthHunter(int n), void newHeavyHunter(int n), void newBattleShip(int n), void newArmoredShip(int n), void newMissileLauncher(int n), void newIonCannon(int n), void newPlasmaCannon(int n)**. Estos métodos servirán para añadir nuevas unidades militares a nuestro ejército army mencionado anteriormente.
  Estos métodos reciben un entero n que indica el número de unidades que queremos añadir, si no tenemos suficientes recursos para añadir las unidades que queremos, lanzará una excepción del tipo ResourceException indicando el mensaje informativo. Pero se añadirán todas las unidades posibles que permitan nuestros recursos.
  Es decir, si queremos añadir 10 lanzamisiles, y sólo tenemos recursos para añadir 5, se lanzará una excepción del tipo ResourceException, pero se añadirán los 5 lanzamisiles que podemos construir y se nos mostrará también un mensaje informativo indicando el número de lanzamisiles que se han añadido.

- **Void printStats().** Este método nos servirá para mostrar una visión del estado de nuestro planeta, una posible salida cuando llamamos a este método podría ser:



Classe ResourceException

Para gestionar las excepciones lanzadas cuando queramos subir un nivel de alguna tecnología y no dispongamos de los recursos suficientes, vamos a crear nuestra propia excepción, la clase **ResourceException**, que extenderá de Exception, y nos mostrará un mensaje descriptivo del motivo de la excepción.

Classe Ship

Para crear nuestra flota, es decir, unidades militares no defensivas, vamos a crear la clase abstracta Ship.

Esta clase tendrá como propiedades:

```
int armor;
int initialArmor;
int baseDamage;
```

armor será la armadura restante de nuestra nave durante las batallas.
InitialArmor será la armadura inicial de nuestra nave, es decir, la armadura al momento de crearla.
BaseDamage el poder de ataque, le llamaremos baseDamage pensando en alguna posible ampliación del proyecto.
Esta clase implementará las interfaces MilitaryUnit i Variables.

Interface MilitaryUnit

MilitaryUnit será una interfaz que implementarán todas nuestras unidades (defensas y naves), y que nos servirá para dos cosas, para gestionar todas las unidades militares por igual, y para obligarnos a implementar los métodos que serán comunes a todas las unidades militares.

La interfaz MilitaryUnit que tan sólo declarará los métodos:

- abstract int attack();
  Nos devolverá el poder de ataque que tenga la unidad.
- abstract void tekeDamage(int receivedDamage);
  Restará a nuestro blindaje el daño recibido por parámetro.
- abstract int getActualArmor();
  Nos devolverá el blindaje que tengamos actualmente, después de haber recibido un ataque.

- abstract int getMetalCost();
  Nos devolverá el coste de Metal que tiene crear una nueva unidad.
- abstract int getDeuteriumCost();
  Nos devolverá el coste de Deuterium que tiene crear una nueva unidad.
- abstract int getChanceGeneratinWaste();
  Nos la probabilidad de generar residuos al ser totalmente eliminada (blindaje 0 o inferior).
- abstract int getChanceAttackAgain();
  Nos la probabilidad de generar volver a atacar.
- abstract void resetArmor();
  Nos restablecerá nuestro blindaje a su valor original.

Estos métodos se tendrán que implementar en todas las clases que hereden de Ship.

Classes LigthHunter, HeavyHunter, BattleShip, ArmoredShip

Para cada tipo de unidad de nuestra flota, crearemos una clase que heredará de la clase Ship.
Todas ellas dispondrán de dos constructores.

Constructor 1:
Al que le pasaremos dos parámetros; int armor, int baseDamage.

Para establecer el blindaje de nuestra nave, armor, tendremos que tener en cuenta dos constantes, por ejemplo, en el caso de un cazador ligero.
ARMOR_LIGTHHUNTER i *PLUS_ARMOR_LIGTHHUNTER_BY_TECHNOLOGY*.
La primera es la armadura que tendrá nuestro cazador ligero en el caso que nuestra tecnología de defensa sea cero, y la segunda es el plus de armadura que tendrá en función del nivel de tecnología de defensa que tengamos.

Si *ARMOR_LIGTHHUNTER=1000, PLUS_ARMOR_LIGTHHUNTER_BY_TECHNOLOGY=5* y nuestro nivel de tecnología =2
La armadura de un nuevo cazador ligero será de 1000 + (2*5)1000/100 = 1100.
Es decir, la armadura con un nivel de tecnología de defensa igual a cero, más el 2*5 = 10 %.

En caso de tener un nivel de tecnología en defensa igual a 3, la armadura de un nuevo cazador ligero sería 1000 + (3*5)%1000 = 1150.

Para establecer al poder de ataque baseDamage, tendremos que tener en cuenta las constantes, BASE_DAMAGE_LIGTHHUNTER i PLUS_ATTACK_ LIGTHHUNTER_BY TECHNOLOGY.
El funcionamiento será el mismo que hemos descrito en el establecimiento de la armadura, pero ahora teniendo en cuenta el nivel de tecnología de ataque.

Por último, estableceremos la propiedad initialArmor al mismo valor que armor, que nos servirá para restablecer la armadura a su valor original después de una batalla.

Constructor 2:

Al que no le pasaremos ningún parámetro.
Este constructor creará una nueva y establecerá la armadura y el poder de ataque a sus valores básicos, sin tener en cuenta las tecnologías.

Este constructor nos servirá para crear unidades para un supuesto ejército enemigo desde nuestra clase principal Main.

Cada una de estas clases tendrá que implementar los métodos definidos en la interfaz MilitaryUnit.

Classe Defense

Para crear nuestra defensa, vamos a crear la clase abstracta Defense.
Esta clase tendrá como propiedades:

int armor;
int initialArmor;
int baseDamage;

armor será la armadura restante de nuestra defensa durante las batallas.
InitialArmor será la armadura inicial de nuestra defensa, es decir, la armadura al momento de crearla.
BaseDamage el poder de ataque, le llamaremos baseDamage pensando en alguna posible ampliación del proyecto.
Esta clase implementará las interfaces MilitaryUnit i Variables.

Classes MissileLauncher, IonCannon, PlasmaCannon

Para cada tipo de unidad de nuestra defensa, crearemos una clase que heredará de la clase Defense. Todas ellas dispondrán de un único constructor al que le pasaremos como argumentos int armor, int baseDamage.

El funcionamiento del constructor de nuestras defensas, será igual que el de nuestra flota.

Se establecerán el blindaje y la armadura dependiendo del nivel de nuestras tecnologías.

Cada una de estas clases tendrá que implementar los métodos definidos en la interfaz MilitaryUnit.

Interficie Variables

Esta será una interfaz que nos permitirá parametrizar el juego, de forma que después de testearlo varias veces, encontremos unos valores equilibrados, en principio la podemos definir como sigue:

```
public interface Variables {
          // resources available to create the first enemy fleet
          public final int DEUTERIUM_BASE_ENEMY_ARMY = 26000;
          public final int METAL_BASE_ENEMY_ARMY = 180000;


          // percentage increase of resources available to create enemy fleet
          public final int ENEMY_FLEET_INCREASE = 6;


          // resources increment every minute
          public final int PLANET_DEUTERIUM_GENERATED = 1500;
          public final int PLANET_METAL_GENERATED = 5000;


          // TECHNOLOGY COST
          public final int UPGRADE_BASE_DEFENSE_TECHNOLOGY_DEUTERIUM_COST = 2000;
          public final int UPGRADE_BASE_ATTACK_TECHNOLOGY_DEUTERIUM_COST = 2000;
          public final int UPGRADE_PLUS_DEFENSE_TECHNOLOGY_DEUTERIUM_COST = 60;
          public final int UPGRADE_PLUS_ATTACK_TECHNOLOGY_DEUTERIUM_COST = 60;


          // COST SHIPS
          public final int METAL_COST_LIGTHHUNTER = 3000;
          public final int METAL_COST_HEAVYHUNTER = 6500;
          public final int METAL_COST_BATTLESHIP = 45000;
```

```
        public final int METAL_COST_ARMOREDSHIP = 30000;
        public final int DEUTERIUM_COST_LIGTHHUNTER = 50;
        public final int DEUTERIUM_COST_HEAVYHUNTER = 50;
        public final int DEUTERIUM_COST_BATTLESHIP = 7000;
        public final int DEUTERIUM_COST_ARMOREDSHIP = 15000;


        // COST DEFENSES


        public final int DEUTERIUM_COST_MISSILELAUNCHER = 0;
        public final int DEUTERIUM_COST_IONCANNON = 500;
        public final int DEUTERIUM_COST_PLASMACANNON = 5000;
        public final int METAL_COST_MISSILELAUNCHER = 2000;
        public final int METAL_COST_IONCANNON = 4000;
        public final int METAL_COST_PLASMACANNON = 50000;


        // array units costs
        public final int[] METAL_COST_UNITS =
{METAL_COST_LIGTHHUNTER,METAL_COST_HEAVYHUNTER,METAL_COST_BATTLESHIP,METAL_COST_ARMOREDSHIP,METAL_COST_MISSILELAUNCHER,METAL_COST_IONCANNON,METAL_
COST_PLASMACANNON};
        public final int[] DEUTERIUM_COST_UNITS =
{DEUTERIUM_COST_LIGTHHUNTER,DEUTERIUM_COST_HEAVYHUNTER,DEUTERIUM_COST_BATTLESHIP,DEUTERIUM_COST_ARMOREDSHIP,DEUTERIUM_COST_MISSILELAUNCHER,DEUTERI
UM_COST_IONCANNON,DEUTERIUM_COST_PLASMACANNON};


        // BASE DAMAGE SHIPS
        public final int BASE_DAMAGE_LIGTHHUNTER = 80;
        public final int BASE_DAMAGE_HEAVYHUNTER = 150;
        public final int BASE_DAMAGE_BATTLESHIP = 1000;
        public final int BASE_DAMAGE_ARMOREDSHIP = 700;


        // BASE DAMAGE DEFENSES


        public final int BASE_DAMAGE_MISSILELAUNCHER = 80;
        public final int BASE_DAMAGE_IONCANNON = 250;
        public final int BASE_DAMAGE_PLASMACANNON = 2000;


        // REDUCTION_DEFENSE
        public final int REDUCTION_DEFENSE_IONCANNON = 100;


        // ARMOR SHIPS
        public final int ARMOR_LIGTHHUNTER = 400;
        public final int ARMOR_HEAVYHUNTER = 1000;
        public final int ARMOR_BATTLESHIP = 6000;
        public final int ARMOR_ARMOREDSHIP = 8000;


        // ARMOR DEFENSES
        public final int ARMOR_MISSILELAUNCHER = 200;
        public final int ARMOR_IONCANNON = 1200;
        public final int ARMOR_PLASMACANNON = 7000;


        //fleet armor increase percentage per tech level
        public final int PLUS_ARMOR_LIGTHHUNTER_BY_TECHNOLOGY = 5;
        public final int PLUS_ARMOR_HEAVYHUNTER_BY_TECHNOLOGY = 5;
        public final int PLUS_ARMOR_BATTLESHIP_BY_TECHNOLOGY = 5;
        public final int PLUS_ARMOR_ARMOREDSHIP_BY_TECHNOLOGY = 5;



        // defense armor increase percentage per tech level
        public final int PLUS_ARMOR_MISSILELAUNCHER_BY_TECHNOLOGY = 5;
        public final int PLUS_ARMOR_IONCANNON_BY_TECHNOLOGY = 5;
        public final int PLUS_ARMOR_PLASMACANNON_BY_TECHNOLOGY = 5;



        // fleet attack power increase percentage per tech level
        public final int PLUS_ATTACK_LIGTHHUNTER_BY_TECHNOLOGY = 5;
        public final int PLUS_ATTACK_HEAVYHUNTER_BY_TECHNOLOGY = 5;
        public final int PLUS_ATTACK_BATTLESHIP_BY_TECHNOLOGY = 5;
        public final int PLUS_ATTACK_ARMOREDSHIP_BY_TECHNOLOGY = 5;



        // Defense attack power increase percentage per tech level
        public final int PLUS_ATTACK_MISSILELAUNCHER_BY_TECHNOLOGY = 5;
        public final int PLUS_ATTACK_IONCANNON_BY_TECHNOLOGY = 5;
        public final int PLUS_ATTACK_PLASMACANNON_BY_TECHNOLOGY = 5;



        // fleet probability of generating waste
        public final int CHANCE_GENERATNG_WASTE_LIGTHHUNTER = 55;
        public final int CHANCE_GENERATNG_WASTE_HEAVYHUNTER = 65;
        public final int CHANCE_GENERATNG_WASTE_BATTLESHIP = 80;
        public final int CHANCE_GENERATNG_WASTE_ARMOREDSHIP = 90;


        // Defense probability of generating waste
        public final int CHANCE_GENERATNG_WASTE_MISSILELAUNCHER = 55;
        public final int CHANCE_GENERATNG_WASTE_IONCANNON = 65;
        public final int CHANCE_GENERATNG_WASTE_PLASMACANNON = 75;
```

```
    // fleet chance to attack again
    public final int CHANCE_ATTACK_AGAIN_LIGTHHUNTER = 3;
    public final int CHANCE_ATTACK_AGAIN_HEAVYHUNTER = 7;
    public final int CHANCE_ATTACK_AGAIN_BATTLESHIP = 45;
    public final int CHANCE_ATTACK_AGAIN_ARMOREDSHIP = 70;


    //Defense chance to attack again
    public final int CHANCE_ATTACK_AGAIN_MISSILELAUNCHER = 5;
    public final int CHANCE_ATTACK_AGAIN_IONCANNON = 12;
    public final int CHANCE_ATTACK_AGAIN_PLASMACANNON = 30;


    // CHANCE ATTACK EVERY UNIT
    // LIGTHHUNTER, HEAVYHUNTER, BATTLESHIP, ARMOREDSHIP, MISSILELAUNCHER, IONCANNON, PLASMACANNON
    public final int[] CHANCE_ATTACK_PLANET_UNITS = {5,10,15,40,5,10,15};
    // LIGTHHUNTER, HEAVYHUNTER, BATTLESHIP, ARMOREDSHIP
    public final int[] CHANCE_ATTACK_ENEMY_UNITS = {10,20,30,40};


    // percentage of waste that will be generated with respect to the cost of the units
    public final int PERCENTATGE_WASTE = 70;


}
```

Observamos que todo son constantes que caracterizan la dificultad del juego.

Classe Battle

La clase Battle nos servirá para gestionar el desarrollo de las batallas. Ésta será posiblemente la clase más compleja de implementar del proyecto.

Los variables obligatorias con las que tendremos que trabajar:

- **ArrayList<MilitaryUnit>[] planetArmy** → para almacenar la flota enemiga
- **ArrayList<MilitaryUnit>[] enemyArmy** → para almacenar la flota de nuestro planeta.
- **ArrayList[][] armies** → que es un array de ArrayList de dos filas y siete columnas, donde almacenaremos nuestro ejército en la primera fila, y el ejército enemigo en la segunda fila;

- **String battleDevelopment** → Donde guardamos todo el desarrollo de la batalla paso a paso

- **int[][] initialCostFleet** → coste de metal deuterio de los ejercitos iniciales initialCostFleet = [[metal][deuterio],[metal][deuterio]] , donde initialCostFleet[0] costes unidades del planeta , initialCostFleet[1] costes unidades enemigas. Lo necesitamos para saber las pérdidas en materiales de cada flota.

- **int initialNumberUnitsPlanet, initialNumberUnitsEnemy** → La batalla se acabará cuando uno de los dos ejércitos se quede con el 20% o menos de sus unidades iniciales, por tanto es necesario saber la cantidad de unidades iniciales de cada ejército.

- **int[] wasteMetalDeuterium** → residuos generados en la batalla [metal, deuterio].

- **int[] enemyDrops, int[] planetDrops**, necesarios para generar reporte de batalla, y para calcular las pérdidas materiales de cada ejército.

- **int[][] resourcesLooses** → array de dos filas y tres columnas, resourcesLooses[0] = {perdidas metal planeta, perdidas deuterio planeta, perdidas metal planeta + 5*

perdidas deuterio planeta}, resourcesLooses[1] lo mismo pero para el ejercito enemigo.
Lo de multiplicar por 5 las pérdidas de deuterio, es debido al mayor valor de este material. Para decidir el ganador, será que que tenga el numero menor en la tercera columna. ResourcesLooses[0][2] y ResourcesLooses[1][2], que representan las pérdidas ponderadas.

- **int[][] initialArmies** → Array de dos filas y 7 columnas. Servirá para cuantificar cada tipo de unidad de los ejércitos iniciales.
  InitialArmies[0] serán las unidades iniciales de nuestro planeta y InitialArmies[1] las unidades iniciales enemigas.
  InitialArmies[0][0] cazadores ligeros en nuestro planeta antes de iniciar batalla.
  InitialArmies[0][1] cazadores pesados en nuestro planeta antes de iniciar batalla.
  .
  .

  Este array nos ayudará a calcular los costes de las flotas iniciales y por tanto, las pérdidas.

- **int[] actualNumberUnitsPlanet, int[] actualNumberUnitsEnemy** --> arrays que cuantifican las unidades actuales de cada grupo, tanto para el planeta, como para el enemigo. El orden seria:
  actualNumberUnitsPlanet[0] --> cazadores ligeros
  actualNumberUnitsPlanet[0] --> cazadores pesados
  actualNumberUnitsPlanet[0] --> Naves de battalla
  actualNumberUnitsPlanet[0] --> Acorazados
  actualNumberUnitsPlanet[0] --> Lanzamisiles
  actualNumberUnitsPlanet[0] --> Cañones de iones
  actualNumberUnitsPlanet[0] --> Cañones de Plasma
  Es necesario tener contabilizadas las unidades actuales de cada clase, debido a que en la mecánica de batalla, se escoge un defensor en función de la cantidad de unidades de cada clase.

Mecánica de la batalla:

Partimos de dos ejércitos con diferentes unidades, el planeta tiene defensas y el ejercito que nos ataca no tiene defensas.
A cada tipo de unidad diferente le llamamos grupo, para abreviar.
Empieza a atacar un ejercito aleatoriamente.
De ese ejército, cada grupo tiene una probabilidad de atacar, definido en la interfaz Variables.

```
// LIGTHHUNTER, HEAVYHUNTER, BATTLESHIP, ARMOREDSHIP, MISSILELAUNCHER, IONCANNON, PLASMACANNON
public final int[] CHANCE_ATTACK_PLANET_UNITS = {5,10,15,40,5,10,15};
// LIGTHHUNTER, HEAVYHUNTER, BATTLESHIP, ARMOREDSHIP
public final int[] CHANCE_ATTACK_ENEMY_UNITS = {10,20,30,40};
```

Es decir, que la probabilidad de ser los acorazados los que ataquen es un 40% en ambos casos, la de los cazas ligeros es un 5% en el caso del planeta y un 10% en caso de la flota enemiga

Más adelante explicaremos un algoritmo sencillo para hacer uso de los arrays de probabilidades.

Una vez se ha escogido el grupo atacante, se selecciona una nave al azar del grupo que será la nave atacante.

Seguidamente escogeremos un grupo defensor, y para ello, las probabilidades de escoger cada uno de los grupos, estará basada en la cantidad de unidades del grupo.
Si por ejemplo la flota defensora está compuesta por:

90 cazadores ligeros
60 cazadores pesados
30 naves de batalla.
20 acorazados

La probabilidad de escoger un defensor del primer grupo (cazadores ligeros) será de un 45%, es decir:

100*(Cantidad de cazadores ligeros) / (total de unidades) = 9000/200 = 45

La probabilidad de escoger un defensor del segundo grupo será del 30%.

100*(Cantidad de cazadores ligeros) / (total de unidades) = 6000/200 = 30.

También explicaremos un algoritmo sencillo para escoger un grupo en función de las probabilidades definidas arriba.

Una vez escogido el grupo defensor, escogeremos uno de los defensores dentro del grupo de forma aleatoria.

Una vez escogidas las unidades atacante y defensora, la atacante infligirá el daño igual a su poder de ataque y reducirá el blindaje de la defensora en una cantidad igual al poder de ataque de la unidad atacante.

Si el blindaje de la unidad defensora se queda en cero o un número negativo, se eliminará.
Antes de eliminarla, comprobaremos si genera residuos, la probabilidad de generar residuos está definida en la interfaz Variables, por ejemplo, int CHANCE_GENERATNG_WASTE_LIGTHHUNTER = 55

Si se generan residuos, se generará un porcentaje de los recursos que costó construir dicha unidad, este porcentaje estará definido en la interfaz Variables, final int PERCENTATGE_WASTE = 70. Igual para todas las unidades.

Una vez realizada las acciones antes mencionadas, comprobaremos si la unidad atacante tiene opción de atacar de nuevo, la probabilidad de atacar de nuevo de cada unidad, está descrita en la interfaz Variables, por ejemplo, int CHANCE_ATTACK_AGAIN_BATTLESHIP = 45, indica que las naves de batalla tienen un 45% de probabilidades de repetir ataque.

En caso de repetir ataque, se vuelve a escoger un grupo aleatorio, y después un defensor del grupo escogido y volvemos a realizar los mismos pasos.

En caso de no repetir ataque, se cambia el turno, el ejercito atacante pasa a ser ejército defensor, el defensor pasa a ser atacante y aplicamos la misma mecánica descrita.

Este proceso se repetirá mientras uno de los dos ejércitos mantenga al menos el 20% de las unidades totales del que estaba compuesto antes de empezar el ataque.

Esta probabilidad del 20 por ciento, también podría definirse en la interfaz Variables, a modo de calibrar el juego.

Al final de la batalla, si ganamos la batalla, podremos quedarnos con los residuos generados. Para calcular quien gana la batalla, calcularemos el siguiente número para cada flota:

(total de pérdidas de metal) + (total de pérdidas de deuterio)*5

Que serán las perdidas ponderadas de forma que 5 unidades de metal tengan el mismo valor que 1 unidad de deuterio.

La flota que tenga menos pérdidas, será la flota ganadora.

Todos los eventos que ocurren durante la batalla, deberán ser guardados en algún String de forma que obtengamos el siguiente formato cuando queramos ver el **reporte paso a paso** de la batalla:

```
********************CHANGE ATTACKER********************
Attacks fleet enemy: Heavy Hunter attacks Missile Launcher
Heavy Hunter generates the damage = 150
Missile Launcher stays with armor = 80

********************CHANGE ATTACKER********************
Attacks Planet: Armored Ship attacks Ligth Hunter
Armored Ship generates the damage = 700
Ligth Hunter stays with armor = -300
we eliminate Ligth Hunter
Attacks Planet: Armored Ship attacks Battle Ship
Armored Ship generates the damage = 700
Battle Ship stays with armor = 5300

********************CHANGE ATTACKER********************
Attacks fleet enemy: Armored Ship attacks Missile Launcher
Armored Ship generates the damage = 700
Missile Launcher stays with armor = -470
we eliminate Missile Launcher
```

También tendremos que devolver un reporte resumen de unidades y recursos perdidos por flota:

```
4)View Battle Reports
Option >
4
Battle Reports
Select Report Read (0 go back): (1)
Option >
1
BATTLE NUMBER: 1
BATTLE STATISTICS

Army planet              Units     Drops     Initial Army Enemy       Units     Drops

Ligth Hunter                11         8     Ligth Hunter                19        17

Heavy Hunter                 3         1     Heavy Hunter                 7         5

Battle Ship                  1         0     Battle Ship                  1         1

Armored Ship                 1         0     Armored Ship                 1         0

Missile Launcher            11         9

Ion Cannon                   1         1

Plasma Cannon                1         0


********************************************************************************
Cost Army planet                            Cost Army Enemy

Metal:        203500                        Metal:       177500
Deuterium:     28200                        Deuterium:     23300

********************************************************************************
Losses Army planet                          Losses Army Enemy

Metal:         52500                        Metal:       128500
Deuterium:       950                        Deuterium:     8100
Weighted:      57250                        Weighted:     169000
```

```
*******************************************************************************
Waste Generated:
Metal          52150
Deuterium        910

Battle Winned by Planet, We Collect Rubble


#########################################################################

View Battle development?(S\n)
```

En caso de seleccionar ver desarrollo de la batalla, mostraremos el **reporte paso a paso**.

Estos dos últimos reportes serán devueltos cuando llamemos a los métodos:

> **String getBattleReport(int battles)** → resumen, battles será el número de batallas que hayamos acumulado
> **String getBattleDevelopment()** → paso a paso.

Los siguientes métodos pueden ser útiles para el desarrollo de la batalla:

> **void initInitialArmies()** → Para inicializar el array initialArmies y poder calcular los reportes.
> **void updateResourcesLooses()** -→Para generar el array de pérdidas.
> **fleetResourceCost(ArrayList<MilitaryUnit>[] army)** → Para calcular costes de las flotas.
> **initialFleetNumber(ArrayList<MilitaryUnit>[] army)** → Para calcular el número de unidades iniciales de cada flota
> **int remainderPercentageFleet(ArrayList<MilitaryUnit>[] army)** → Para calcular los porcentajes de unidades que quedan respecto los ejércitos iniciales.
> **int getGroupDefender(ArrayList<MilitaryUnit>[] army)** → para que dado un ejército, nos devuelva el grupo defensor, 0-3 en el caso de la flota enemiga, 0-6 en el caso del ejército de nuestro planeta.
> **int getPlanetGroupAttacker(), int getEnemyGroupAttacker()** → Que nos servirán para escoger el grupo atacante tanto del planeta como de la flota enemiga.
> **void resetArmyArmor()** → que restablecerá los blindajes de nuestro ejército.

Clase Main

Desde la clase Main podremos acceder a diferentes opciones como crear unidades en nuestro planeta, ya sean defensivas como flota, mejorar nuestras tecnologías, ver los reportes de batalla de al menos las últimas 5 batallas, o en caso de haber una flota enemiga a punto de atacarnos, ver la flota enemiga.

Tendremos la opción de ver nuestros stats: flota, defensas, recursos, tecnologías …

Se controlará desde esta clase el aumento automático de recursos de nuestro planeta. Esto se podrá realizar mediante una tarea “TimerTask” que se explicará más adelante.

Se creará una flota enemiga cada 3 minutos y una batalla. Estas dos tareas también se podrán implementar mediante la clase TimerTask

En el vídeo que se facilitará se podrá ver cómo sería una versión del juego en modo consola.

Url del vídeo:
https://drive.google.com/file/d/1ShIDi9EeHXH4k2BYgPqtwgLOVrAgsLi1/view?usp=sharing

createEnemyArmy()

Para crear el ejército enemigo, dispondremos de unos recursos iniciales, que conforme vayan sucediendo batallas, serán mayores .
Iremos creando unidades enemigas aleatoriamente pero con las siguientes probabilidades:
Cazador ligero 35%, Cazador pesado 25%, Nave de Batalla 20%, acorazado 20%.
Mientras tengamos suficientes recursos para crear la unidad con menor coste, es decir, cazador ligero, iremos creando unidades aleatoriamente según las probabilidades anteriores.

ViewThread()

Este método nos mostrará que tipo de ejército nos viene a atacar:

```
NEW THREAD COMMING

Ligth Hunter                16

Heavy Hunter                12

Battle Ship                  1

Armored Ship                 1
```

La aplicación aquí descrita está basada toda en modo consola. Se deberá implementar una aplicación gráfica desde la cual podamos controlar lo mismo que desde la consola.
El diseño de la aplicación gráfica será libre.

Ayudas:

Elección grupo defensor en función de las unidades del grupo y elección del atacante en función de unas probabilidades de atacar predefinidas.:

Supongamos que tenemos un array de n posiciones, y en cada posición del array tenemos un número entero que representa la cantidad de unidades que tenemos de una categoría. Representando cada posición del array una categoria diferente a las demás. Por ejemplo:

Array = {10,35,27,70}

Y queremos escoger una categoría, pero con una probabilidad proporcional a número de unidades de dicha categoría. En nuestro ejemplo, como en la posición 3 tenemos 70 unidades y en la posición 1 tenemos la mitad, queremos que la categoria 3 tenga el doble de probabilidad de salir escogida respecto de la categoria 2.

Una posible forma de resolver este problema sería:

Calculamos la suma total de todas las categorías. SumaTotal = 10 + 35 + 27 + 70 = 152.
Escogemos un número aleatorio numAleatorio entre 1 y la suma total, 152 en nuestro caso.
Y vamos recorriendo el array sumando las cantidades de cada posición, hasta que dicha suma sea mayor que el número aleatorio.

En nuestro caso, si numAleatorio = 70.

array[0] = 10 < numAleatorio. No escogemos la categoria 0.
array[0] + array[1] = 45 < numAleatorio. No escogemos la categoria 1.
array[0] + array[1] + array[2] = 72 > numAleatorio. Escogemos la categoria 2.

Repetición de un evento cada X milisegundos:

Para ello crearemos un objeto del tipo TimerTask:

```
    TimerTask task = new TimerTask() {

        public void run()
        {



        }
        };
```

Dentro del método run implementaremos la tarea que querremos ejecutar cada cierto tiempo, en nuestro caso imprimir “Hola Mundo”

```
TimerTask task = new TimerTask() {

        public void run()
        {

         System.out.println("Hola mundo");

        }
        };
```

Ahora sólo necesitamos programar dicha tarea cada 5 segundos, para ello crearemos un objeto de la clase Timer.

```
Timer timer = new Timer();
```

Este objeto dispone del método schedule

```
public void schedule(TimerTask task,long delay, long period)
```

task es la tarea que queremos ejecutar.
delay el tiempo que queremos que pase antes de que se ejecute la primera vez en milisegundos
period es el tiempo que queremos que pase entre cada ejecución de la tarea, en milisegundos también.

Si queremos imprimir el mensaje “Hola Mundo” cada 5 segundos (5000 milisegundos) , pero que el primero aparezca a los 10 segundos (10000 milisegundos), podremos ejecutar.

```
timer.schedule(task, 10000, 5000);
```

Comprueba el resultado.

Luego crea una segunda tarea, por ejemplo:

```
TimerTask task2 = new TimerTask() {

            public void run()
            {

               System.out.println("Bienvenido");

            }
            };
```

y ejecuta:

```
    timer.schedule(task1, 10000, 5000);
    timer.schedule(task2, 8000, 3000);
```

Es decir, la primera tarea cada 10 segundos con un dejay de 5, y la segunda cada 2 segundos con un delay de 8 segundos.

De esta manera ya podemos planificar diversas tareas con tiempos diferentes con el mismo objeto de la clase Timer.

ACCESO A BASE DE DATOS:

La aplicación tendrá que guardar y recuperar los datos en una base de datos Oracle. Os pasamos un ejemplo de conexión a una base de datos.

Mejoras:
Que resetear las armaduras de nuestro ejército tenga un coste.
Que los ataques vengan de otro planeta, que produce recursos en grandes cantidades y por tanto tiene opción de crear una flota para enviárnosla a atacar.
Que los ataques sean cada cierto tiempo aleatorio.
Crear Minas que puedan alcanzar cierto nivel.
Limitar el nivel máximo de las tecnologías.

Explicaciones:
Excepciones
Clase TimerTask, Timer, Date, Timestamp.
Acceso a BBDD
system.out.printf