CS 349 Task 3: Singleton and Strategy Patterns

Description

This task demonstrates the Singleton and Strategy patterns for managing a breadth and depth of numerous representative compositional plug-and-play components dynamically. The premise is to provide a family of vehicles, each of which can select from a family of powerplants for propulsion. As in the previous tasks, the implementation details of the world model are irrelevant. In other words, we do not care how the vehicles and powerplants actually operate; their operation is rendered by simple print statements as placeholders. Rather, the creational and structural aspects of correctly and safely building and connecting the components are the focus.

The vehicles are various notional aircraft, ships, and submarines. The powerplants are various notional means of motive force that vehicles could presumably use. The legal mapping of vehicles to powerplants is as follows, where the names correspond to the Java class names to implement below:

	PowerplantGasTurbine	PowerplantElectric	PowerplantNuclear	PowerplantReciprocatingDiesel	PowerplantReciprocatingGas	PowerplantTurbofan	PowerplantTurbojet	PowerplantTurboshaft
AircraftCommercialAirliner						■		■
AircraftFighterJet						■	■	
AircraftHelicopter					■			■
AircraftTrainer				■	■			
ShipAircraftCarrier			■					
ShipDestroyer	■							
ShipSpeedboat					■			■
ShipYacht				■	■			
SubmarineUBoot		■		■				
SubmarineVirginiaClass			■					

You will not enforce these constraints, but you should be thinking about how you would do so based on what you learned in Tasks 1 and 2.

Requirements

Write the appropriate Java classes such that the following requirements are satisfied.

For all vehicles:

1. A constructor that takes an arbitrary string identifier and a powerplant.

   It stores a salted version of this identifier, which is generated by the Singleton class described below.

   It informs the powerplant that it is the host of the powerplant. See the description of powerplants below.

2. A `getID()` method that returns the identifier given in (1).
3. A `getIDSalted()` method that returns the salted identifier generated in (1).
4. A `getPowerplant()` method that returns the powerplant given in (1) or (6), if there is one; otherwise, throw a `RuntimeException` with the message "no installed powerplant".
5. A `removePowerplant()` method that removes the powerplant given in (1) or (6) from service, if there is one; otherwise, throw a `RuntimeException` with the message "no installed powerplant". Be sure to keep the ownership relationships consistent.

6. An `installPowerplant()` method takes a powerplant and puts it in service. If there is already one, throw a `RuntimeException` with the message "powerplant *powerplant* already installed", where *powerplant* is the identifier of the one in service.
7. A `hasPowerplant()` method that returns whether there is a powerplant in service.
8. A `move()` method that prints to standard output the result of calling `generate()` on the powerplant in service. The form is "*idSalted*`:` *result*", where *idSalted* is available from (3) and *result* is the result of the call to `generate()`. If there is no powerplant in service, *result* should be "no powerplant"

For `SubmarineUBoot`:

1. A constructor that takes an arbitrary string identifier and primary and secondary powerplants, which must not be the same powerplant object. Throw a `RuntimeException` to this effect.

   The salt processing is the same as for all vehicles above.

   The primary powerplant is the same as the powerplant above. It is in operation by default; see (4).
2. A `getPowerplantPrimary()` method that applies to the primary powerplant in the same way `getPowerplant()` does for vehicles in general.
3. A `getPowerplantSecondary()` that applies to the secondary powerplant in the same way `getPowerplant()` does for vehicles in general.
4. An `isPrimaryOrSecondary()` method that takes a boolean indicating whether the primary or secondary powerplant is in operation, respectively.
5. An `isPrimaryOrSecondary()` method that returns the state selected in (4).
6. A `removePowerplantPrimary()` method that applies to the primary powerplant in the same way `removePowerplant()` does for vehicles in general.
7. A `removePowerplantSecondary()` method that applies to the secondary powerplant in the same way `removePowerplant()` does for vehicles in general.
8. A `hasPowerplantPrimary()` method that applies to the primary powerplant in the same way `hasPowerplant()` does for vehicles in general.
9. A `hasPowerplantSecondary()` method that applies to the secondary powerplant in the same way `hasPowerplant()` does for vehicles in general.
10. An `installPowerplantPrimary()` method takes a powerplant and puts it in service as the primary in the same way `installPowerplant()` does for vehicles in general.
11. An `installPowerplantSecondary()` method takes a powerplant and puts it in service as the secondary in the same way `installPowerplant()` does for vehicles in general.
12. An `move()` method that applies to the primary or secondary powerplant as appropriate in the same way `move()` does for vehicles in general.

For the powerplants:

1. A constructor that takes an arbitrary string identifier. There is no salt processing.
2. A `getID()` method that returns the identifier given in (1).
3. A `setHost()` method that takes a vehicle and records it as its host. If the host has already been set, throw a `RuntimeException` that indicates who the current host is.

4. A `getHost()` method that returns the host given in (3). If the host has not been set, throw a `RuntimeException` to this effect.
5. A `removeHost()` method that removes the host given in (3). If the host has not been set, throw a `RuntimeException` to this effect.
6. A `hasHost()` method that returns a boolean indicating whether the host has been set in (3).
7. A `generate()` method that returns a string based on the type of powerplant as follows. Remember, the English matters; any deviation is incorrect.

`PowerplantElectric`	"generating electricity"
`PowerplantGasTurbine`	"spinning a gas turbine"
`PowerplantNuclear`	"splitting atoms"
`PowerplantReciprocatingDiesel`	"chattering away"
`PowerplantReciprocatingGas`	"vroom vroom!"
`PowerplantTurbofan`	"bypassing lots of air"
`PowerplantTurbojet`	"exhausting a jet"
`PowerplantTurboshaft`	"spinning a shaft"

The singleton aspect is captured by class `IdentifierSaltManager`:

1. A `getInstance()` method that returns the singleton instance.
2. A `getIDSalted()` method that takes a string identifier and returns a string identifier with an appended running salt value in the form "*identifier*`#`*salt*", where *identifier* is the given identifier and *salt* is an integer that starts at 1 and increments after every call to this method.
3. A `getSaltNext()` method that returns the salt value to be assigned upon the next call to (2).

Deliverables

Submit all your source files in a zip file.

It is not necessary to comment your code. Do not package your code. Include additional error checking where appropriate. You may add any code to facilitate your design, but you must not deviate from the specifications above. Code that does not compile will not be graded.