War Games
SENG301

William Scott | 11876177

A note on finding changes within the codebase

Almost all changes in the codebase have been indicated through the use of hashtags within Javadoc and comments. For example any function which has been changed will feature Javadoc above it with "#taskX" (where X is the number e.g. #task1, #task4), or a line of code will have a comment above or next to it with a small explanation followed by "#taskX".

Task 1: Vehicle.setWeapon()

Gang of Four Pattern: Factory Method	War Games
Creator	Vehicle
ConcreteCreator	AttackHelicopter, FighterJet, Tank, InfantryMobilityVehicle, Submarine, Warship
ConcreteProduct	GuidedMissileSystem, GrenadeLauncher, Cannon, MachineGun, RocketLauncher, TorpedoTube
Product	WeaponSystem
factoryMethod()	makeVehicleWeapon()
doSomething()	setWeapon()
ConcreteCreator -> Creator	- AttackHelicopter (extends)-> Vehicle - FighterJet (extends)-> Vehicle - Tank (extends)-> Vehicle - InfantryMobilityVehicle (extends)-> Vehicle - Submarine (extends)-> Vehicle - Warship (extends)-> Vehicle
ConcreteCreator (dependency)-> Concrete Product	- AttackHelicopter -> GrenadeLauncher - FighterJet -> GuidedMissileSystem - Tank -> Cannon - InfantryMobilityVehicle -> MachineGun - Submarine -> TorpedoTube - Warship -> RocketLauncher
ConcreteProduct -> Product	- GrenadeLauncher (extends)-> WeaponSystem - GuidedMissileSystem (extends)-> WeaponSystem - Cannon (extends)-> WeaponSystem - MachineGun (extends)-> WeaponSystem - TorpedoTube (extends)-> WeaponSystem - RocketLauncher (extends)-> WeaponSystem

A vehicle can be many different things, for example it could be a fighter jet, a tank, a submarine, or any other class which extends from it. All of these children of Vehicle have different weapons that they want to set. How the fighter jet will attack is different from the tank, and so on. Romeo originally wrote the code such that the parent, `Vehicle` is checking what type of vehicle it is, going through and seeing if the vehicle's name matches one of its children and if it does then setting the weapon accordingly. The parent class `Vehicle` now must know about the children (`AttackHelicopter`, `Tank`, `Warship`, etc.) which it really shouldn't, resulting in coupling between these classes. If we want to add another child, we now must go back and modify code up in the parent; this is a huge violation of the open/closed principle, it's not open for extension and is not closed for modification.

To improve the design of Romeo's code I implemented the factory method pattern. This was accomplished through the introduction of the factory method `makeVehicleWeapon()`, which was added to `Vehicle` and its children. Instead of `setWeapon()` checking for the vehicle's name, it simply sets the weapon returned by our factory method which it makes a call to. After adding the abstract method `makeVehicleWeapon()`, I went into all of the concrete creators (e.g. `AttackHelicopter`, `Tank`, etc.) and added the method with the desired weapon system they should equip. I found the factory method design pattern easy to implement, and think someone coming on to the project would find it easy to understand if they wished to add new vehicles. Confusion prevented from having to know you must change the parent class is definitely good, and overall I think this design pattern would be a good improvement for Romeo to bring into his codebase.

Task 2: The Vehicle hierarchy

To begin my assessment of the Vehicle hierarchy I reviewed the 'Common Inheritance Mistakes' as learned in class. Starting with 'Inheritance for implementation' I looked to make sure all of the functions within the parent classes were in-fact being used and not stubbed out, and that all the function inherited did in fact make sense for use in the sub-class. I concluded that all of the vehicles which Romeo implemented were capable of fighting and movement, and all of the sub-classes from vehicle had implementations for the abstract functions of `moveToLocation()` and `completeAttack()`.

While researching and looking into stories of bad inheritance I read about a humorous inheritance error which occurred within a war simulation much like ours, except set in Australia[1] [2].. Within their simulator they had kangaroos, and these kangaroos needed the behaviour to scatter when helicopters flew overhead. As a quick fix the kangaroos were hastily added as a subclass of the soldiers which already had the behaviour to scatter when helicopters flew overhead. This worked well and the kangaroos scattered away as the helicopters flew overhead, that was until the kangaroos then remerged firing missiles back in retaliation to the advancing helicopters. Reading about an error like this made me really check that all the methods inherited were appropriate for

[1] http://www.snopes.com/humor/nonsense/kangaroo.asp
[2] http://www.cs.utexas.edu/~eberlein/cs305j/InheritanceInterfaces.html

each of the subclasses. Upon inspection I found there to be no weird behaviours such as ships going on land, or jets traveling by sea.

Next up I checked that “Is a-role-of” and “becomes” were not violated. To check for that I looked to the design to make sure there would not be cases where a vehicle would want be two different classes (violating multiple inheritance) or any case where a vehicle would become another vehicle. As the design currently stands with the functionality expected, the inheritance in place should work well and not violate either “is a role of” or “becomes”. However, if Romeo desired a vehicle which could travel by land or sea requiring both `ArmouredPersonnelCarrier` and `NavalVessel` there may be a problem. If this is something he anticipates needing I think using interfaces rather than inheritance may be more desirable as he can require the unique naval vessel or land vehicle methods at that time. Moffat gave us the wonderful quote, “If it can change, it ain’t inheritance” so with that as a starting point I looked at all the classes within the Vehicle package to see if there was no potential for change. With how the design is currently made I see no opportunity or reason for one vehicle to change into another. Attack helicopters will not become fighter jets, etc. This is good design.

As for over-specialisation this is not much actual functionality to look at, and it is primarily use of strings and integers, however the `WeaponSystem` for all `Vehicles` is stored as a `WeaponSystem` and not as a concrete class. `Tanks` for example use polymorphism to contain their `Cannon` within the `WeaponSystem`; again this is good design as it appears vehicle’s always use the highest level class possible.

Next I ensured there were no violations of the Liskov Substitute Principle (LSP). In all the children of `Vehicle`, I checked that all method signatures were the same, that nothing was being stubbed out, and that polymorphism could still be used to make collections of these children and behave as expected. After this investigation I conclude there is no violation of the LSP, for example anywhere you would use a tank you could also use a vehicle. The code will work without having to know what the actual class of the `Vehicle` is.

With ‘Design By Contract’ I do believe we run into one small problem and that is with moving and attacking with a `NavalVessel`. The functions `moveToLocation`, `fight`, and `completeAttack` all have tightened pre-conditions. The function `moveToLocation` requires that the location contain the name “coast”, if we told the NavalVessel to go to say “Enemy Base” and we thought this was true but the Naval Vessel was not actually there this would cause issues. Similarly `fight` and `completeAttack` require that a new boolean `movedToLocation` is true. Again this is a tightening of the precondition when compared to the function in vehicle. I also believe this Boolean `movedToLocation` could be moved up into the vehicle class as all vehicles could and probably should make use of it. Perhaps all of the moving and fighting across all vehicles could return a Boolean to better indicate if the desired action was successful.

Aside from this small issue within `NavalVessels`, and with the implementation of the factory method pattern for the equipping of weapons I feel that the inheritance within the Vehicle family is overall pretty good.

Task 3: Identify GoF patterns

I have identified the use of the Strategy pattern within weaponry, and the Template Method pattern within vehicles.

Gang of Four Pattern: Strategy	War Games
Context	Vehicle and its children
Strategy	WeaponSystem
ConcreteStrategy	Cannon, GrenadeLauncher, GuidedMissileSystem, MachineGun, RocketLauncher, TorpedoTube
Algorithm	Load(), Fire()
CallAlgorithm	Fight()
Context -> Strategy	Vehicle and its children (aggregates)-> WeaponSystem
Concrete Context -> Concrete Strategy	- AttackHelicopter -> GrenadeLauncher - FighterJet -> GuidedMissileSystem - Tank -> Cannon InfantryMobilityVehicle -> MachineGun Submarine -> TorpedoTube Warship -> RocketLauncher

Gang of Four Pattern: Template Method	War Games
AbstractClass	Vehicle
ConcreteClass	AttackHelicopter, FighterJet, Tank, InfantryMobilityVehicle, Submarine, Warship
templateMethod()	moveToLocation(), fight(), completeAttack(),makeVehicleWeapon()
ConcreteClass (extends)-> AbstractClass	- Submarine -> NavalVessel -> Vehicle - Warship -> NavalVessel -> Vehicle - Tank -> ArmouredPersonnelCarrier -> Vehicle - InfantryMobilityVehicle -> ArmouredPersonnelCarrier -> Vehicle - FighterJet -> CombatAircraft -> Vehicle - AttackHelicopter -> CombatAircraft -> Vehicle

Within my code base I also add the templateMethod `makeVehicleWeapon()` which is added by all of the concrete children such as `Tank`, `Submarine`, etc. Also within the case of the template pattern the classes which are implementing the template method (`NavalVessel`, `CombatAircraft`, `ArmouredPersonnelCarrier`) are abstract themselves and the concrete children inherit from this.

Task 4: Only one CommandCentre

As there will only ever be one `CommandCentre` in the game, I implemented the singleton design pattern to remove the ability to have multiple instances of the CommandCentre from existing. I began by adding a single private static variable to contain the unique instance of the `CommandCentre`. This can be found in `CommandCentre` and is named `uniqueInstance`. I then changed the constructor from public to private to ensure that only CommandCentre itself can make a call to the constructor. Finally to complete the implementation of the Singleton pattern I added a getter, `getInstance()` which is static allowing the class never to be instaniated. I went with the "Lazy initialization approach" as taught in class which checks if the instance is null, and adding one if it is; otherwise it returns the existing instance.

Gang of Four Pattern: Singleton	War Games
-$ uniqueInstance	private static CommandCentre uniqueInstance
+$ instance()	public static CommandCentre getInstance()
- Singleton()	private CommandCentre()

Task 5: A Military Unit

Gang of Four Pattern: Composite	War Games
Component	VehicleComponent
Leaf	Vehicle
Composite	VehicleUnit
doSomething()	Vehicle classes methods; attack, moveToLocation, toString
add(Component)	add(VehicleComponent vehicle)
remove(Component)	remove(VehicleComponent vehicle)
getChild(int)	getComponent(int componentIndex)
Composite (aggregation)-> Component	VehicleUnit -> VehicleComponent
Leaf (extends)-> Component	Vehicle -> VehicleComponent
Composite (extends)-> Component	VehicleUnit -> VehicleComponent

Romeo had not yet implemented units, however after reading the requirements that they should be able to do everything other vehicles do such as attack or move, and that units should be able to make up other units I decided to implement the GoF pattern 'Composite' which I think lends itself well to these requirements. I did briefly consider the observer design pattern and having a subject unit to broadcast changes, but felt this didn't really lend itself to forming a containment hierarchy. To achieve the implementation of the composite pattern I created a new package within vehicles called 'unit' which contains the two classes I added to make units possible (`VehicleComponent`, `VehicleUnit`) and modified `Vehicle` to become the leaf in the pattern by extending it from the `VehicleComponent`. The functions `add` and `remove` of course do not make sense in the leaf node, but as taught in class this is balance of forces we must deal with so that vehicles can be treated with transparency. Overall I think the composite pattern works well and the use of the composite being treated the same as a leaf, and allowing for units of units as described outweighs safety lost from the use of the pattern.

Task 6: Super Armoured Personnel Carrier

Gang of Four Pattern: Composite	War Games
Component	ArmourComponent
Leaf	ArmourPlate, ArmourBulletproofglass
Composite	ArmourGroup
doSomething()	defend()
add(Component)	add(ArmourComponent armour)
remove(Component)	remove(ArmourComponent armour)
getChild(int)	getComponent(int componentIndex)
Composite (aggregation)-> Component	ArmourGroup -> ArmourComponent
Leaf (extends)-> Component	ArmourPlate -> ArmourComponent ArmourBulletproofglass -> ArmourComponent
Composite (extends)-> Component	ArmourGroup -> ArmourComponent

To allow for Armoured Personnel Carriers to be fitted with armour to provide extra protection and to allow it to be added in several layers while still allowing behaviour to continue as prior to armour plating, I made use of the composite design pattern. This was done through the classes `ArmourComponent`, `ArmourGroup`, `ArmourPlate`, and an extra class to ensure my armour components were working as expected, `ArmourBulletproofGlass`; I also modified `ArmouredPersonnelCarrier` to contain an instance of `ArmourGroup` which can be added to, like in layers. I considered use of the decorator pattern alone, and use of the decorator pattern in combination with the composite pattern, but ultimately decided I would be using extra design patterns for the sake of using extra design patterns. I felt all armoured personnel carriers should be able to defend, and that attaching additional armor always made sense. Moreover if we really wanted to indicate one as super we could simply check if armour was attached; and if they needed to be upgraded to super to allow for that first, a function and boolean could easily be added to allow for the upgrade and the indication of its “super” status. Wrapping APCs in decorators and decorators to provide layers of armour seemed like overkill as those decorators themselves would not really be adding any additional functionality to the vehicle. One of the weaknesses I have identified in my implementation is possible armour incompatibility is currently unhandled; perhaps we want tanks to only be equipped with at most two pieces of side armour, as it currently stands more than that could be added. If pieces of armour were made tank specific instead of infantry mobile vehicle specific both of these armour pieces could be added without issue. Handling for these cases could of course be added but as it was not requested I left my implementation of the design pattern simpler than not. What the function I added `defend()` currently does within armour is not entirely clear, and differences between what defend does could cause issues or contradictions. As well the `defend()` function in the `ArmourGroup` class currently iterates through all of the child armour it contains in order calling defend on them. If the order armour protected was important, or certain layers should not begin defending until exposed by exterior layers, functionality would have to be implemented for that. Overall, I think the composite design pattern is a good choice for adding the functionality requested of layering armour in the task and could be expanded with further to allow for desired behaviour in how specifically armour should work within the war game.