SENG301 ASSIGNMENT2

Yuan Cui

Ycu20

63483319

Task1:

(1) Set variables spinningDegree and spinningDirection for choosing speed and direction. Make spinning by makeSpinner() function. If you'd like to make two signs spinning with different speed and direction. Variables LeftSpinningDegree, LeftSpinningDirection, RightSpinningDegree and RightSpinningDirection should be set separately.
(2) Create a new CompoundSign object and add two spinning signs and one base sign into it. Call setLocation() function to set location for compound sign. Variables as movingSignLoc_x and movingSignLoc_y are used for defining the location.
(3) Variables movePixelRight and movePixelDown are used for setting the speed of moving sign move right and down separately. Function makeMovingSign is used for making a moving sign.
(4) Add moving sign to signs collection.

Task2:

(1) Change type of variable args from string array to hashset<String>. Hashset can help eliminate duplicated elements.
(2) Because elements in hash set are unordered, tick() method will fetch sign from signs randomly. One solution is change the type of signs from HashSet to LinkedHashSet. Another way is use the TreeSet type. But comparator is needed for decide the order of signs. First implementation is used for this task.

Task3:

Decorator Pattern

Component	Sign
ConcreteComponent	PlainSign
Decorator	DynamicSign
doSomething()	tick()
ConcreteDecoratorA	AnimatedSign
addedState	None
addedBehavior()	advanceImage()
ConcreteDecoratorB	MovingSign
addedState	bounds (if reach bounds reset location to original)
addedBehavior()	move() / checkBounds() (triggered when reach boundary)
ConcreteDecoratorC	SpinningSign
addedState	deltaDegrees (change the degree of rotation)
addedBehavior()	turn()
ConcreteComponent->Component	PlainSign->Sign

Decorator->Component	DynamicSign->Sign
ConcreteDecoratorA->Decorator	AnimatedSign->DynamicSign
ConcreteDecoratorB->Decorator	MovingSign->DynamicSign
ConcreteDecoratorC->Decorator	SpinningSign->DynamicSign

Note: Just consider the decorator pattern excluding composite pattern

Composite Pattern

Client	FunkySignsApp
Component	Sign
Leaf	PlainSign
Composite	CompoundSign
doSomething()	tick()
add(Component)	addSign(Sign)
remove(Component)	missing
getChild(int)	missing
Client->Component	FunckySignsApp->Sign
Leaf->Component	PlainSign->Sign
Composite->Component	CompoundSign->Sign

Note: Just consider the composite pattern excluding decorator pattern

Observer Pattern

Subject	Spy
ConcreteSubject	PlainSign
Observer	Updater
ConcreteObserver	FunkySignsView
attach(Observer)	missing
detach(Observer)	missing
notify()	update() in Spy class
doSometing()	update() in PlainSign class
getterA()	getLocation()
getterB()	getIcon()
getterC()	getRotation()
update()	update() in Updater class
Subject->Observer	Spy->Updater
ConcreteSubject->Subject	Sign->Spy
ConcreteObserver->Observer	FunkySignsView->Updater
ConcreteObserver->ConcreteSubject	FunkySignsView->PlainSign

Task4:

(1) Add or delete updaters method missing.
Solution: add functions as below.
public void addUpdater(Updater u) //add an update to the list
public void deleteUpdater(Updater u) //delete an update from the list
public void deleteUpdaters() //delete the updaters list, it no longer has any updaters

(2) The getUpdaters() function is meaningless, because we just need to add/delete them. Do not need to get them. If get updaters function is necesarry, its type must be change to unmodifiable set.
Solution: getUpdaters() can be changed to countUpdaters(), which will return the number of updaters. Or return Collections.unmodifiableSet(updates).

(3) The updaters type should not be ArrayList, because it will be a risk to add two same updaters into the list.
Solution: Use Set instead of List will be a good solution.
(4) It's better to let Spy class extends Object class. In java documentation java.util.observable extends Object, but it is not necessary.
(5) It's better to ensure when the object changed, all updaters can be notified. And if without change, updaters will not be notified.
Solution: add functions as below.
hasChanged(), setChanged(), clearChanged()
If object changed, setChanged() -> if (hasChanged) -> update() -> clearChanged()

Task5:

Spinning Sign spins the sign by calling function turn(). Howerver, the function setRotation() in function turn() does not do anything. Because setRotation function in CompoundSign is empty. Thus, a CompoundSign wrapped by SpinningSign will not spin. Similarly, vegasSign is an object of CompoundSign. Thus, it will not spin by calling makeSpinner() method.

Design by Contract

Sign
-Pre-condition:
Property rotation is existed, and its type is int.
Variable degrees is given, and its type is int.

-Post-condition:
Sign rotates after calling setRotation() method.

PlainSign
-Pre-condition:
Property rotation is existed, and its type is int.
Variable degrees is given, and its type is int.

It’s Ok.

-Post-condition:
Sign rotates after calling setRotation() method and its angle is between 0 and 360.
Update is done.

Tighten, it’s OK

Class is OK

CompoundSign
-Pre-condition:
Property rotation is existed, and its type is int.
Variable rot is given, and its type is int.

It’s OK

Post-condition:
Sign will not rotate after calling setRotation() method.

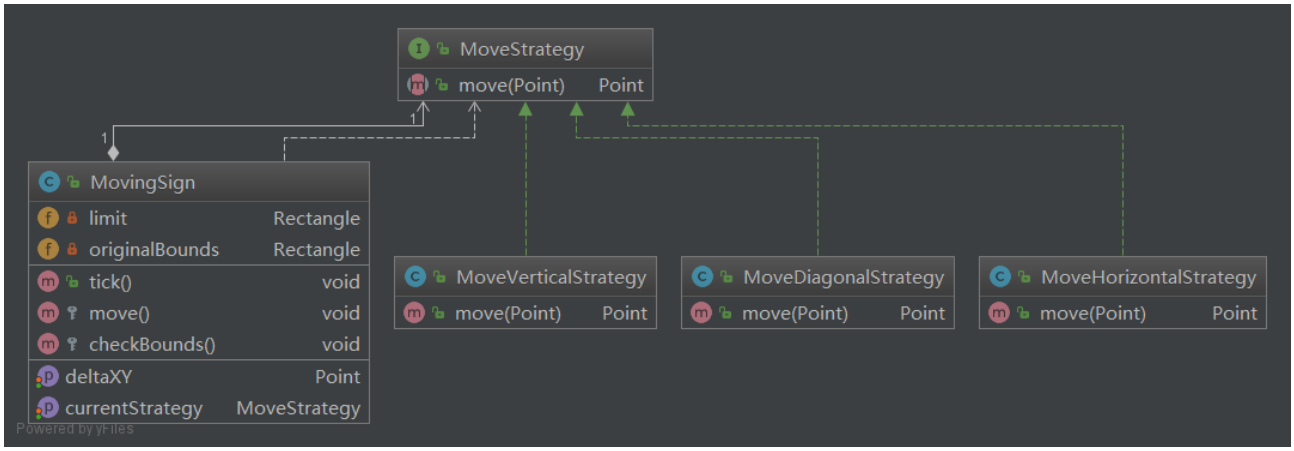
Loosen. It's not OK.

Class is not OK.

Note: About dynamic sign, in its setRotation function, an attribute called baseSign will execute setRotation() method. So, instead of rotating dynamicSign, just baseSign can rotate. As the analysis shows above, if baseSign instanceOf PlainSign, it will rotate. Otherwise, if it instanceOf CompoundSign, it will not rotate.

Task6:

In my solution, strategy pattern was used. That's because strategy pattern can change an object's algorithm dynamically, rather than through inheritance. The structure of strategy pattern can be found from UML below.

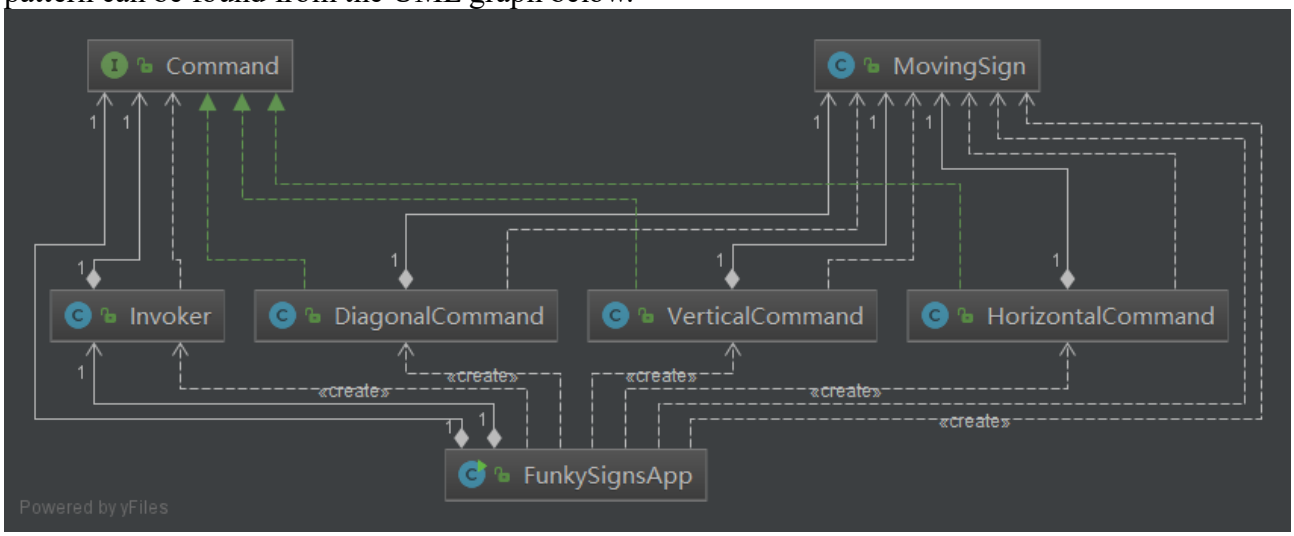


Class MoveStrategy was an interface which owns three implementations, MoveVerticalStrategy, MoveDiagonalStrategy, MoveHorizontalStrategy separately. Class MovingSign was the contexts which knew different strategies existed. Another reason for me to use strategy pattern was that AWT (& Swing) LayoutManager is a Strategy, which means it is powerful to use strategy pattern in UI design.
MoveStrategy class owned a function move() which needed a Point type attribute from MovingSign, and would return a Point type value. MovingSign class would use this value to change its moving style. A property currentStrategy was added into MovingSign, which would record the current strategy of moving sign. setStrategy() can change the currentStrategy value and change the moving strategy as well.

Strategy Pattern

Context	MovingSign
Strategy	MoveStrategy
ConcreteStrategyA	MoveVerticalStrategy
ConcreteStrategyB	MoveDiagonalStrategy
ConcreteStrategyC	MoveHorizontalStrategy
Algorithm()	move(Point)
Strategy->Context	MoveStrategy->MovingSign
ConcreteStrategyA->Strategy	MoveVerticalStrategy->MoveStrategy
ConcreteStrategyB->Strategy	MoveDiagonalStrategy->MoveStrategy
ConcreteStrategyC->Strategy	MoveHorizontalStrategy->MoveStrategy

Because clients want to change different moving style at any time, there should not be MovingSign object to call setStrategy() method. Because movingSign is the context in Strategy pattern not the real client. For decoupling the MovingSign class and FunkySignsApp, a command pattern is added. Command pattern is good to parameterize clients with different requests. The structure of command pattern can be found from the UML graph below.



Three different concrete commands implement the interface Command. Receiver is MovingSign. In FunkySignsApp, a new function called buildCommands is established for building commands for specific sign. In other word, any movingSign can build command by this method. Finally, the moving style can be adjusted in the main() function of FunkySignsApp (Line 72). A while loop and sleep() method are used for change moving style every 3 seconds. Thus, it implements the requirement of **dynamically change moving style of any MovingSign at any time**. More details can be checked from my codebase.

Command pattern

Client	FunkySignsApp
Recever	MovingSign
action()	setCurrentStrategy
Invoker	Invoker
Command	Command
ConcreteCommand	VerticalCommand/HorizontalCommand/DiagonalCommand
execute()	execute()
unexecute()	N/A
Client->Receiver	FunkySignsApp->MovingSign
Client->ConcreteCommand	FunkySignsApp->VerticalCommand/ FunkySignsApp->HorizontalCommand/ FunkySignsApp->DiagonalCommand
ConcreteCommand->Command	VerticalCommand->Command/ HorizontalCommand->Command/ DiagonalCommand->Command
Command->Invoker	Command->Invoker
ConcreteCommand->Receiver	VerticalCommand->MovingSign/ HorizontalCommand-> MovingSign / DiagonalCommand-> MovingSign