Introduction :

Cet exercice introduit le principe du balisage dans le domaine maritime, qui dans ce cas désigne un ensemble de balises qui se placent dans la mer avec des mouvements aléatoires dans le but d'effectuer des mesures qui servira à l'étude du milieu marin.

Les balises ont deux fonctionnalités principales :

- Effectuer des mesures
- Surveiller des satellites

Les balises ont une capacité de mémoire assez limitée, on fait intervenir des satellites qui sont en charge de récupérer les données capturées par les satellites et donc réinitialiser leur mémoire.

Il est demandé de simuler un système de déplacement aléatoire et d'échange d'informations entre deux acteurs principaux : satellites et balises

Les besoins :

Les satellites effectuent des mouvements de déplacement horizontal dans l'atmosphère.

Les balises effectuent des mouvements de déplacement horizontal, vertical et sinusoïdal dans l'atmosphère.

Les balises doivent remonter à la surface de l'eau, une fois la mémoire pleine.

Les balises doivent implémenter un système de détection de satellite.

Pour se synchroniser avec un satellite, celui-ci doit remplir les deux conditions : être dans le périmètre de la balise, et être disponible.

Un satellite disponible est un satellite qui n'est synchronisé avec aucune balise à un instant T.

Avant de lancer une balise dans l'océan, on doit fixer son paramètre de mode de déplacement.

La solution

Les design patterns proposés :

Strategy :

Principe :

Le patron stratégie consiste à définir une famille d'algorithmes, **encapsuler** chacun d'eux et les rendre interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

Structure :

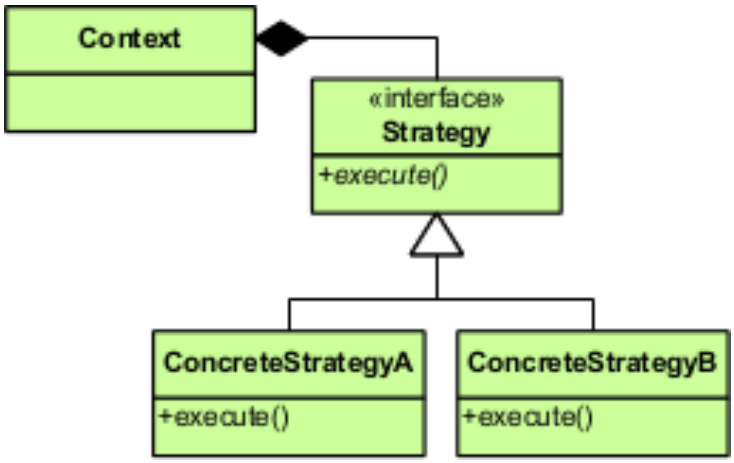


Diagramme UML : Strategy

Dans le cas de cet exercice :

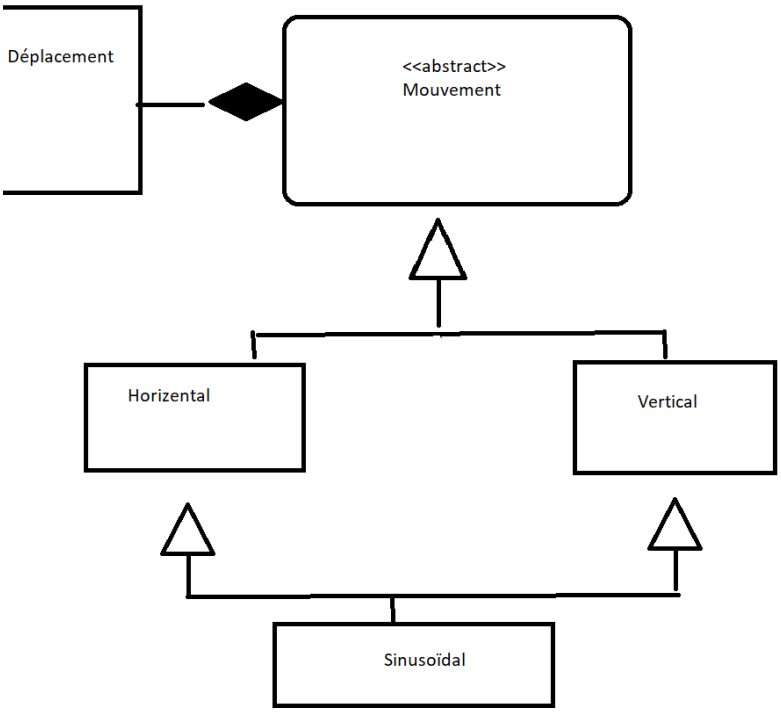
L'intérêt est d'avoir un code évolutif sans modification de la classe qui met en œuvre le choix effectué.

Pour l'implémentation, on définit une class abstraite qui représente les différentes stratégies possibles pour un mouvement de balise.

On définit ensuite des sous classes, chacune représente un cas de mouvement possible, et pour ce faire, elle met en œuvre une fonction membre pour effectuer l'action du déplacement.

L'exécution de l'action du mouvement est déléguée à la stratégie « Mouvement » qui décide de la nature du mouvement de la balise sans définir à paramètre particulier correspondant à la nature du mouvement.

Le mouvement est indépendant de la nature de l'objet, un satellite peut avoir des mouvements pareils que ceux qu'on propose aux balises.


Déplacement
<<abstract>>
Mouvement
Horizental
Vertical
Sinusoïdal

Observer :

Principe :

L'observateur définit une relation un-à-plusieurs de sorte que lorsqu'un objet change d'état, les autres sont notifiés et mis à jour automatiquement.

Structure :

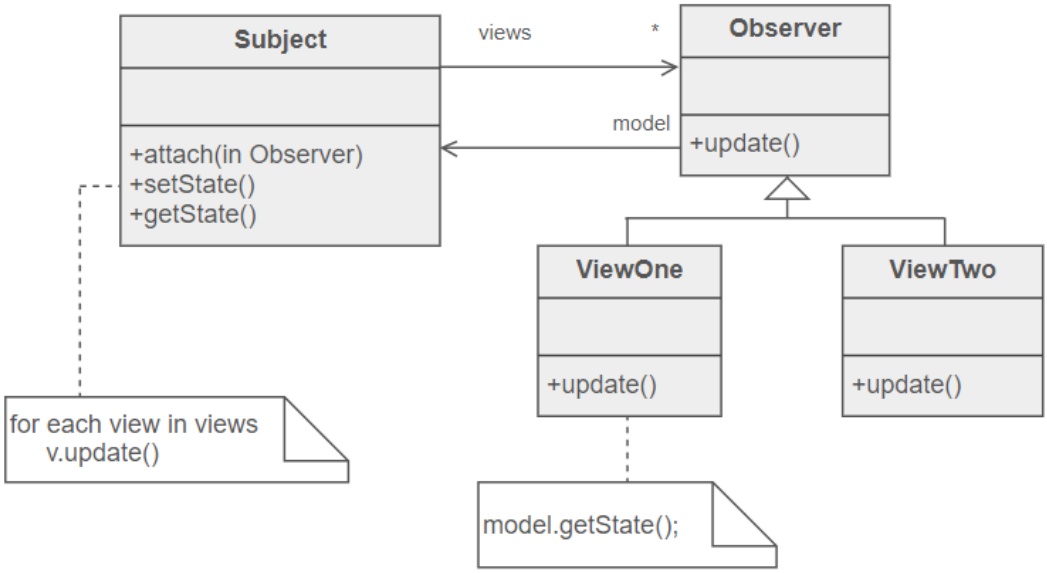


Diagramme UML : Observer

Dans le cas de cet exercice :

Dans le cadre de cet exercice, on s'intéresse à déclencher une action suite à un évènement.

L'évènement est la synchronisation entre une balise est un satellite, l'action est d'envoyer un message de notification à la balise afin qu'elle commence à transmettre les données au satellite.

Un bon patron de concéption pour créer ce type d'interactions celui du modèle « Observable/Observeur ».

Chaque satellite observable contient une liste de satellites observateurs, ainsi à l'aide d'une méthode de notification l'ensemble des balises est prévenu quand un satellite est dans le périmètre, et aussi disponible.

Announcers :

Définition :

Les annonces sont un nouveau cadre pour la notification d'événements qui a été porté sur plusieurs implémentations de Smalltalk. Les annonces fournissent une implémentation simple et générique du modèle Observer.

Dans le cas de cet exercice :

Announcements offre une solution entièrement orientée objet où les événements sont des objets simples. De tels événements en tant qu'objets peuvent être entièrement personnalisés avec des données contextuelles spécifiques à l'application. Un ChangeEventobjet peut contenir des informations telles que le modèle d'origine, la nature du changement ou le déclencheur du changement.

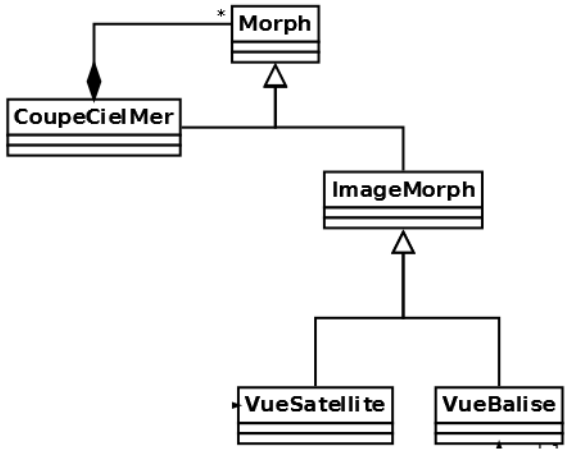
Dans le cadre de cet exercice, on a besoin de notifier les balises de la position des satellites, mais pas que, il faut aussi annoncer l'état du satellite (disponible ou non disponible), de ce fait, l'utilisation des annonceurs répond au besoin.

Comment ça marche ?

A travers des séquences d'écoute, un announcer envoie une notification une fois le satellite dans le périmètre, et se remet à écouter une fois le message envoyé.

L’implémentation de la solution :

La partie graphique :

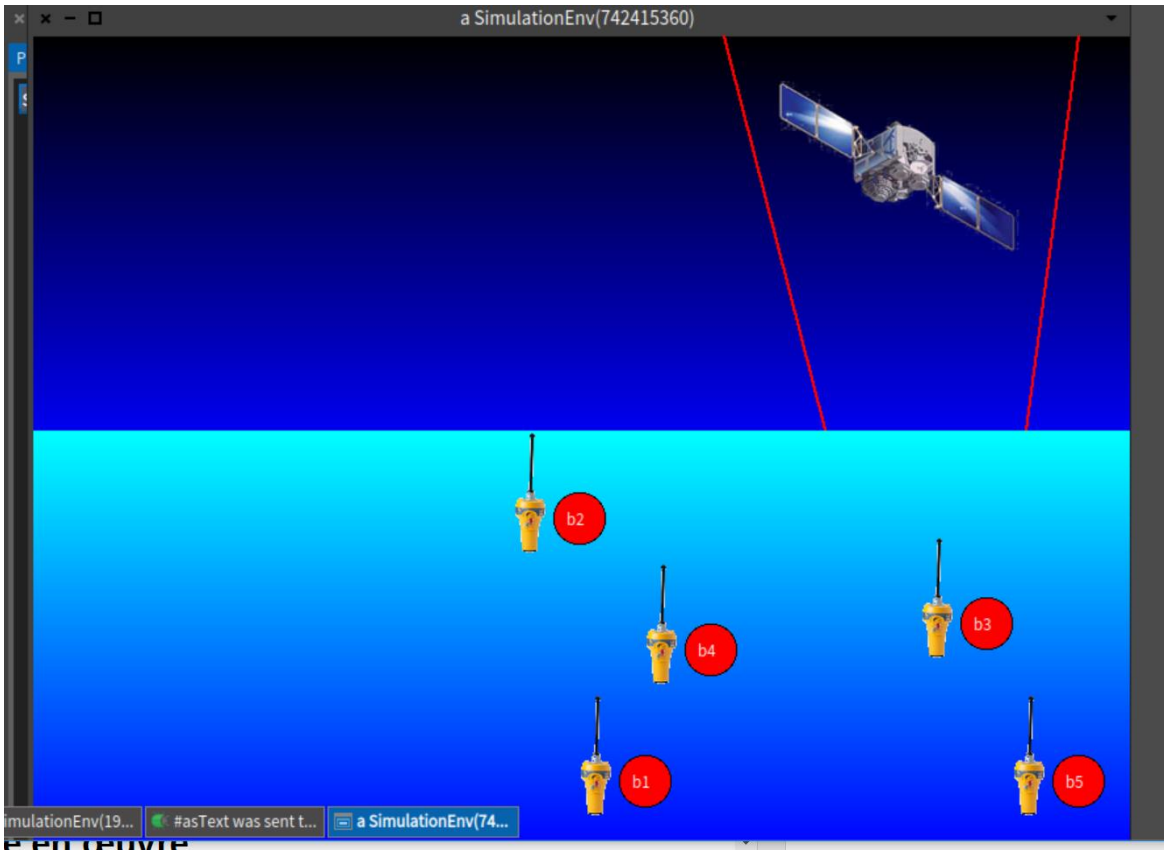


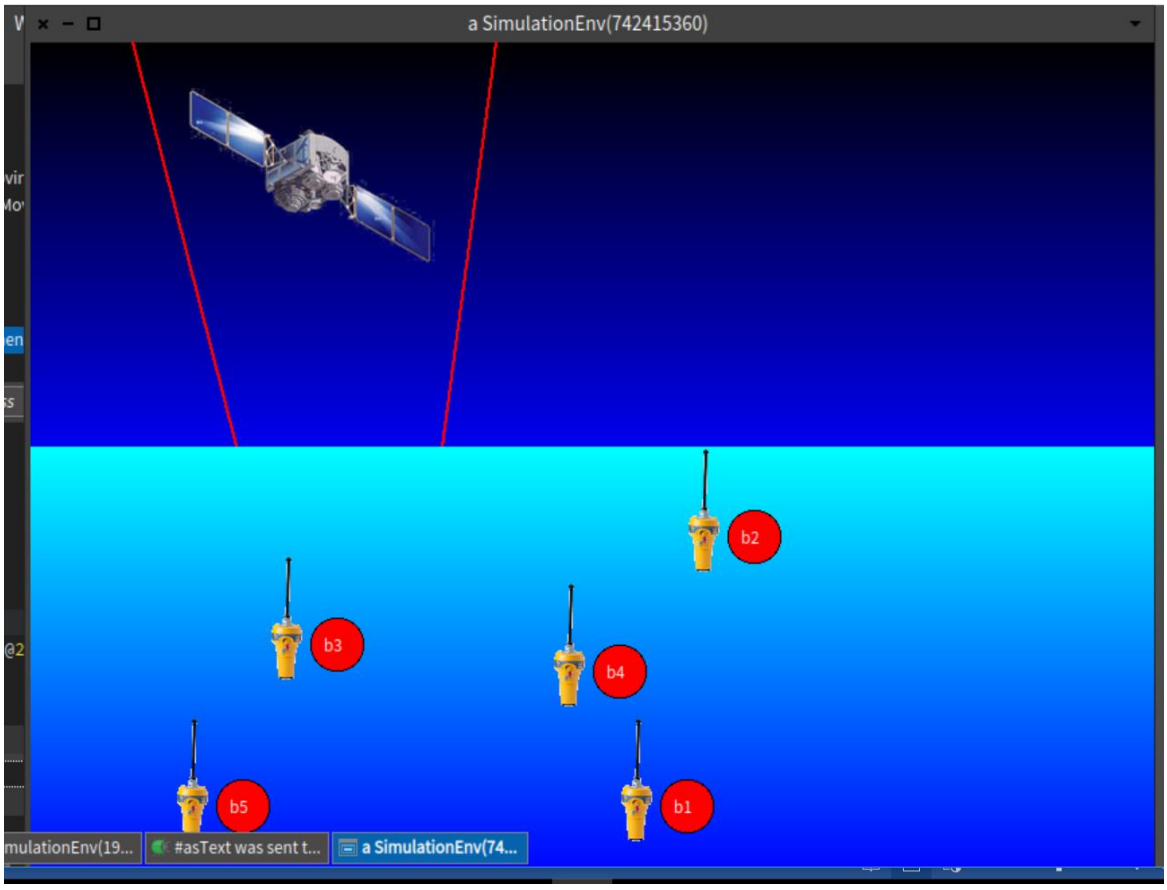
CoupeCielMer est l’environnement sur lequel on va pouvoir visualiser une instance de la coupe coupe ciel et une autre de la coupe mer. Elle hérite de la classe Morph.

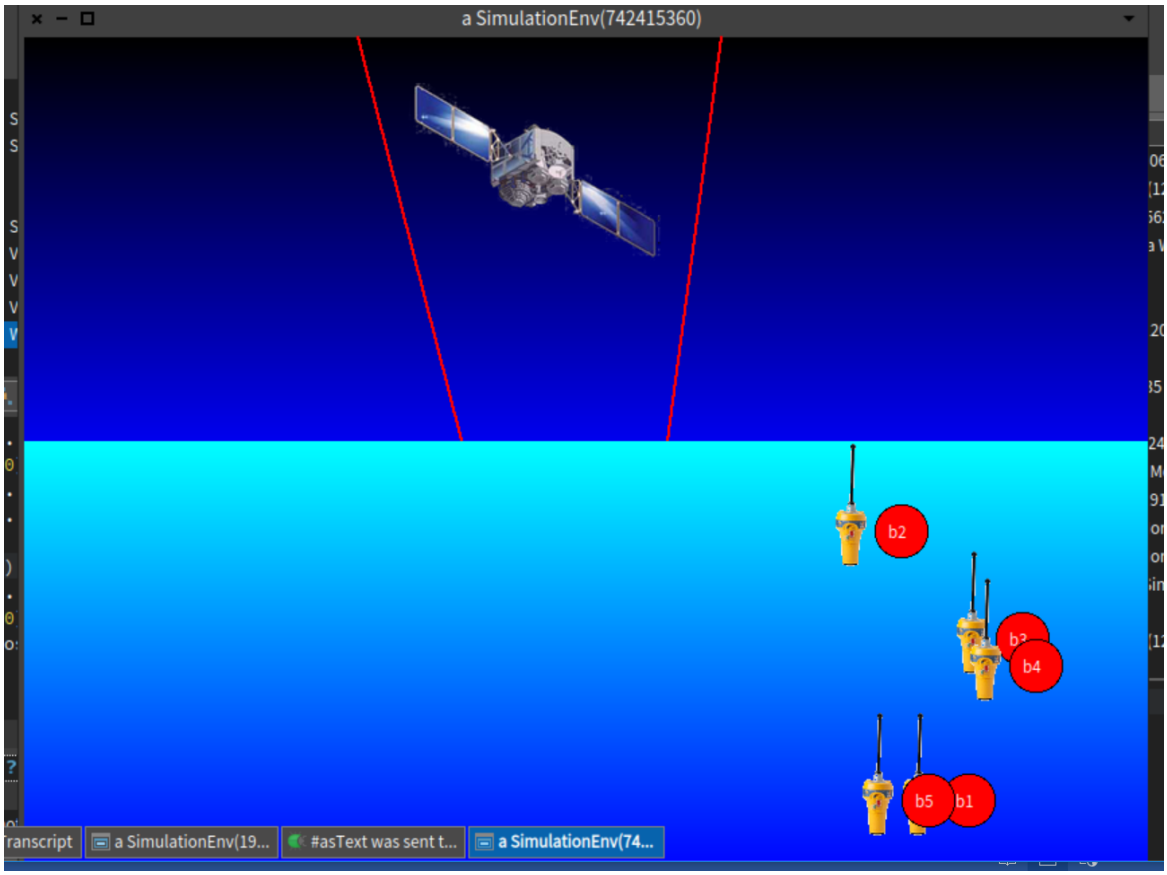
VueSatellite est la vue qui comprend le ciel et les satellites (pour simplifier on va présenter seulement un seul satellite dans la partie graphique). Elle hérite de la classe ImageMorph car le satellite est représenter par une image .png.

VueSatellite est la vue qui comprend la mer et plusieurs balises. Elle hérite de la classe ImageMorph également pour les mêmes raisons.

Appercu des objets dans l’environnementdans differentes positions de déplacement :


a SimulationEnv(742415360)
b2
b4
b3
b1
b5
#asText was sent t...
a SimulationEnv(74...


a SimulationEnv(742415360)
b2
b3
b4
b5
b1
#asText was sent t...
a SimulationEnv(74...



Pour distinguer la couleur de la mer, voici le code couleur avec une surcharge de la méthode useGradientFill:

```
self color: Color cyan.

self useGradientFill.
```

OrderedCollection :

Pour encapsuler plusieurs satellites. La classe OrderedCollection est plus général que Array ; la taille d'une OrderedCollection se développe au fil du temps à chaque fois qu'on a besoin de rajouter des satellites.

Pour se repérer dans le graphique et identifier chaque satellite, on peut extraire l'index de l'instance « balise » depuis la collection « balises » après l'avoir ajoutée :

```
x := balises indexOf: b3 ifAbsent: [0].
```

Et l'afficher dans une étiquette en bulle :

```
t := TextMorph new contents: 'b'+x.

c := CircleMorph new color: Color red.
```

```
c position: balise width+30@balise height/2.
t position : c position.
c addMorph t.
balise addMorph: c .
```

Le déplacement :

Extrait du code de la fonction du déplacement (exemple de déplacement vertical)

```
move: objet
    moveDown
        ifTrue: [ objet position: satellite position x  @ (objet position  y - 1).
            objet position y < objet owner bounds origin y
                ifTrue: [ self moveDown: false ] ]
        ifFalse: [ objet position: objet position x @ (objet position y + 1) .
            satellite position y > ((satellite owner bounds corner y * 2/3) - objet height )
                ifTrue: [ self moveDown: true ] ]
```

La partie traitement de données :