IN201 : 9 – Diagrammes d'états-transitions

Solution

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

ISAE Institut Supérieur de l'Aéronautique et de l'Espace SUPAERO

Résumé

Ce sujet de TP est un sujet récapitulatif de conception. Il vous permettra également de construire un diagramme d'états-transitions.

1 Présentation du problème

L'objectif de cet exercice est de modéliser un système de navigation par satellites (comme le GPS ou Galileo) pour réaliser un logiciel de simulation. On ne s'attachera pas ici à décrire le cœur du simulateur, mais simplement à construire un modèle du système réel.
Les systèmes de navigation comme GPS ou Galileo utilisent une constellation de satellites. Une constellation est constituée d'un ensemble de satellites. On peut considérer qu'une constellation ne possède pas de satellites si ceux-ci ne sont pas encore lancés. Une constellation est caractérisée par sa taille, et on peut ajouter un satellite dans une constellation si celle-ci n'est pas encore complète.
Chaque satellite évolue sur une orbite. Il existe plusieurs types d'orbites différentes : par exemple, une orbite géostationnaire ou une orbite de Kepler sont deux orbites particulières. Une orbite de Kepler de type J2 est elle même une orbite de Kepler particulière.
Un satellite peut embarquer des capteurs comme par exemple un radar, un altimètre ou un capteur infra-rouge.
Un satellite est caractérisé par une position courante, une vitesse courante et une date courante. On peut demander à un satellite de renvoyer toutes les informations concernant sa position.
Sur la Terre, les satellites d'une constellation peuvent être vus par des stations sol. Celles-ci ont une position représentée par des coordonnées cartésiennes. Une station sol peut être par exemple un récepteur, qui peut lui même être un récepteur Galileo ou GPS.

2 Questions

1. proposer un modèle de classes UML de conception préliminaire (analyse) identifiant les classes, les rôles et les multiplicités (ou cardinalités) ;

   Solution :
   Un diagramme de classes est proposé sur la figure 1.

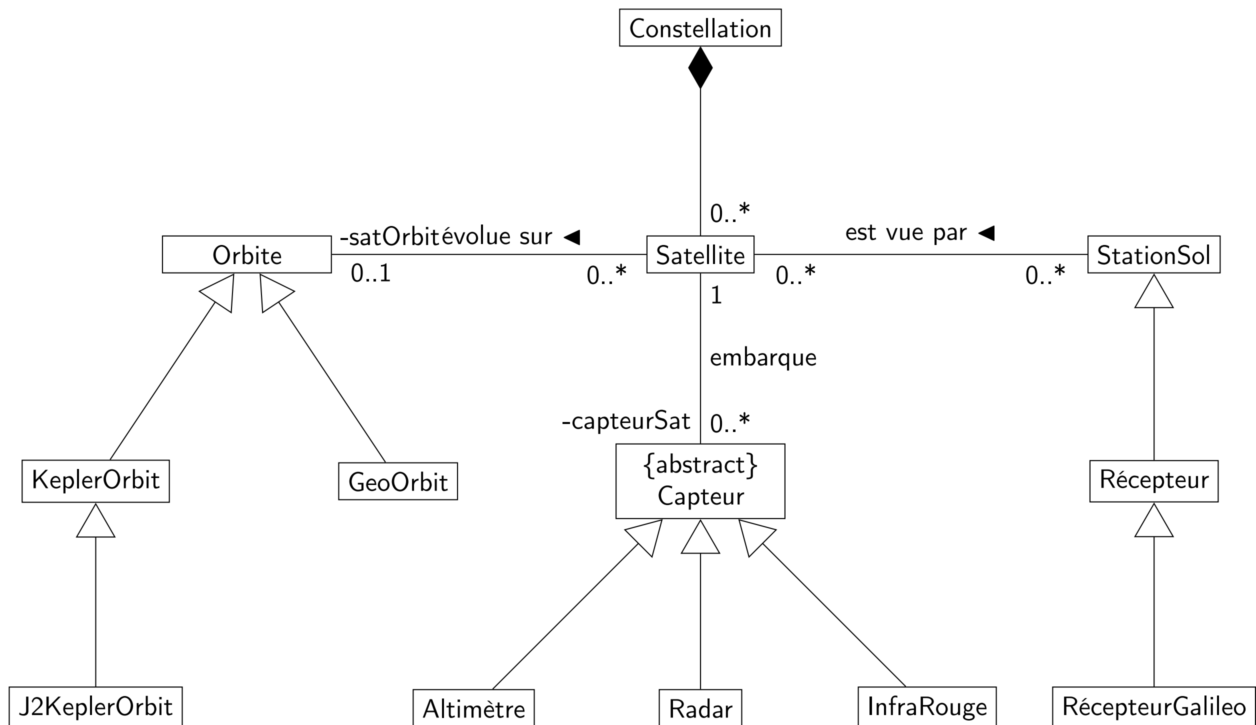

FIGURE 1 – Un diagramme de classes modélisant le problème

Quelques remarques :

- une constellation est constituée de satellites. La relation qui lie la classe `Constellation` à la classe `Satellite` est donc une association de type composition ou agrégation. On considère ici que si la constellation disparaît, les satellites disparaissent aussi. On choisit donc une relation de composition. Une constellation peut être constituée de 0 satellite (si les satellites ne sont pas encore lancés par exemple), ce qui explique la multiplicité. Par contre, la composition entre `Constellation` et `Satellite` impose implicitement une multiplicité 1 du côté de `Constellation`.
- un satellite peut embarquer des capteurs. On considère dans l'assocation `embarque` qu'un satellite peut n'embarquer aucun capteur et qu'un capteur n'est embarqué que par un seul satellite. On peut remarquer que l'on aurait pu modéliser cette relation par une relation de composition ou d'agrégation.
- une `StationSol` peut être vue par plusieurs satellites (elle peut ne pas être vue par un satellite également). Un satellite peut voir plusieurs stations sol ou ne pas en voir.
- un satellite évolue sur une orbite. S'il n'est lié à aucune orbite, c'est par exemple qu'il n'a pas encore été lancé. Par contre, plusieurs satellites peuvent évoluer sur une orbite.
- les relations de généralisation/spécialisation se trouvaient aisément dans le texte. Il n'y avait pas de difficulté particulière.
- j'ai choisi de modéliser `Orbite` par une classe abstraite, car je pense que l'on peut embarquer du code dans une classe `Orbite` : toutes les orbites sont définies par les paramètres d'une ellipse, les différents types d'orbites imposant des contraintes particulières sur ces paramètres. De plus, toutes les orbites fournissent des services identiques.
- j'ai choisi de modéliser `Capteur` par une classe abstraite, car je pense qu'il peut y avoir du code commun à tous les capteurs (renvoyer une mesure par exemple).
- enfin, `StationSol` est une classe concrète, spécialisée par les récepteurs.

2. proposer une description détaillée de la classe `Satellite` (attributs et opérations). Vous vous limiterez à un petit nombre d'opérations nécessaires à la réalisation de cette classe. Vous considérerez que vous avez à votre disposition une classe `Date` ;

Solution :

Une description détaillée de la classe `Satellite` est présentée sur la figure 2.

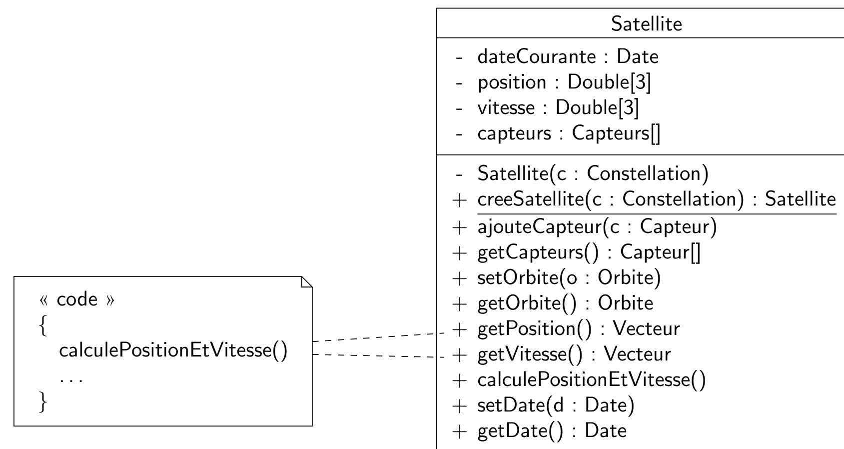

FIGURE 2 – Une modélisation de la classe `Satellite`

On remarquera que l'on ne peut pas changer directement la position et la vitesse d'un satellite. Ce calcul peut être fait en « interne » en connaissant l'orbite du satellite grâce à la méthode `calculePositionEtVitesse`. Cela signifie aussi que la méthode de calcul de la position et de la vitesse à partir de l'orbite est suffisamment précise pour ne pas nécessiter un recalibrage extérieur [1]. J'ai choisi de laisser cette méthode publique et d'imposer son utilisation lors de l'appel à `getPosition` et `getVitesse` (cf. note correspondante sur le diagramme UML). Un utilisateur extérieur souhaitant utiliser la classe devra également ne pas oublier de positionner la date avant de demander un calcul.

Physiquement, lorsqu'un satellite veut changer d'orbite, il faut effectuer des manœuvres (i.e. envoyer des ordres de poussée aux moteurs). Nous avons donc ici un modèle de satellite « haut niveau » dans lequel on suppose que les problèmes de changements d'orbite sont résolus.

Reste le problème du constructeur. Dans le modèle proposé, un satellite n'a pas forcément d'orbite, donc on utilisera une méthode `setOrbite` pour positionner l'orbite du satellite (idem pour les capteurs). Par contre, un satellite appartient obligatoirement à une constellation. Il faudrait donc passer une constellation en paramètre du constructeur de `Satellite`. La date courante ne sera pas passée en paramètre du constructeur, mais calculée « à la volée » par le constructeur.

Si l'on considère qu'une constellation à un nombre limité de satellites, il ne faut pas que l'on puisse créer un satellite et l'attacher à une constellation si celle-ci est déjà complète. Or, dès que l'on utilise un constructeur, on créé un objet. On va donc rendre le constructeur de `Satellite` privé et utiliser une méthode *factory* `creeSatellite` qui va renvoyer un objet de type satellite. Cette méthode vérifiera que la constellation n'est pas complète avant de construire le satellite. Elle est *statique*, de sorte qu'on n'a pas besoin d'une instance de `Satellite` pour l'appeler.

3. nous nous intéressons maintenant à la création des capteurs embarqués sur un satellite. Nous ne considérerons ici que les capteurs `Radar` et `Altimetre`. On suppose que chaque capteur a besoin de trois éléments pour pouvoir être construit et embarqué sur le satellite :
   - un support physique pour mettre en place le capteur sur le satellite
   - les instruments constituant le capteur lui-même
   - une interface logicielle et matérielle permettant au capteur de dialoguer avec le système informatique du satellite

   On suppose que l'on dispose donc de la hiérarche présentée sur la figure 3. On suppose également que la classe `Capteur` a la structure présentée sur la figure 4. Quelques précisions :
   - la méthode `initComponents` est une méthode abstraite dont le rôle est d'initialiser le support, l'instrument et l'interface satellite du capteur.
   - la méthode `createCapteur` est une *factory method* permettant de créer un capteur à partir d'une chaîne de caractères. Cette méthode sera utilisée pour créer des capteurs.

   On cherche à assurer la cohérence des composants de chaque capteur : par exemple, un capteur de type `Radar` ne pourra avoir utiliser que `SupportRadar`, `InstrumentRadar` et `InterfaceSatRadar`.

   Le code de la classe `Capteur` est donné sur le listing 1.

Listing 1– La classe Capteur

```
public abstract class Capteur {
```

```
    protected Support support;
    protected Instrument instrument;
    protected InterfaceSat interfaceSat;

    public static Capteur createCapteur(String type) {
        Capteur c = null;

        if (type.equals("radar")) {
            c = new Radar();
        } else if (type.equals("altimetre")) {
            c = new Altimetre();
        }

        if (c != null) {
            c.initComponents();
        }

        return c;
    }

    public abstract void initComponents();

}
```

(a) écrire un diagramme de séquence représentant la création d'un capteur de type `Radar`.

Solution :

Rien de bien compliqué, le diagramme de séquence est présenté sur la figure 5. On remarquera la syntaxe pour un appel à une méthode statique.

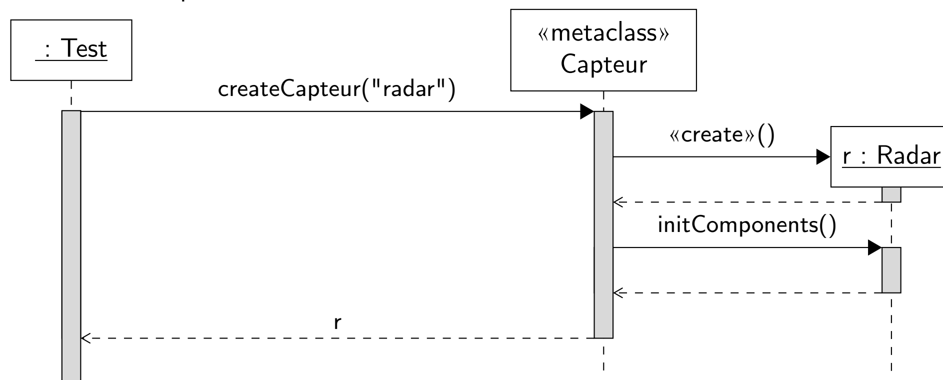

FIGURE 5 – Diagramme de séquence présentant la création d'une instance de `Capteur`

(b) peut-on garantir qu'un capteur sera bien construit avec les composants avec cette architecture ?

Solution :

On ne peut pas garantir que les capteurs seront correctement construits, car rien n'empêche une spécialisation de `Capteur` de redéfinir la méthode `initComponents` en initialisant les composants de façon « illicite ».

(c) pour pouvoir construire une famille d'objets, on peut utiliser le patron de conception *abstract factory* présenté sur la figure 6. Adaptez le patron de conception au problème de la création de capteur. La méthode `initComponent` reste-t-elle abstraite ?

Solution :

On peut utiliser le patron *abstract factory* pour créer des familles de composant pour capteurs. Le diagramme de classes ainsi créé est présenté sur la figure 7.

En ce qui concerne la classe `Capteur`, on peut maintenant écrire la méthode `initComponents` directement dans cette classe et imposer la *factory* utilisée à ses sous-classes, la liaison dynamique se chargeant de créer les bons composants. De plus, on peut facilement rendre les attributs de la classe privés. Le code de la classe est présenté

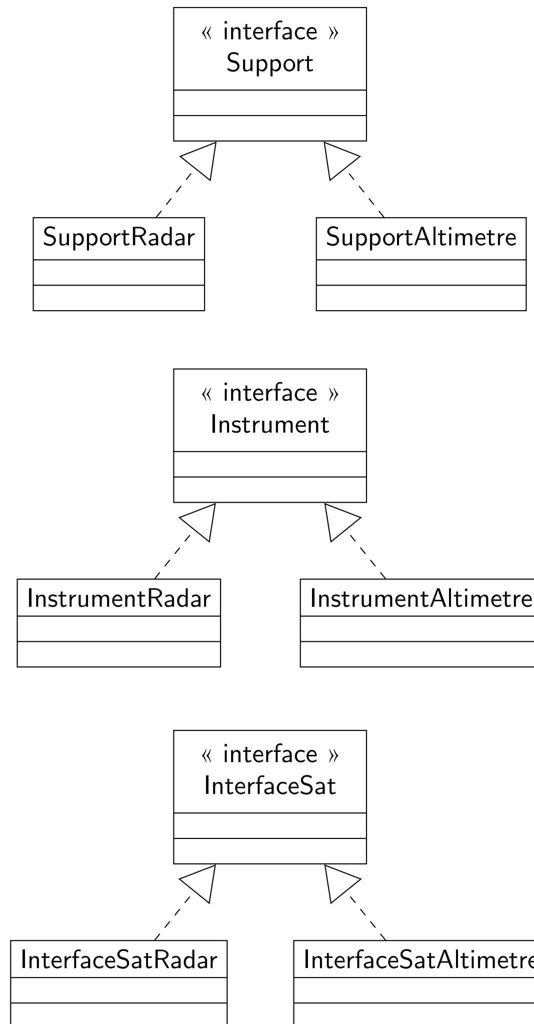

FIGURE 3 – Hiérarchie des composants des capteurs

sur le listing 2. Bien sûr, rien n'empêche le développeur de la classe `Radar` de ne pas utiliser la *factory* passée en paramètre et d'utiliser une autre *factory* lors de l'appel à **super**...

Listing 2– La classe `Capteur` utilisant l'*abstract factory*

```
public abstract class Capteur {

    private Support support;
    private Instrument instrument;
    private InterfaceSat interfaceSat;
    private ComponentsFactory componentsFactory;

    public Capteur(ComponentsFactory f) {
        this.componentsFactory = f;
    }

    public static Capteur createCapteur(String type) {
        Capteur c = null;

        if (type.equals("radar")) {
            c = new Radar(new ComponentsFactoryRadar());
        } else if (type.equals("altimetre")) {
            c = new Altimetre(new ComponentsFactoryAltimetre());
```

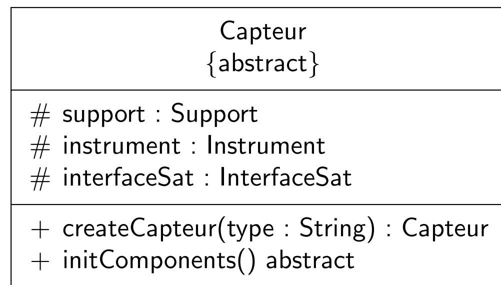

FIGURE 4 – La classe abstraite `Capteur`

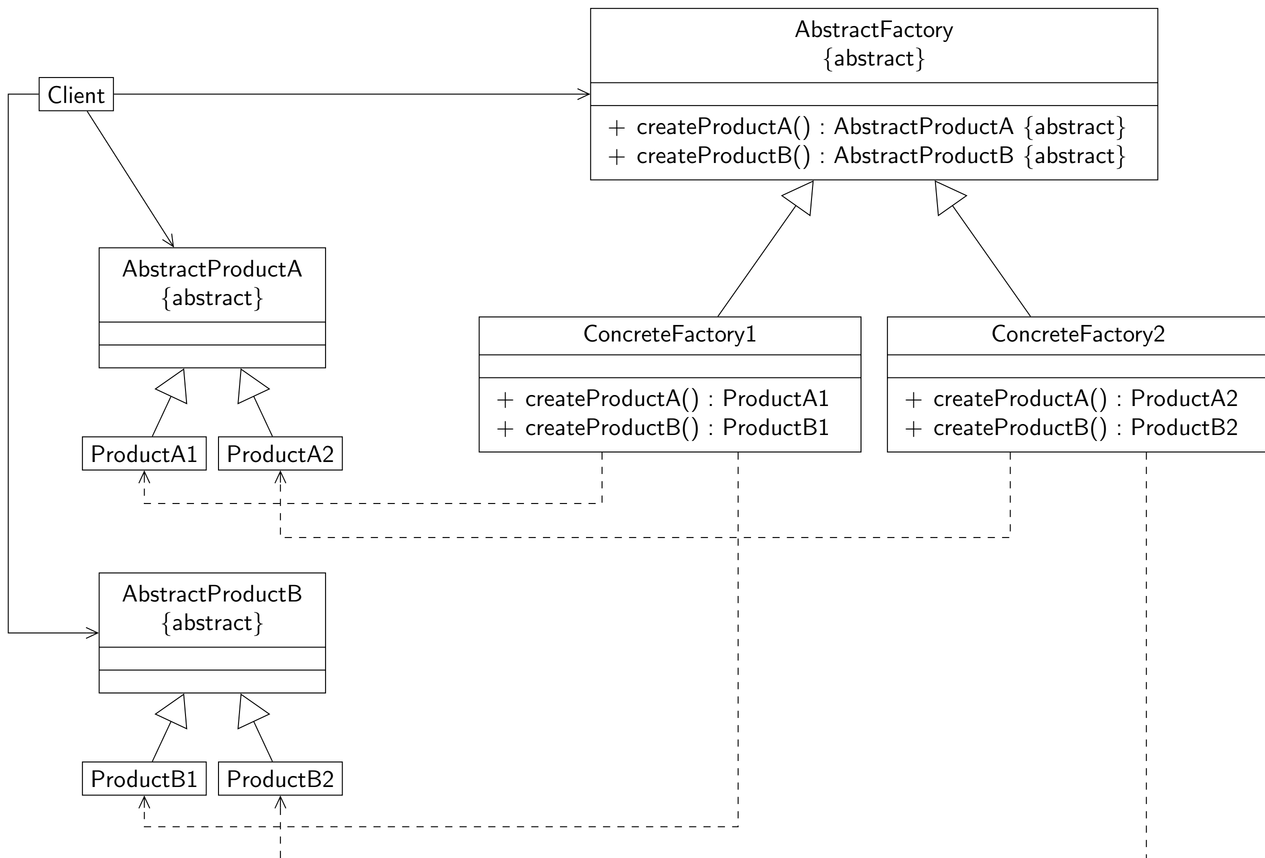

FIGURE 6 – La patron de conception *abstract factory*

```
        }

        if (c != null) {
            c.initComponents();
        }

        return c;
    }

    private void initComponents() {
        this.support = this.componentsFactory.createSupport();
        this.instrument = this.componentsFactory.createInstrument();
        this.interfaceSat = this.componentsFactory.createInterfaceSat();
    }
}
```

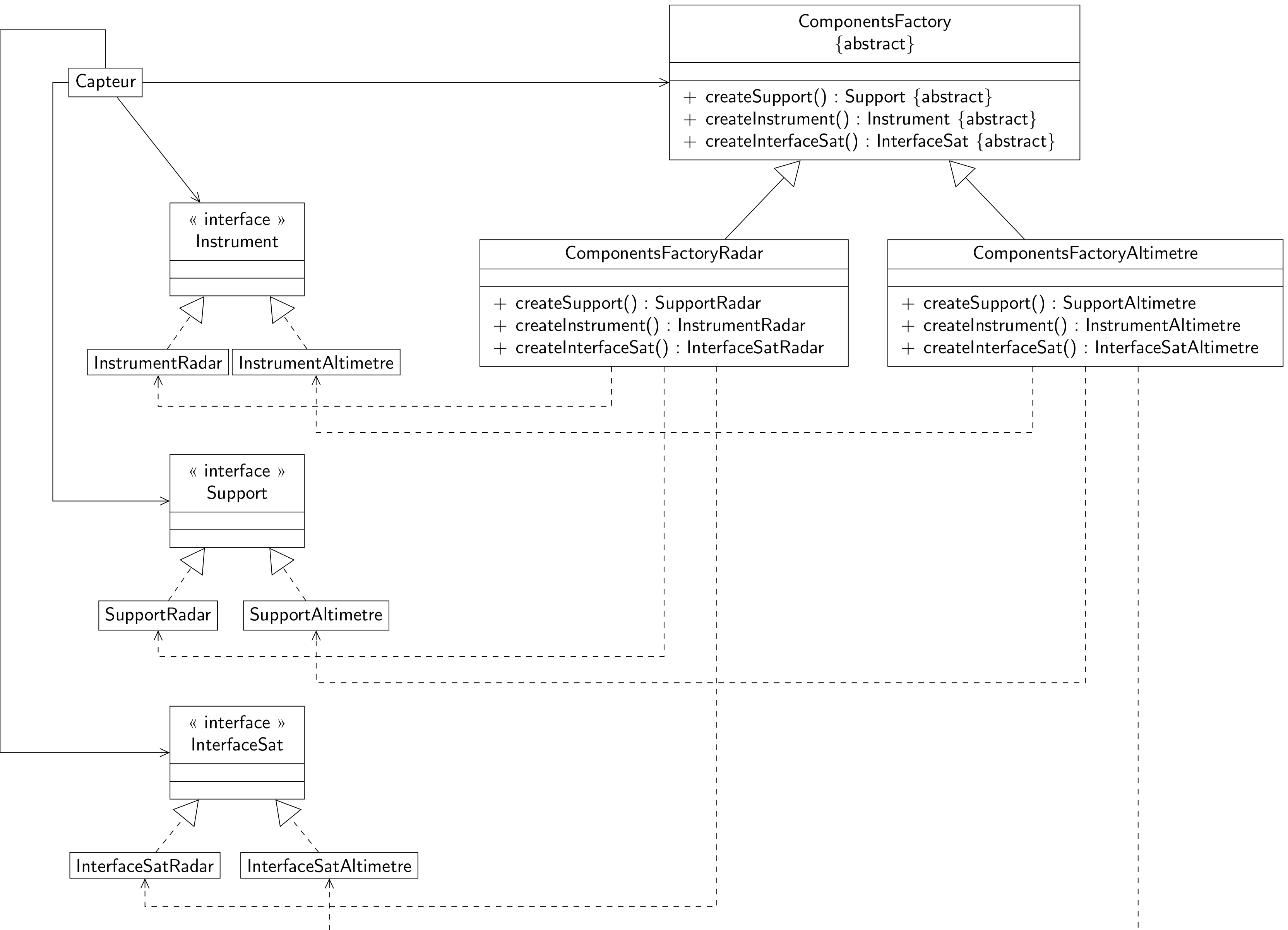

Figure 7 – Le patron de conception *abstract factory* adapté à notre problème

4. nous proposons dans cette question de modéliser grâce à un diagramme d'états-transitions le comportement d'un récepteur de type Galileo[2].

   Lorsque l'on met en marche un récepteur Galileo, celui-ci cherche tout d'abord à savoir s'il dispose d'une première approximation de la date actuelle. S'il ne l'a pas, il entre dans un état et :
   - soit il dispose d'une interface réseau et récupère une approximation de la date via un serveur ntp (*Network Time Protocol*) par exemple ;
   - soit il attend que l'utilisateur lui fournisse une date en utilisant un clavier par exemple.

   Lorsque le récepteur dispose d'une approximation de la date actuelle, il entre dans un état dans lequel il cherche à orienter son antenne pour trouver un premier satellite en utilisant les éphémérides dont il dispose. Cette phase peut prendre un certain temps. Lorsque le récepteur a trouvé un satellite, il entre dans un nouvel état dans lequel il ajuste la date actuelle.

   À ce moment, dès que le récepteur va « voir » 4 satellites de la constellation, il va entrer dans un état `NavigationSolution` qui va lui permettre de calculer sa position et la date actuelle. Sinon, il entre dans un mode `NoNavigationSolution`. Ce mode est obligatoirement atteint dès que moins de 4 satellites sont visibles. Dans le mode `NoNavigationSolution`, si un certain laps de temps s'écoule (modélisé par une fonction f que l'on suppose connue) le récepteur va de nouveau rechercher un premier satellite. Si 4 satellites sont de nouveau visibles avant ce laps de temps, le récepteur est de nouveau capable de calculer sa position et la date actuelle.

   On suppose également que le récepteur Galileo est couplé à une centrale inertielle. Celle-ci permet de connaître une position pendant un certain temps. Si la position donnée par la centrale est valide, la condition `INS` (pour *Inertial Navigation System*) est vraie. Lorsque le récepteur est dans l'état `NoNavigationSolution`, la position donnée par la centrale peut être utilisée par l'utilisateur pour connaître sa position.

   De plus, si la durée f s'est écoulée depuis l'entrée du récepteur dans l'état `NoNavigationSolution` et que la centrale inertielle est valide, le récepteur entre dans une une phase de recalibrage utilisant la centrale inertielle plutôt que de retourner directement dans l'état de recherche du premier satellite. Au bout d'une unité de temps, si 4 satellites sont vus, le récepteur devient alors opérationnel, sinon le récepteur recherche à nouveau un premier satellite. Cela permet de gagner éventuellement du temps par rapport à la phase d'initialisation.

   Donner le diagramme d'états-transitions correspondant au fonctionnement du récepteur.

   Solution :

   Un diagramme d'états-transitions est donné sur la figure 8. Il n'y avait pas de difficultés particulières sur cet exercice, il fallait juste faire attention aux problèmes de non-déterminisme en imposant des conditions de garde exclusives. On suppose dans le corrigé que `lessThanFourSatellitesVisible` est une condition dont la négation est `fourSatellitesVisible`.

   On remarquera que l'on utilise beaucoup de conditions de garde et peu d'événements. Ceci est tout à fait normal, car on est ici en phase d'analyse et on ne dispose pas encore de solutions d'implantation proposant des signaux/méthodes permettant de gérer les transitions. Il aurait fallu par exemple proposer des diagrammes plus complexes intégrant la constellation pour générer les événements correspondants aux conditions de visibilité des satellites.

   Les transitions n'utilisant qu'une garde (et pas d'événement) sont déclenchées (à condition que la garde correspondante soit vraie) dès que les activités internes de l'état sont finies. Cela était correct pour la sortie de l'état `AdaptingTime` par exemple, mais pour certaines transitions entre `NavigationSolution` et `NoNavigationSolution`, on était obligé d'utiliser des événements **when** sur les changements de valeur de vérité des conditions (car par exemple, l'activité `computeXYZT` de `NavigationSolution` ne s'« arrête » jamais).

   On remarquera enfin que pour respecter la sémantique d'UML, les transitions partant des pseudo-états initiaux n'utilisent pas de gardes ou d'événements : on utilise un vrai état comme `Booting` par exemple.

2. Ce modèle de comportement est très simplifié par rapport à la réalité !

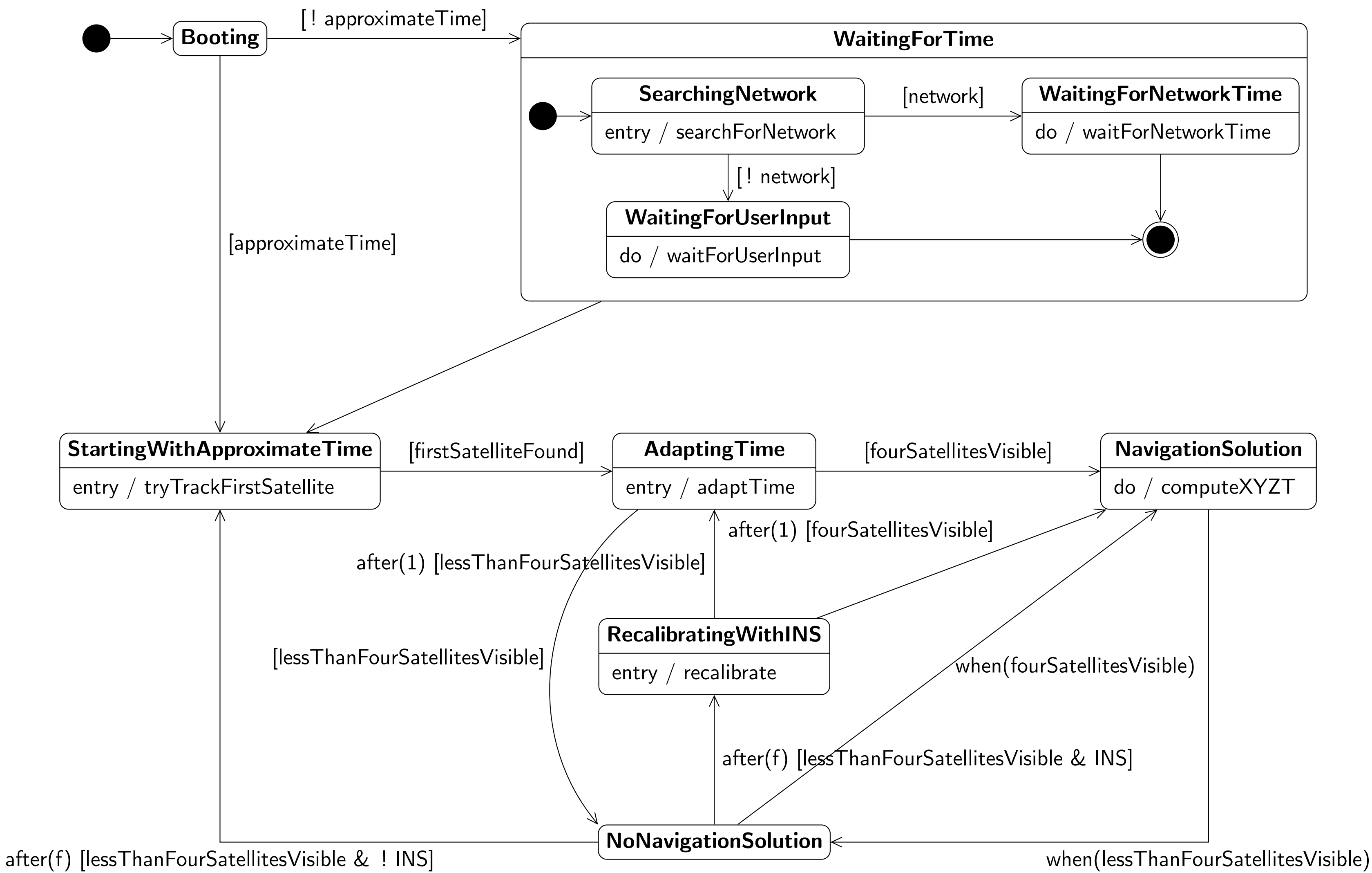

Figure 8 – Diagramme d'états-transitions du récepteur Galileo