

Software Engineering

(SWE)

Medard Rieder (medard.rieder@hevs.ch)

Thomas Sterren (thomas.sterren@hevs.ch)

© 2018 HES-SO Valais - Systèmes Industriels

Table des matières

Table des matières	1
La notation UML.....	1
Historique.....	1
Introduction	1
Diagrammes UML statiques	2
<i>Les diagrammes de classes.....</i>	2
<i>Les diagrammes d'objets.....</i>	3
<i>Les diagrammes de paquets.....</i>	4
<i>La dépendance ou bien l'utilisation.....</i>	5
<i>L'association.....</i>	5
<i>La généralisation.....</i>	9
<i>L'implémentation.....</i>	9
<i>Exercices.....</i>	11
Diagrammes UML dynamiques	12
<i>Les diagrammes de cas d'utilisation.....</i>	12
<i>Scénarii.....</i>	14
<i>Diagrammes de séquence.....</i>	15
<i>Diagrammes de collaboration</i>	17
<i>Exercices.....</i>	19
Diagrammes UML physiques.....	20
<i>Exercices.....</i>	22
Diagrammes UML d'implémentation.....	23
<i>Diagrammes d'états-transitions.....</i>	23
<i>Diagrammes d'activité.....</i>	26
<i>Diagrammes de composants.....</i>	28
<i>Exercices.....</i>	29
Développement basé modèle	30
Cycle de vie	30
Un processus simple	31
Description du système.....	33
Analyse.....	33
<i>Diagramme physique.....</i>	33
<i>Analyse comportementale.....</i>	34
<i>Analyse structurelle.....</i>	36
<i>Consolidation des différents résultats d'analyse.....</i>	37
Conception.....	41
<i>Conception au niveau des composants</i>	41
<i>Conception relative aux classes.....</i>	44
Implémentation.....	47
<i>Implémentation des méthodes.....</i>	47
<i>Implémentation du comportement</i>	47
<i>Implémentation des composants.....</i>	53
Vérification	62
<i>Vérification du comportement.....</i>	62
<i>Vérification de la stabilité.....</i>	63

La notation UML



L'objectif de ce chapitre est de donner une introduction à la notation UML. Les éléments les plus importants sont expliqués et illustrés par des exemples simples et faciles à comprendre.

Historique

UML a été introduit par Booch, Rambough et Jacobson en 1995. Chacun des trois était un spécialiste en développement et amenait sa propre partie à cette nouvelle notation, d'où vient aussi son nom : Unified Modelling Language. Depuis 1995, cette notation a été continuellement développée et améliorée et elle a atteint le statut de "la notation". UML est aujourd'hui disponible en version 2 et est reconnue comme notation standard pour la modélisation orientée objet. UML n'est pas seulement utilisé pour développer de gros systèmes mais il commence aussi à devenir un standard pour les systèmes embarqués.

Introduction

Cette section a comme but de donner un aperçu des différents diagrammes qui sont disponibles en UML. Il est clair qu'on peut classer ces diagrammes de différentes manières. Celle qui est utilisée ici est un classement d'après différentes vues d'un système. L'énumération suivante donne ces vues :

- Vue statique
- Vue dynamique
- Vue physique
- Vue d'implémentation

Figure 1 : Différentes vues d'un système

Chacune de ces vues utilise un certain nombre de diagrammes UML pour décrire un système plus ou moins complexe. Les sections suivantes vont donner plus de détails concernant ces diagrammes et leur application et interprétation.

Diagrammes UML statiques

Dans la section suivante sont introduits les diagrammes UML utiles à décrire la vue statique d'un système. Ceux-ci sont :

- Diagrammes de classes
- Diagrammes d'objets
- Diagrammes de paquets

Figure 2 : Diagrammes statiques

Un autre élément qui sera aussi abordé dans cette section est les relations entre classes et objets, énumérées ci-dessous :

- Utilisation
- Association
- Généralisation

Figure 3 : Relations

Les diagrammes de classes

Les diagrammes de classes servent à représenter une ou plusieurs classes. Ils sont capables de décrire :

- le nom de la classe
- les attributs de la classe
- les méthodes de la classe.

Figure 4 : Éléments d'une classe

La figure suivante montre la classe Human en UML.

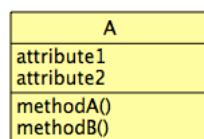


Figure 5 : La classe Human

Comme on peut voir dans ce diagramme, les attributs et les méthodes se distinguent uniquement par les parenthèses qui suivent les noms des méthodes. UML offre par contre la possibilité de décrire plus précisément soit les attributs soit les méthodes. La syntaxe utilisée dans ce cas-là ressemble à celle du langage de programmation PASCAL. La figure suivante montre cette syntaxe :

- visibility nameOfAttribute:typeOfAttribute
- visibility nameOfMethod(paramList) :typeOfMethod
- nameOfParam1:typeOfParam1,...,nameOfParamN:typeOfParamN

Figure 6 : Attributs et méthodes

Pour la visibilité, trois types différents sont connus en UML :

- **private : (-)** : L'élément n'est pas visible hors de la classe
- **public : (+)** : L'élément est librement accessible
- **protected : (#)** : L'élément est visible dans la classe elle-même et dans les classes qui héritent de cette classe

Figure 7 : Types de visibilité

En appliquant cette information à la classe `Human`, elle prendra la forme montrée ci-dessous :

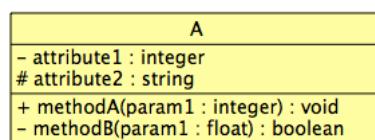


Figure 8 : La classe Human version 2

Il reste à mentionner qu'il est possible de laisser tomber certaines informations dans un diagramme de classes. Le choix est laissé à l'ingénieur-e entre les informations d'une classe qu'il veut montrer et celles qu'il veut cacher. Dans des gros systèmes, on voit souvent différentes représentations d'un même élément. L'information montrée dépend du contexte et de son importance. La figure suivante montre des variantes possibles de diagrammes de classes en UML :

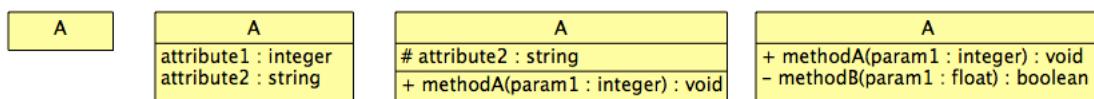


Figure 9 : Représentations différentes de classes en UML

Les diagrammes d'objets

Les diagrammes d'objets représentent des objets qui sont instanciés à partir de certaines classes. La représentation d'un objet est généralement la même que celle d'une classe. La vraie différence consiste dans la notation du nom d'un objet.

Pour les méthodes et les attributs il n'y a aucune différence. La figure suivante montre l'objet `Frederik` qui est une instance de la classe `Human`.

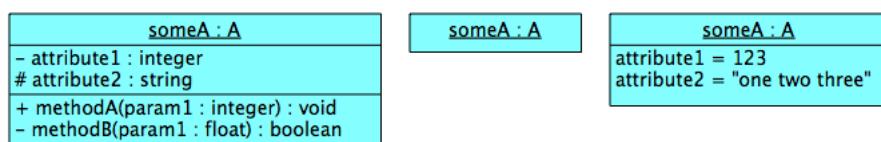


Figure 10 : L'objet Frederik

Remarques :

- Comme on peut voir dans la figure précédente, il est possible de représenter un objet de différentes manières. En tout cas, le nom de l'objet est suivi par deux-points et le nom de la classe. Les attributs et les méthodes peuvent être représentés comme c'était déjà le cas pour des classes. Mais il est aussi possible de seulement représenter les attributs avec leurs valeurs instantanées. Cette dernière représentation est plutôt rarement utilisée.
- La représentation d'objets est surtout utilisée dans les trois situations suivantes :
 - Dans des classes composites (attributs objets)
 - Dans des diagrammes de séquence
 - Dans des diagrammes de collaboration

Les deux derniers diagrammes seront introduits dans les sections suivantes.

Les diagrammes de paquets

Les diagrammes de paquets sont utiles pour structurer les systèmes. Ils regroupent un certain nombre d'éléments selon des critères logiques. Ces critères sont uniquement choisis par l'ingénieur-e ou bien imposés par des modèles spécifiques (pattern). Les paquets ne sont pas traduits en code, ils ont plutôt l'effet de créer des répertoires contenant les éléments du paquet. Un thème très important est la création d'interfaces entre différents paquets. Ce thème sera abordé dans un chapitre ultérieur. La figure suivante montre un diagramme UML avec des paquets.

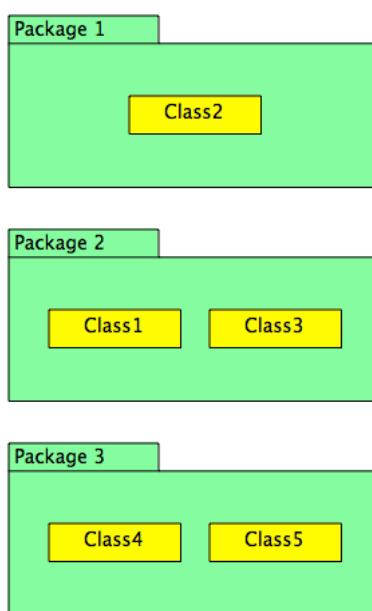


Figure 11 : Paquets

La dépendance ou bien l'utilisation

Les prochaines trois sections vont décrire les relations principales qui peuvent être représentées en UML. La section actuelle décrit la relation la moins forte qui est l'utilisation. Cette relation peut exister entre des classes ou des paquets ou bien même entre classes et paquets. Elle est représentée par une flèche traitillée avec une pointe simple. La figure suivante montre quelques exemples de cette relation.

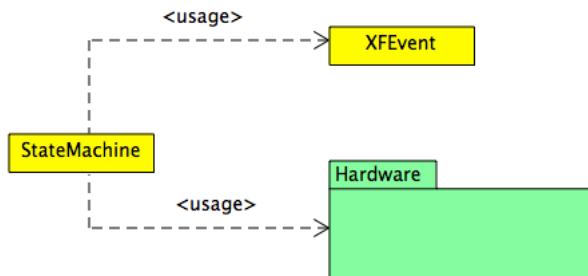


Figure 12 : Utilisation

Remarques :

- Une utilisation ne s'applique jamais entre objets.
- L'implémentation d'une utilisation s'effectue au moyen de ce qui est possible dans le langage de programmation cible. Entre autre, les constructions suivantes sont possibles :
 - l'instruction `import` de Java
 - l'instruction `#include` de C / C++

Si plusieurs possibilités existent, des stéréotypes peuvent être utilisés pour préciser l'option désirée. Un stéréotype est un mot-clé entre des parenthèses plus grand - plus petit.

L'association

Une association est une façon d'indiquer qu'il existe une relation entre deux classes. Il y a trois formes d'association :

- **L'association** : C'est la forme la plus générale. Elle est plutôt vague, mais il est clair qu'une relation existe. En programmation, cette relation est normalement exprimée par un pointeur ou bien une référence. En UML, cette forme est représentée par un trait simple.

Figure 13 : L'association générale

- **L'agrégation** : C'est une relation déjà plus distincte et les classes concernées ont à faire l'une avec l'autre, mais une classe peut toujours exister sans l'autre. En programmation, cette relation est exprimée par un pointeur ou bien une référence. En UML, cette forme est représentée par un trait simple avec un losange vide d'un côté.

Figure 14 : L'agrégation

- **La composition :** C'est une liaison physique. Une classe ne peut pas exister sans l'autre, donc elles sont difficilement dissociables. En programmation, cette relation est exprimée par une classe composite, ça veut dire une classe dont les attributs sont des objets d'une ou plusieurs autres classes. En UML, cette forme est représentée par un trait simple avec un losange rempli d'un côté.

Figure 15 : La composition

La figure suivante donne des exemples pour les formes d'associations :

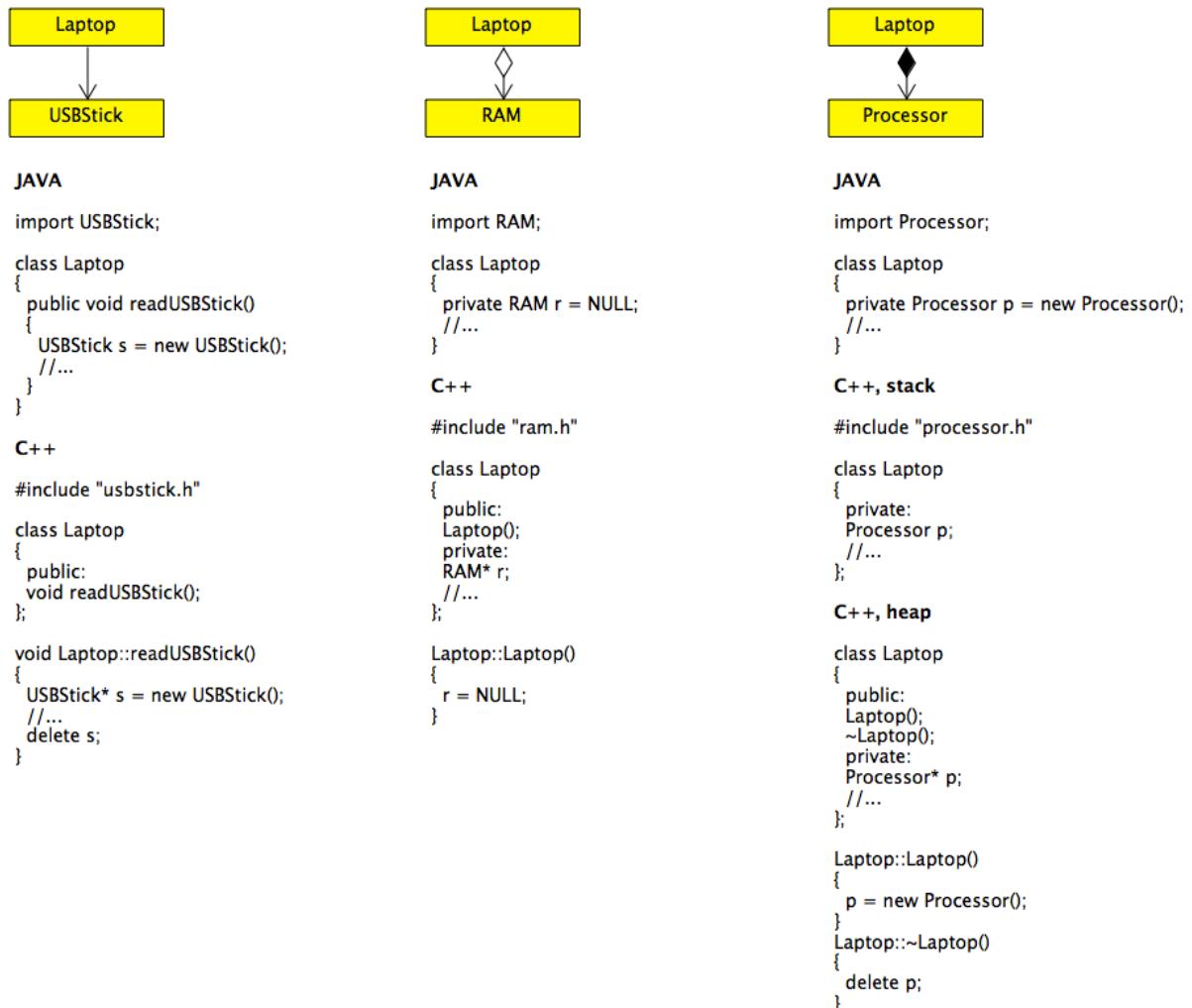


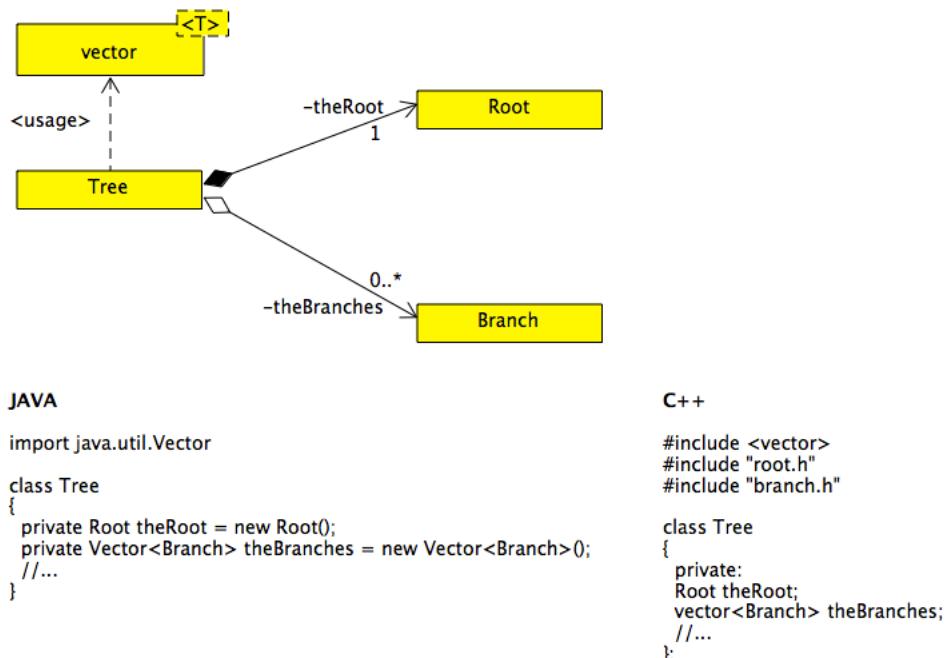
Figure 16 : Exemples d'associations

Des associations peuvent être décorées de plusieurs manières. Les décosrations possibles et souvent utilisées sont les suivantes :

- Le nom de l'association, exprimé par un texte
- La multiplicité d'une association, exprimée par un nombre N, une plage sous forme de N .. M ou bien une étoile * si le nombre est variable
- Les noms du rôle des deux extrémités d'une association, exprimé sous forme de texte
- La direction d'une association, exprimée par une flèche simple.

Figure 17 : Décorations d'une association

La figure suivante donne des exemples d'associations décorées :



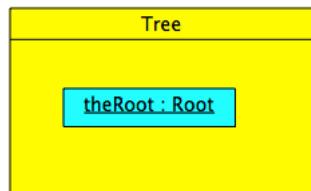
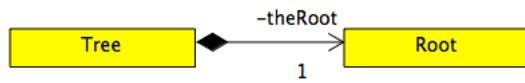


Figure 19 : Classe composite

Dans les diagrammes d'objets, les associations entre les classes deviennent des liens entre les objets. Il peut sans autre être dit qu'un lien est une instance d'une association. La figure suivante illustre ce fait :

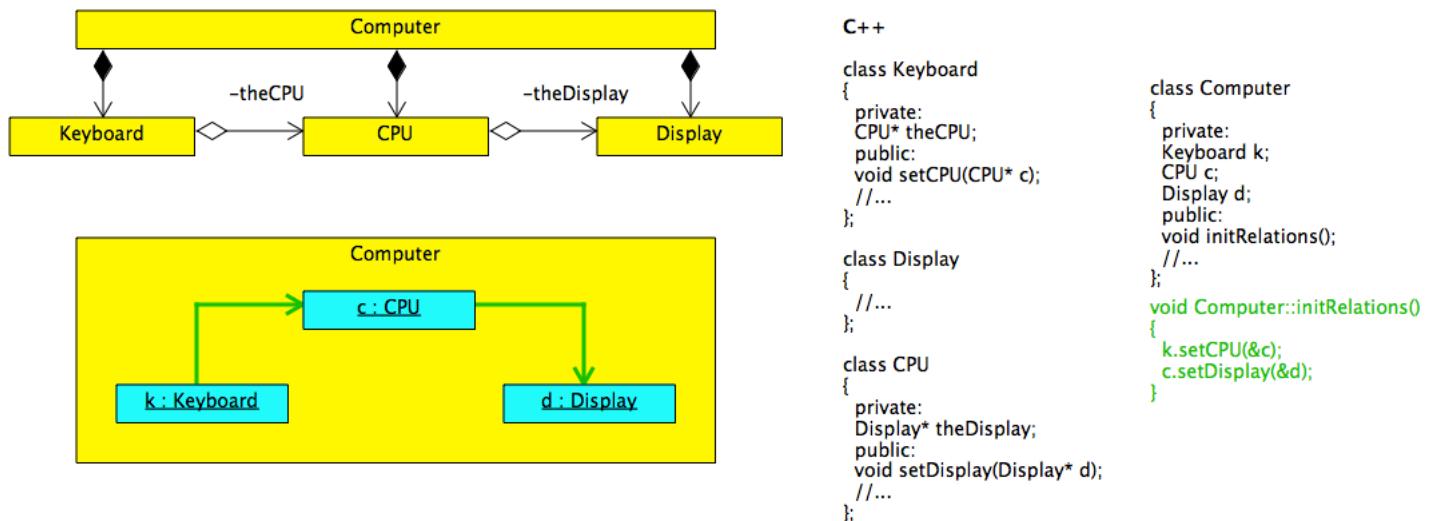


Figure 20 : Association et lien

La généralisation

La généralisation est la relation la plus forte entre deux classes. En UML, elle est représentée par une flèche avec une pointe sous la forme d'un triangle vide. En programmation, elle est exprimée par le concept d'héritage. La figure suivante montre un exemple :

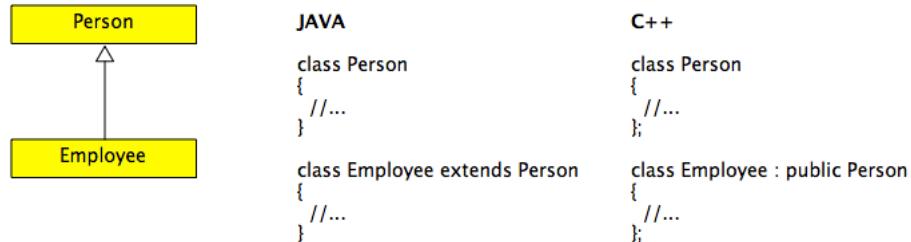


Figure 21 : La généralisation

Comme on peut le remarquer, la flèche pointe de la classe qui hérite vers la classe dont elle hérite. La raison est la suivante : le vrai nom de cette relation est la généralisation. Donc la flèche pointe dans la direction dans laquelle la généralisation s'effectue. Dans l'image ci-dessus, la généralisation va de la classe `Employee` qui est plus spécialisée (ou moins générale) vers la classe `Person` qui est moins spécialisée (ou plus générale). La figure suivante démontre ce qui vient d'être dit :

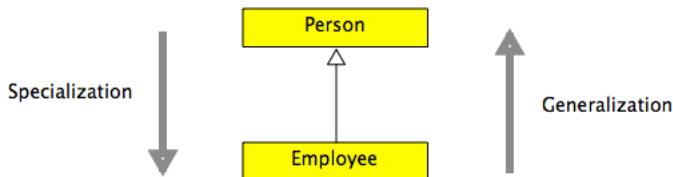
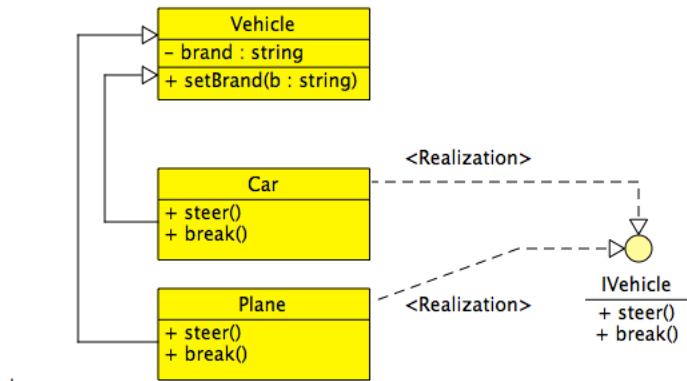
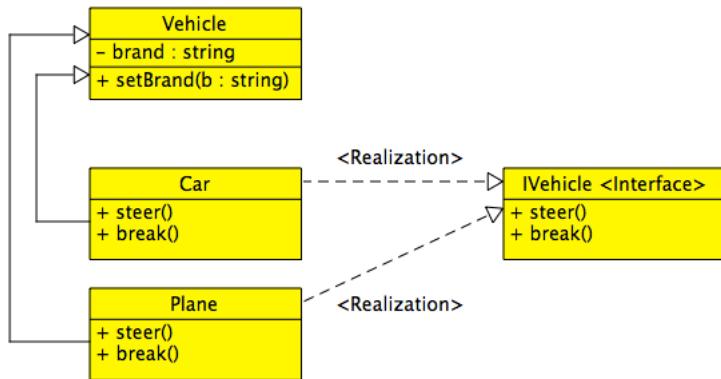


Figure 22 : Spécialisation et généralisation

L'implémentation

Une forme spéciale de la généralisation est l'implémentation. Elle est utilisée pour indiquer qu'une classe implémente une interface. L'implémentation est représentée en UML de la même manière qu'une généralisation, sauf que la ligne de la flèche est traitillée ou bien il y a le stéréotype "implémentation". La figure suivante en montre un exemple :



JAVA

```

class Vehicle
{
    private String brand;
    public void setBrand( String b )
    {
        brand = b;
    }
}

interface IVehicle
{
    public void steer();
    public void break();
}

class Car extends Vehicle implements IVehicle
{
    public void steer()
    {
        System.out.println("use the steering wheel");
    }
    public void break()
    {
        System.out.println("press the brake pedal");
    }
}

class Plane extends Vehicle implements IVehicle
{
    public void steer()
    {
        System.out.println("use the stick and the pedals");
    }
    public void break()
    {
        System.out.println("set the flaps");
    }
}
  
```

```

class Vehicle
{
    private;
    String brand;
    public;
    void setBrand( String b )
    {
    }

    void Vehicle::setBrand( String b )
    {
        brand = b;
    }
}

class IVehicle
{
    public;
    virtual void steer() = 0;
    virtual void break() = 0;
}
  
```

```

class Car : public Vehicle, public IVehicle
{
    public;
    void steer();
    void break();
}

void Car::steer()
{
    cout << "use the steering wheel";
}
void Car::break()
{
    cout << "press the brake pedal";
}
  
```

```

class Plane : public Vehicle, public IVehicle
{
    public;
    void steer();
    void break();
}

void Plane::steer()
{
    cout << "use the stick and the pedals";
}
void Plane::break()
{
    cout << "set the flaps";
}
  
```

Figure 23 : Implémentation

Exercices

- Modélez une pile de chaînes de caractères. Elle dispose d'un nombre maximal d'éléments qui peuvent être empilés ou dépilerés. On peut aussi savoir le nombre actuel d'éléments.
- Modélez une personne. Elle dispose d'un nom, d'un âge, d'un sexe et d'une couleur des cheveux et des yeux.
- Modélez une voiture en UML. Celle-ci possède quatre roues, un moteur et un propriétaire.

Figure 24 : Exercices

Diagrammes UML dynamiques

Cette section traite les diagrammes dynamiques d'UML. Ces derniers sont des diagrammes qui sont capables de décrire le comportement d'un système. Il faut aussi mentionner que ces diagrammes sont très utiles au moment d'une première spécification d'un système. Ils sont conçus d'une manière qui permet aussi à quelqu'un de non formé en informatique de les comprendre. On connaît en principe trois types de diagrammes dynamiques :

- Les diagrammes de cas d'utilisation
- Les diagrammes de séquence
- Les diagrammes de collaboration

Figure 25 : Diagrammes dynamiques en UML

Il faut mentionner que les diagrammes de séquence et les diagrammes de collaboration expriment en principe le même contenu, la différence étant que les diagrammes de séquence sont plutôt orientés temps tandis que les diagrammes de collaboration montrent plutôt l'aspect de la collaboration entre objets (comme son nom l'indique).

Les diagrammes de séquence et les diagrammes de collaboration sont aussi appelés des diagrammes d'interaction (d'objets).

Les diagrammes de cas d'utilisation

Les diagrammes de cas d'utilisation servent à décrire un système du point de vue de l'utilisateur. Il faut par contre dire, qu'un utilisateur ne doit pas forcément être un utilisateur humain. Différentes possibilités existent :

- Utilisateurs humains
- Logiciels externes
- Systèmes externes
- etc.

Figure 26 : Utilisateurs

Un utilisateur est aussi appelé acteur et la représentation d'une telle entité en UML est réalisée par un petit symbole de bonhomme.

La représentation d'un cas d'utilisation en UML s'effectue à l'aide d'une ellipse à l'intérieur de laquelle est inscrit le nom du cas d'utilisation.

L'interaction (la relation) entre l'acteur et le cas d'utilisation est symbolisée par une flèche avec une ligne solide et une pointe simple. La pointe s'applique uniquement au cas où l'interaction n'est pas bidirectionnelle. La figure suivante montre un diagramme de cas d'utilisation simple :

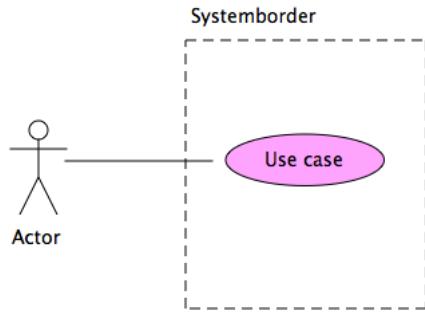


Figure 27 : Cas d'utilisation simple

Comme il était possible d'établir des relations entre des classes, il est aussi possible d'établir des relations entre des acteurs ou bien des cas d'utilisation :

- Pour les acteurs, seule la généralisation est possible. Si un acteur hérite d'un autre, ça veut dire que celui-ci aura accès à tous les cas d'utilisation de l'acteur dont il hérite. En plus, il peut accéder à des cas d'utilisation auxquels l'acteur dont il hérite n'a aucun accès. En UML, cette situation est représentée par une flèche de généralisation.

Figure 28 : Relations entre acteurs

- Les cas d'utilisation peuvent hériter l'un de l'autre. Ce qui veut dire qu'un cas d'utilisation hérite du comportement d'un autre et qu'il a la possibilité de modifier celui-ci ou bien qu'il peut ajouter des comportements spécifiques à ceux dont il hérite.
- De plus, les cas d'utilisation peuvent inclure un autre cas d'utilisation. Dans cette situation, le cas d'utilisation qui inclut l'autre peut utiliser le comportement de cet autre mais ne pas le modifier. Cette relation est représentée en UML par une flèche de dépendance qui porte le stéréotype "include". Cette technique permet de factoriser un comportement commun.

Figure 29 : Relations entre cas d'utilisation

La figure suivante montre un diagramme de cas d'utilisation qui utilise les techniques qui viennent d'être décrites :

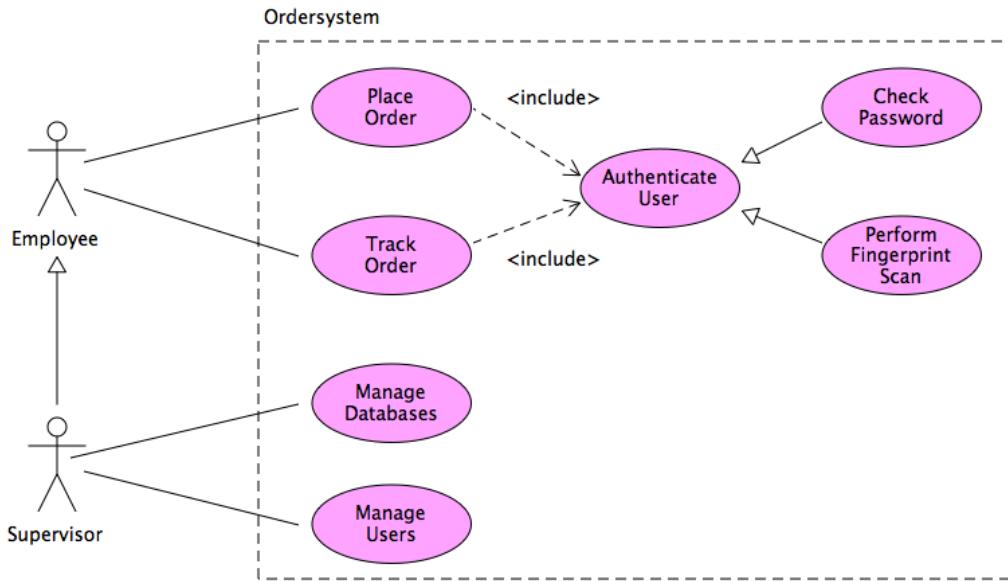


Figure 30 : Un cas d'utilisation complexe

Comme on peut le voir, l'utilisateur `Employee` a un accès aux cas d'utilisation `Place Order` et `Track Order`, il doit par contre être un utilisateur validé. Le `Supervisor` est capable d'effectuer tous les cas d'utilisation de `Employee`, mais il a en plus accès à la gestion des utilisateurs et aux bases de données.

Scénarii

Pour la spécification détaillée des cas d'utilisation, UML propose les diagrammes d'interaction. Pour savoir comment utiliser ce type de diagrammes, il faut d'abord introduire un autre terme, à savoir le terme scénario. Un scénario est un déroulement possible d'un cas d'utilisation particulier. La figure suivante montre comment on doit imaginer des scénarios :

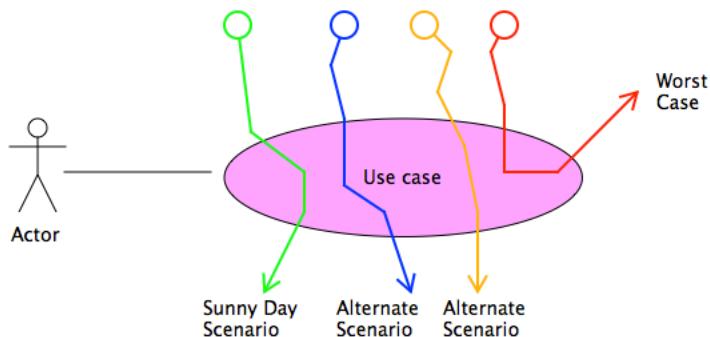


Figure 31 : Scénarios

Un scénario décrit comment une certaine tâche est effectué par la collaboration entre un certain nombre d'objets. Il est donc clair qu'un scénario a seulement une vraie signification quand il introduit de nouveaux objets ou bien quand il montre le déroulement d'un cas d'utilisation qui est vraiment différent des autres déroulements possibles. Pour chaque scénario, les éléments suivants doivent être spécifiés :

- Le nom du scénario
- Les conditions initiales
- Les actions qui sont effectuées pendant ce scénario
- Les conditions finales qui sont atteintes par les actions susmentionnées

Figure 32 : Eléments d'un scénario

Les actions qui mènent des conditions initiales aux conditions finales sont les messages échangés entre les objets. Ces objets et les messages mentionnés sont représentés en UML sous la forme de diagrammes d'interactions, c'est-à-dire dire sous la forme de diagrammes de séquence ou de collaboration. Les prochaines deux sections traitent ces deux types de diagrammes.

Diagrammes de séquence

Les diagrammes de séquence montrent la mise en œuvre d'une tâche au moyen de messages échangés entre les objets. Pour la représentation des objets, on utilise celle qui était montrée dans une des sections antérieures, pour les messages, des flèches avec des pointes simples sont utilisées. De plus, des messages asynchrones (événements) sont représentés par des flèches légèrement pentues. La figure suivante illustre un diagramme de séquence :

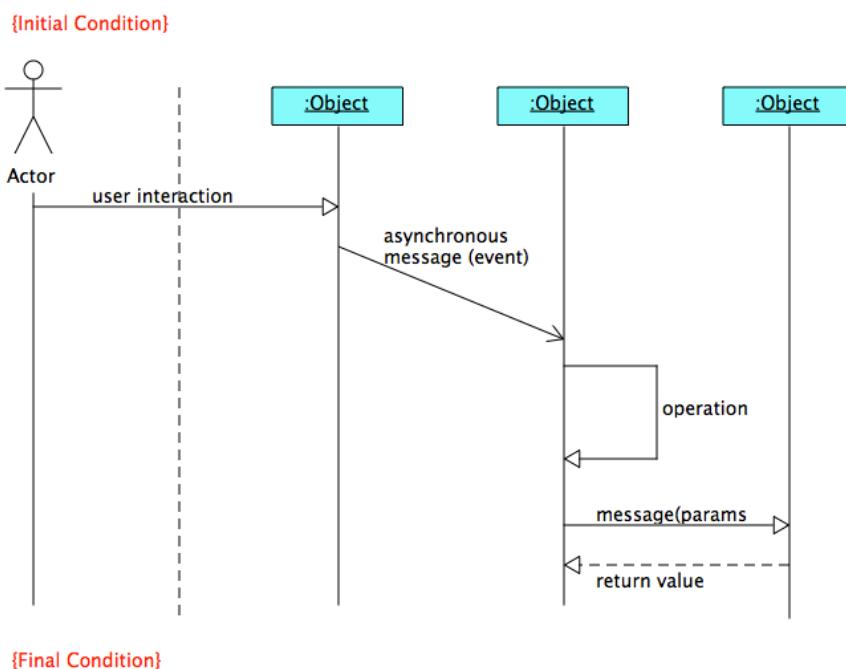


Figure 33 : Diagramme de séquence général

On remarque, que les conditions initiales et les conditions finales sont ajoutées au diagramme à l'aide de contraintes. Des contraintes sont souvent donnés sous forme de texte et doivent être respectées plus tard quand le programmeur implémente le comportement.

Il faut aussi remarquer que l'on peut déduire de ce diagramme de séquence les classes correspondant aux objets ainsi que leurs méthodes. Les messages sont toujours des méthodes de la classe de l'objet vers laquelle ils pointent. De plus, si deux objets communiquent entre eux, on peut dire qu'il doit exister un lien entre les deux et que, par conséquent, leurs classes ont une relation sous la forme d'une association. Les figures suivantes montrent un diagramme de séquence réel, le diagramme d'objets et le diagramme de classes correspondant :

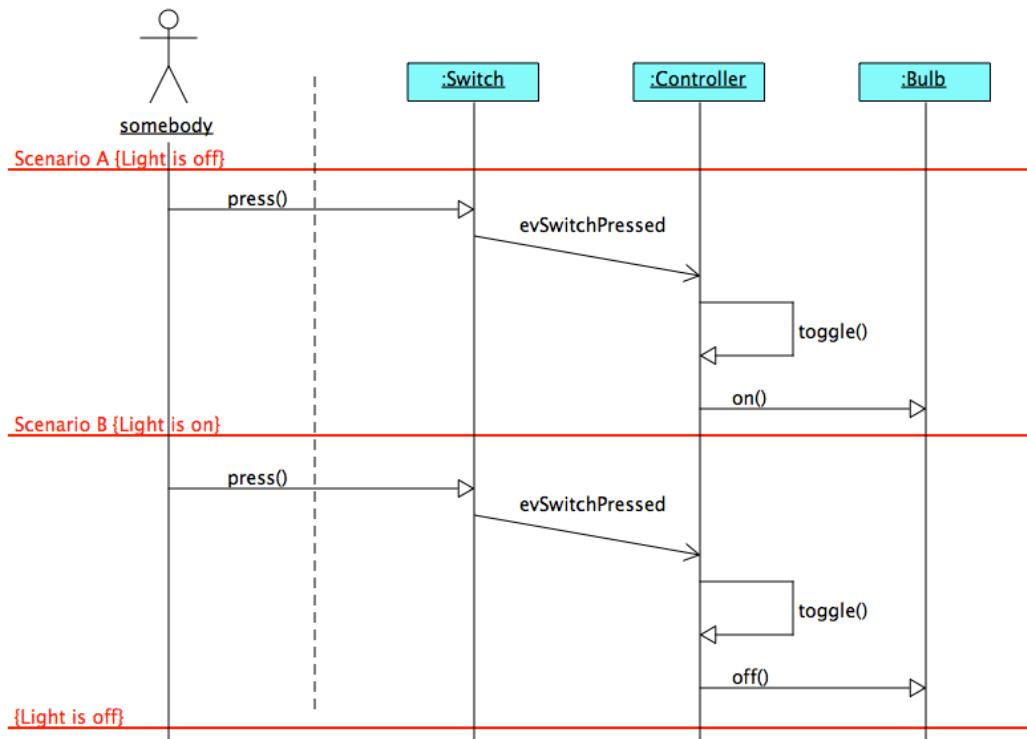


Figure 34 : Diagramme de séquence réel



Figure 35 : Diagramme d'objets

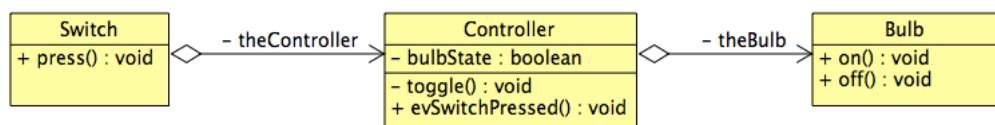


Figure 36 : Diagramme de classes

Quelques explications : Les messages du diagramme de séquence deviennent des méthodes des classes des objets. La méthode `toggle()` de la classe `Controller` est privée parce que c'est la classe elle-même qui l'appelle. De plus, le lien de l'objet `s` qui pointe vers l'objet `c` est une instance de la relation entre la classe `Switch` et la classe `Controller`. La même chose peut être dite pour le lien entre l'objet `c` et l'objet `b`.

Il ne faut pas non plus oublier que les relations `theController` et `theBulb` vont devenir des attributs privés avec un accesseur public dans les classes `Switch` et `Controller`. Le diagramme de classes suivant montre comment :

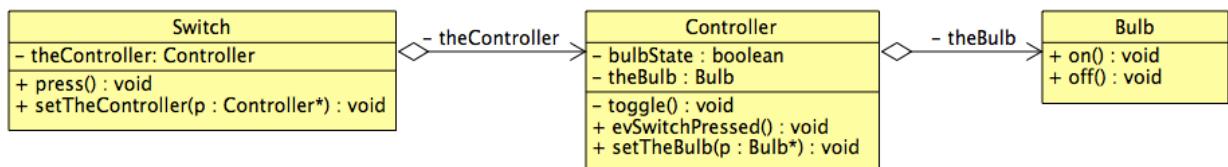


Figure 37 : Diagramme de classes étendu

Il ne faut pas non plus oublier qu'il faudrait encore définir en quelque part une classe composite qui instancie les trois objets et qui les lie.

Diagrammes de collaboration

Les diagrammes de collaboration ont exactement la même signification que les diagrammes de séquence : ils servent à spécifier des scénarios. Seulement la représentation est différente : ils montrent mieux les liens entre les objets le long desquels se propagent les messages échangés entre les objets. Par contre, ils montrent moins l'aspect séquentiel ou temporel. La figure suivante montre le diagramme de séquence précédent sous la forme d'un diagramme de collaboration :



Figure 38 : Diagramme de collaboration

Pour mieux pouvoir montrer le séquencement dans les diagrammes de collaboration, il existe la possibilité de numérotter les messages. La figure suivante montre un exemple concret :

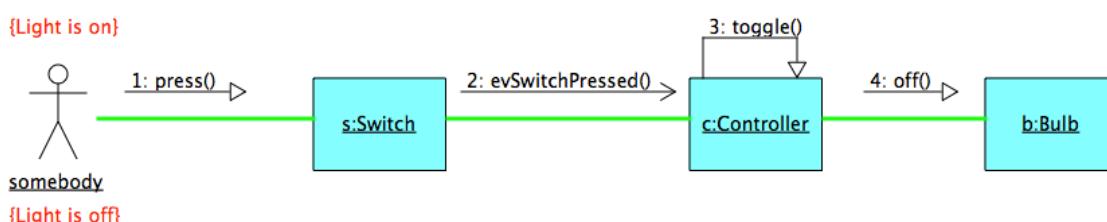


Figure 39 : Messages numérotés

Les figures suivantes montrent des variantes de modélisation supplémentaires comme factoriser et réutiliser des séquences ou bien des séquences alternatives.

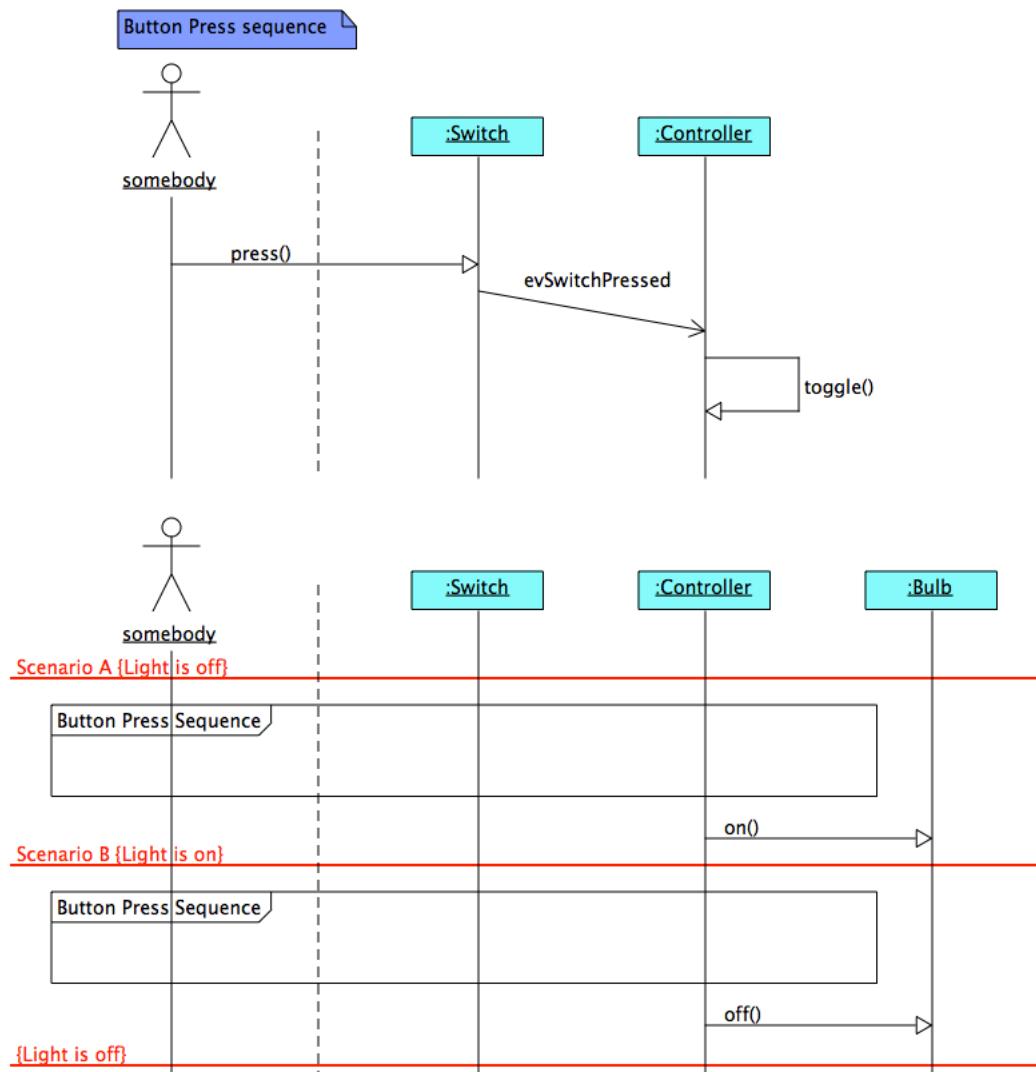


Figure 40 : Factoriser des séquences communes

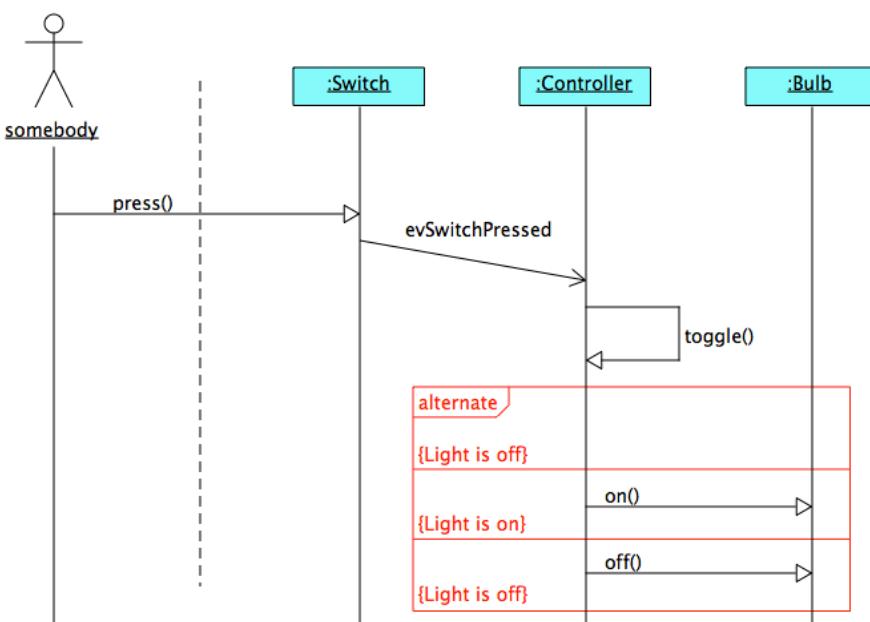


Figure 41 : Des séquences alternatives

Exercices

Les tâches suivantes réfèrent un système d'ascenseur à deux étages.

- Un passager d'un ascenseur peut prendre l'ascenseur. Le technicien peut faire de même, mais en plus il peut arrêter une alarme qui éventuellement a été déclenchée par un passager. Dessinez le diagramme de cas d'utilisation qui représente cette situation.
- Développez le scénario "Monter au premier étage" avec des objets des classes Actor, Button, Bulb, Sensor, Engine, Door et Controller.
- Développez le scénario (les scenarii) "Appeler l'ascenseur" avec des objets des classes sou mentionnées.
- Déduisez un diagramme de classes à partir des scénarii trouvées.
- Dans la section diagrammes de classes vous avec modélisé un véhicule. Développez en utilisant les classes de cet exercice le scenario "Construire et vendre une voiture".

Figure 42 : Exercice

Diagrammes UML physiques

La disposition physique d'un système a souvent un impact non négligeable sur le développement de ses composants. Les éléments physiques qui peuvent jouer un rôle et qui doivent être déterminés sont les suivants :

- Hardware, notamment processeurs, taille de mémoire etc.
- Communication, notamment le type et le débit
- Système d'exploitation
- Répartition géographique
- Architecture logique
- ...

Figure 43 : Éléments physiques

Pour représenter la physique d'un système, UML met à disposition les diagrammes de déploiement. Ces diagrammes sont très simples, ils offrent uniquement deux éléments :

- **Le nœud** : un petit cube qui représente un appareil d'un certain type, quel qu'il soit.
- **Le lien** : un trait simple qui indique un lien physique de type quelconque entre deux appareils.

Figure 44 : Éléments de déploiement

Avec ces deux éléments, il est pourtant possible de représenter précisément la physique d'un système. Le diagramme UML suivant montre le principe :

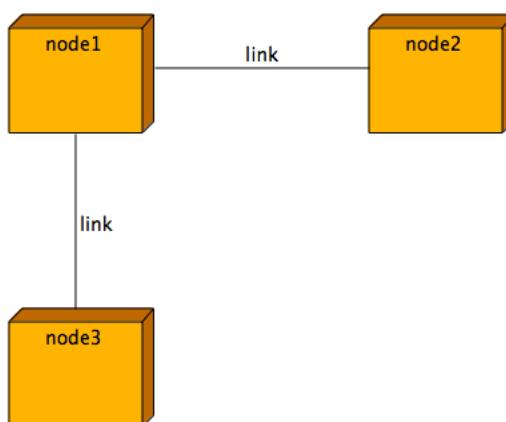


Figure 45 : Diagramme de déploiement

On remarque deux choses dans le diagramme précédent :

- Il y a une différence entre les nœuds ombrés et ceux non-ombrés : les nœuds qui déplacent des logiciels et des tels qui ne déplacent pas. Ceux qui déplacent sont des nœuds actifs avec un processeur, de la mémoire, un système d'exploitation.

Les autres nœuds sont des nœuds passifs comme par exemple un Hub, un souris etc.

- On en déduit qu'il sera plus tard nécessaire de développer du software pour les nœuds actifs, donc il est favorable d'indiquer dans ces diagrammes les caractéristiques nécessaires de chacun des nœuds. Ceci peut être effectué à l'aide de commentaires.
- Dans des systèmes à plusieurs nœuds actives il est très important des spécifier la nature logique ou aussi physique des liens entre ces nœuds, car ces liens cachent très souvent un développement de protocole.
- Depuis des diagrammes physiques, dont le nom anglophone est Deployment Diagram on peut souvent déduire des diagrammes de classes, car les nœuds sont des metaobjets du système. Des metaobjets contiennent d'autre objets respectivement ils sont composés d'autres objets. **Il est donc obligatoire de commencer chaque développement avec un diagramme physique.**

La figure suivante montre un diagramme de déploiement réel :

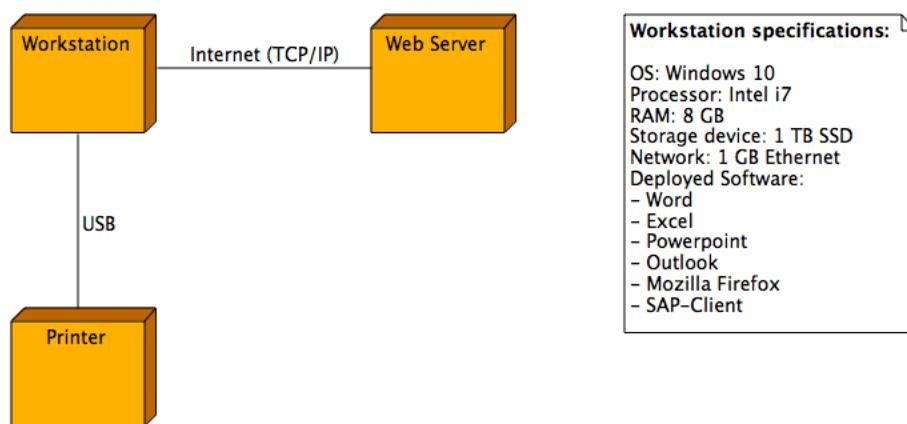


Figure 46 : Diagramme de déploiement réel

Comme on le sent déjà, les diagrammes de déploiement auront plus tard une forte influence sur le développement, parce qu'ils montrent le type et les possibilités des différents composants d'un système. Même si ces diagrammes semblent simples, ils offrent quand même un inventaire d'informations indispensables pour un développement sérieux.

Des diagrammes physiques peuvent aussi contenir des diagrammes de composants. Ces derniers seront expliqués en détail dans le chapitre "Diagrammes UML d'implémentation". Dans des systèmes simples, des nœuds actives sont montrés avec un ou plusieurs diagrammes de composants. Ceci est une alternative pour l'énumération des logiciels déployés par ce nœuds. La figure suivante montre un tel diagramme :

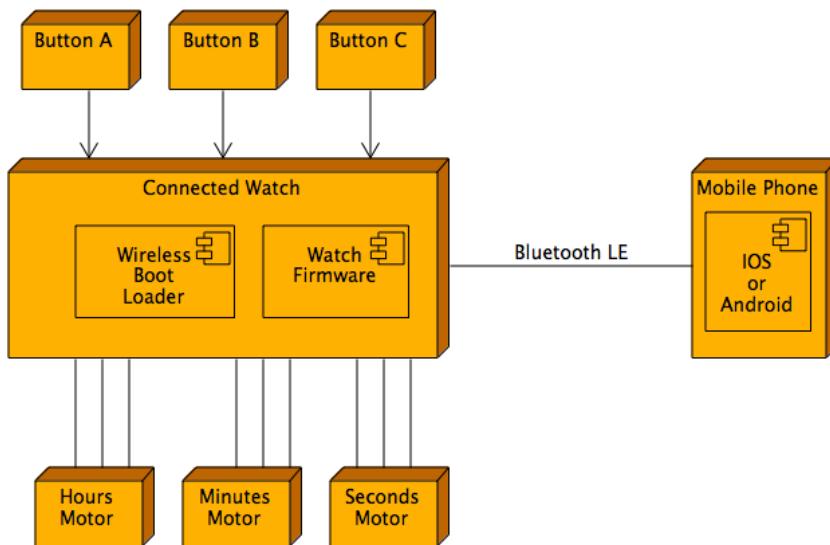


Figure 47 : Diagramme de déploiement réel

Exercices

- Développez le diagramme de déploiement d'un ascenseur à deux étages.
- Dans les trois étages d'un bâtiment sont installés des points d'accès WLAN. La prise téléphonique du bâtiment sort dans les combles. Un routeur / par feu DSL y est installé. Le routeur DSL possède quatre prises ethernet et deux prises USB. Un disque externe de 1 TB est connecté à une des prises USB. Une imprimante est connecté à une des prises ethernet les trois autres ports desservent les points d'accès dans les étages à l'aide d'un câble CAT5. Il y a chaque fois un switch à huit ports. A un port de ceux switch sont chaque fois connectés entre autres les points d'accès. Les utilisateurs peuvent accéder au disque mentionné depuis partout dans le bâtiment. Donnez le diagramme physique de ce système. Réfléchissez quels logiciels sont installés dans quelle appareil.

Figure 48 : Exercices pour diagrammes physiques

Diagrammes UML d'implémentation

Les sections suivantes vont décrire les diagrammes UML qui servent à l'implémentation d'un système ou bien à l'implémentation de ses composants. Ces derniers sont :

- **Diagramme d'états-transitions** : ce diagramme décrit le comportement global d'une classe (d'un objet) sous la forme d'une machine d'états finis
- **Diagramme d'activité** : ce diagramme décrit une méthode d'une classe
- **Diagramme de composants** : ce diagramme décrit un exécutable.

Figure 49 : Diagrammes d'implémentation

Diagrammes d'états-transitions

Les instances d'une classe (les objets) peuvent contribuer à un certain nombre de scénarios. Les scénarios ont été décrits dans une section précédente. Si l'on veut implémenter le comportement global de ces objets de manière à respecter tous ces comportements partiels, il faut prendre en compte tous les scénarios auxquels un objet participe et extraire le comportement "local" pour l'intégrer dans le comportement global. Ceci peut être effectué au mieux à l'aide d'une machine d'états-finis de la classe correspondante, parce que dans ce cas, tous les objets de cette classe particulière vont avoir le comportement décrit par cette machine d'états finis. La représentation d'une machine d'états finis s'effectue à l'aide d'un diagramme d'états-transitions fourni par UML. Les éléments d'un tel diagramme sont les suivants :

- **État** : un objet peut se trouver dans un seul état à un moment donné. Dans cet état particulier, il peut répondre à des événements et effectuer des actions en entrant et en sortant de l'état.
- **Transitions** : les transitions font transiter un objet d'un état à l'autre. Les transitions disposent normalement d'un déclencheur (trigger), typiquement un événement, et d'une condition de garde (guard), typiquement une condition logique. La condition de garde doit être remplie lorsque le déclencheur arrive, sinon la transition n'a pas lieu. Un état peut disposer de multiples transitions entrantes et sortantes.
- **Conditions** : les conditions ou points de décision sont des points avec une transition entrante et au moins deux transitions sortantes. La transition entrante peut disposer d'un déclencheur et d'une condition de garde, les transitions sortantes ont seulement droit à une condition de garde.
- **État initial** : l'état initial est l'état dans lequel se trouve un objet après sa création
- **État final** : quand l'état final est atteint, l'objet est détruit

Figure 50 : Éléments d'un diagramme d'états-transitions

La représentation de ces éléments en UML est montrée dans la figure suivante :

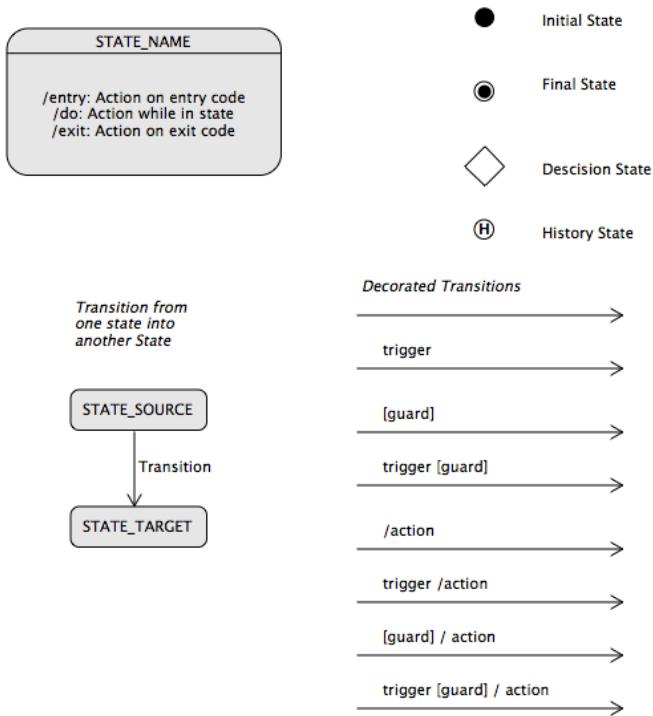


Figure 51 : Représentation des éléments des diagrammes d'états-transitions en UML

Quand une machine d'états transitions est exécuté, la séquence d'actions est comme montrée dans la figure suivante :

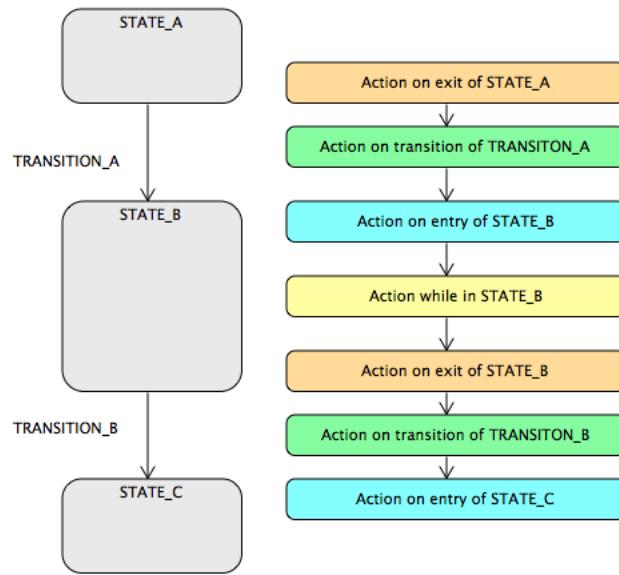


Figure 52 : Séquence d'actions dans une machine d'états-transitions

Une machine à états finis simple est montrée dans la figure suivante :

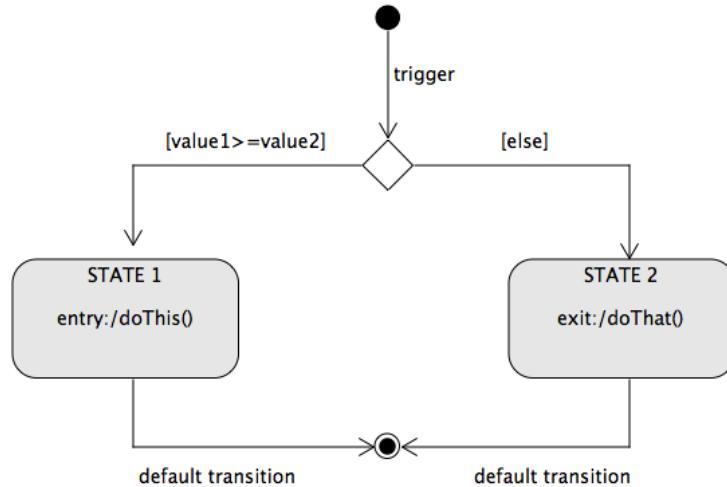


Figure 53 : Une machine à états finis en UML

Un exemple plus réaliste d'une machine à états finis est montré dans la figure suivante. Cette machine d'états reprend l'exemple des diagrammes de séquence montré dans les sections précédentes et modélise la machine à états finis de la classe CPU :

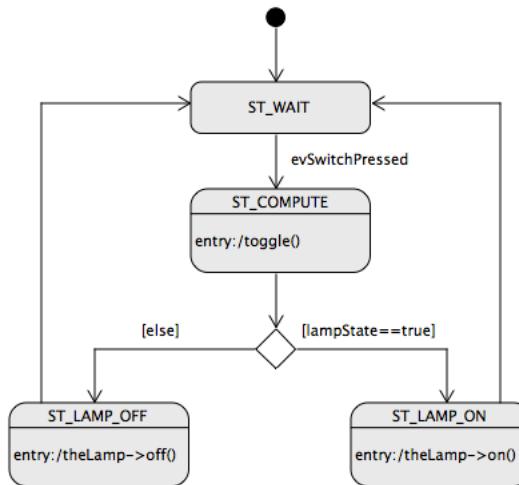


Figure 54 : Diagramme d'états-transitions de la classe CPU

Il faut encore dire qu'une classe spécifique ne peut contenir qu'une seule machine à états finis. Celle-ci peut par contre avoir une complexité très grande et disposer de plusieurs niveaux, exprimés en UML par des diagrammes d'états-transitions imbriqués. Des classes qui héritent d'une autre classe héritent aussi de leur machine à états finis.

Diagrammes d'activité

Les méthodes d'une classe implémentent des comportements partiels des objets de cette classe ou des algorithmes spécifiques pour remplir des tâches spécifiques. Les méthodes peuvent être implémentées de deux manières :

- **Par du code** : ceci est la manière simple et rapide. Elle a par contre le grand désavantage que le modèle devient dépendant d'un langage de programmation et c'est la raison pour laquelle il faut éviter au maximum d'utiliser cette façon d'implémenter.
- **Par un diagramme d'activité** : c'est la manière modèle, elle décrit une méthode sous la forme d'un organigramme dont les nœuds contiennent uniquement des code-actions (des instructions simples) qui sont peu dépendantes d'un langage de programmation spécifique. Toutes les instructions structurées sont exprimées par des éléments graphiques. Il est donc préférable d'utiliser cette manière de faire.

Figure 55 : Manières d'implémenter des méthodes

UML prévoit un certain nombre d'éléments graphiques pour représenter des diagrammes d'activité. Ces éléments sont illustrés dans la figure suivante :

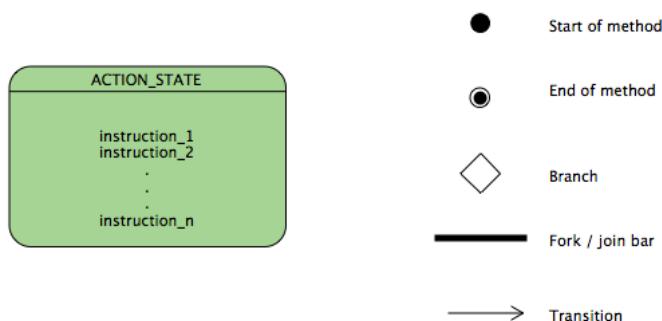


Figure 56 : Éléments des diagrammes d'activité en UML

- **Début et fin de méthode** : ces deux éléments désignent le début et la fin d'une méthode. Ils ont la même signification que le bloc qui délimite une méthode en langage de programmation
- **État d'action** : cet élément contient des instructions simples qui sont exécutées au moment où cet état d'action est entré.
- **Branche** : cet élément permet de laisser continuer le programme selon certaines conditions. Ceci permet la construction de boucles et branchements (simples et multiples).
- **Fork (Join)** : cet élément permet de créer des processus parallèles ou des threads, et de réunir deux processus en un seul.
- **Transition** : Les transitions permettent de définir le chemin de déroulement du programme qui passe d'un état d'action dans un autre.

Figure 57 : Description des éléments des diagrammes d'activité en UML

La figure suivante montre un diagramme d'activité tout simple en UML :

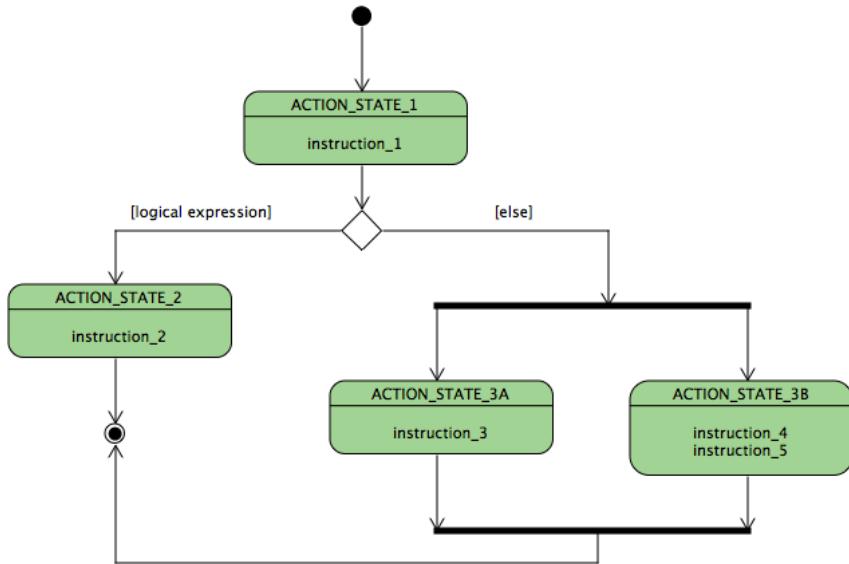


Figure 58 : Diagramme d'activité simple

Remarque : Quand il y a des états d'action en parallèle, le modèle utilisé pour la resynchronisation est normalement celui de "wait to completion", ça veut dire que le processus ou thread plus rapide attend jusqu'à ce que le plus lent soit terminé.

La figure suivante montre un diagramme d'activité réaliste du modèle développé dans la section précédente.

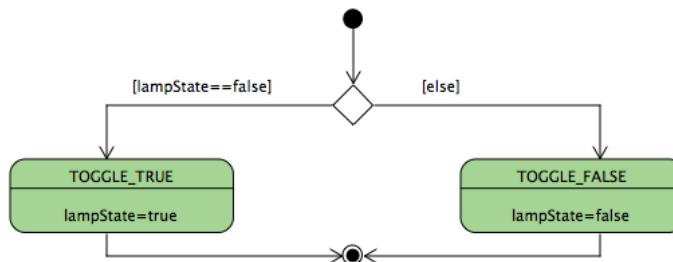


Figure 59 : Diagramme d'activité réel

Comme on peut le voir facilement, il traite du diagramme d'activité de la méthode `toggle()` de la classe `CPU`. Elle change la valeur de l'attribut `lampState` selon sa valeur actuelle.

Diagrammes de composants

Une fois qu'un modèle a été développé, il faut pouvoir en synthétiser les exécutables car tout modèle qui ne peut pas être traduit en code automatiquement n'est pas vraiment utile. Comme un modèle décrit tout un système, il faut qu'un élément permette l'attribution des différents éléments comme des paquets entiers ou des classes simples ou composites à une cible spécifique. Les cibles sont décrites par des diagrammes de déploiement, donc on dispose d'informations comme par exemple le compilateur à utiliser, etc. Le regroupement de toutes ces informations s'effectue à l'aide d'un élément appelé composant. Il cache, au moins dans des environnements de développement automatisés, le paramétrage correct de toute la chaîne d'outils qui crée finalement un exécutable. Comme déjà mentionné, les composants regroupent un certain nombre d'informations utiles à la production d'exécutables. Ces dernières sont :

- Le nom de l'exécutable
- La cible sur laquelle on déploie l'exécutable
- La liste des éléments du modèle qui forment l'exécutable
- Le code d'initialisation pour cet exécutable.

Figure 60 : Informations regroupées par un composant

UML met à disposition un élément nommé composant qui peut contenir cette information. Il est démontré dans la figure suivante :

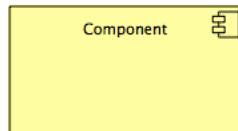


Figure 61 : Diagramme de composant en UML

La manière dont un composant regroupe l'information et comment il accède aux outils de production est spécifique pour chaque outil de modélisation. Il en existent qui n'offrent pas du tout cette option et il en existent qui l'offrent.

Si un outil offre la possibilité de générer un exécutable à partir d'un modèle, un thème très important est l'adaptation des cibles. Ceci comprend tous les travaux de paramétrage des outils de génération de code, du compilateur et du linker etc.

La figure suivante montre encore une fois, comment un composant joue le rôle d'intermédiaire entre le modèle et la cible :

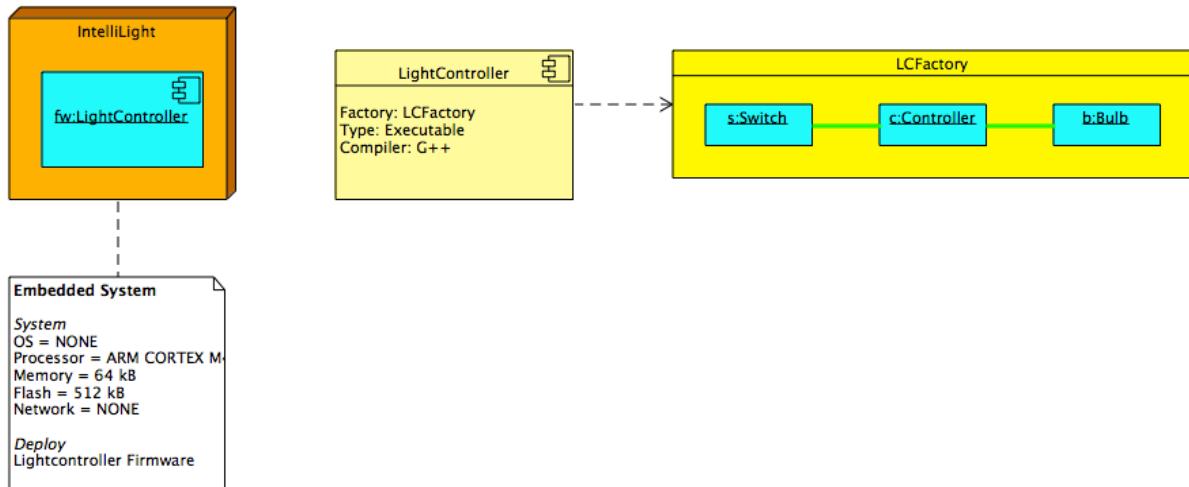


Figure 62 : Le rôle du composant

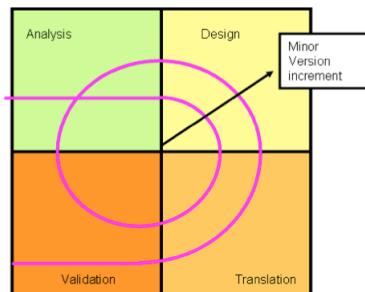
Comme on peut le voir, la classe composite `LightController` sera instanciée par le composant et elle-même va instancier des objets d'autres classes comme `Switch`, `CPU` et `Lamp`. Le code sera généré pour un système embarqué contenant un processeur Java. A partir de l'information du compilateur, il faut aussi disposer d'un outil de téléchargement du fichier exécutable dans la cible embarquée. Mais ceci est une autre histoire.

Exercices

- Modélez la machine d'états-finis de la classe `Controller` de l'ascenseur à partir des scénarios que vous avez développés dans un exercice précédent.
- Modélez le diagramme d'activité de la méthode `solve()` de la classe `P2G`. Elle implémente l'algorithme qui donne la solution d'un polynôme du deuxième degré.

Figure 63 : Exercices pour diagrammes d'états transitions et diagrammes d'activité

Développement basé modèle



L'objectif de ce chapitre est de donner une introduction minimale aux techniques fondamentales du développement de logiciel basé sur des modèles. En outre, les éléments correspondants du langage de modélisation UML sont également décrits pour chaque étape de développement.

Cycle de vie

Par cycle de vie, on entend un processus de développement qui dure de la première idée jusqu'à la mise en service d'un système. Ceci comprend non seulement le développement du matériel et du logiciel, mais aussi tous les aspects économiques ainsi que la formation du client et le suivi après-vente. Dans le cadre de ce cours seront examinés surtout les aspects du développement logiciel.

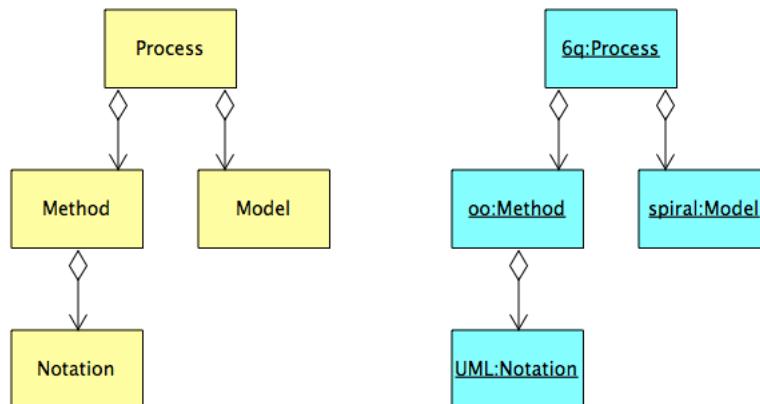


Figure 64 : Processus

La figure ci-dessus montre un processus de développement en général et un exemple concret. Un processus se compose d'une méthode qui en général dispose d'une ou plusieurs formes de présentation (notations) et d'un modèle. Un processus concret est le processus 6q développé par l'unité Infotronique de la HES-SO Valais. Sa méthode est orientée objets, la notation est UML et le modèle est une spirale qui permet un procédé itératif.

Un processus simple

Le processus 6q mentionné dans la section précédente peut aussi être représenté différemment. Une manière populaire est la représentation sous la forme de spirale comme c'est le cas dans la figure ci-dessous. Elle met en évidence la procédure progressive ainsi que les notions de version mineure (Minor Version) et version majeure (Major Version).

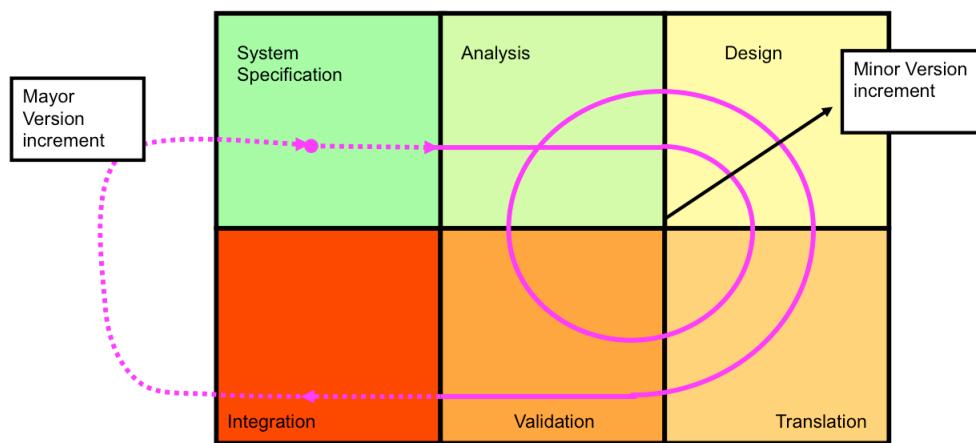


Figure 65 : Représentation d'un processus de développement simple

Une autre représentation est celle de la représentation de phases qui met en évidence le fait qu'il n'y a pas de délimitation claire entre les différentes phases, mais qu'il y a toujours des moments pendant lesquels le développeur se trouve à la fin d'une phase N et au début d'une phase N+1.

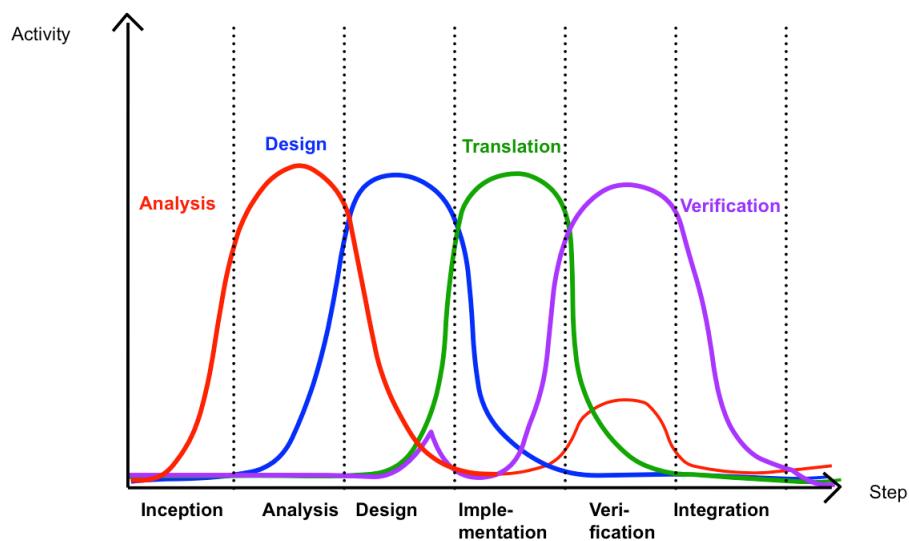


Figure 66 : Représentation d'un processus de développement sous forme de phases

La méthode de représentation la plus importante est toutefois celle ci-dessous, car elle met en évidence la relation entre la méthode de développement (orientée objets) et le modèle (incrémentale). Elle montre aussi quel type de diagramme UML est utilisé pendant quelle des étapes de développement.

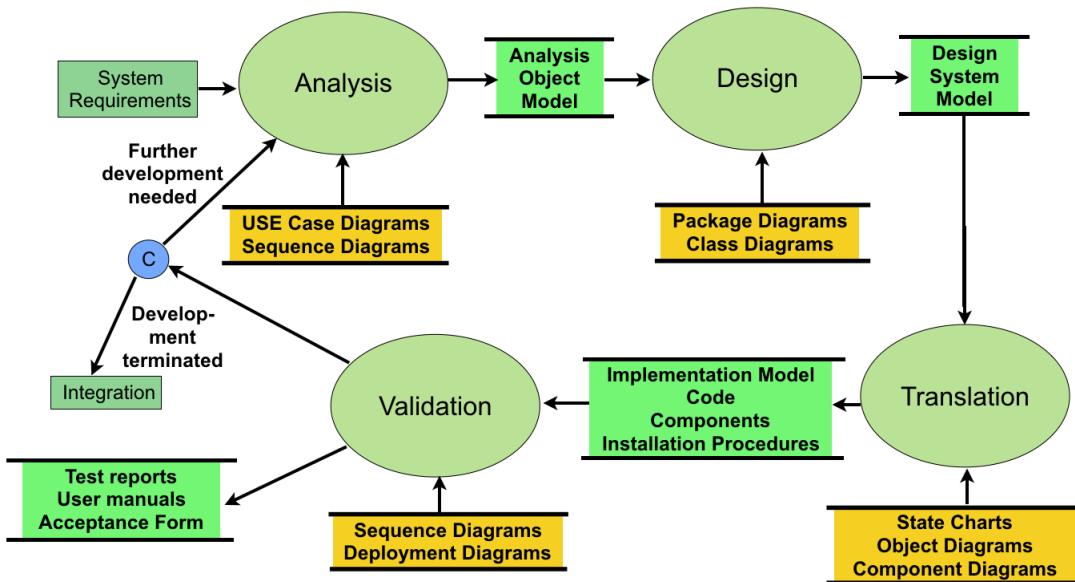


Figure 67 : Représentation d'un simple processus de développement en relation avec les différents diagrammes de la notation UML

Comme représenté dans la prochaine illustration, il n'est pas possible de complètement transposer le monde réel en modèle. Si on voulait y arriver, l'effort nécessaire serait beaucoup trop grand, surtout s'il était mis en relation avec les coûts. En conclusion, un logiciel sera donc toujours seulement une image virtuelle partielle du monde réel dans un processeur ou un contrôleur. Pour finir, c'est une question financière.

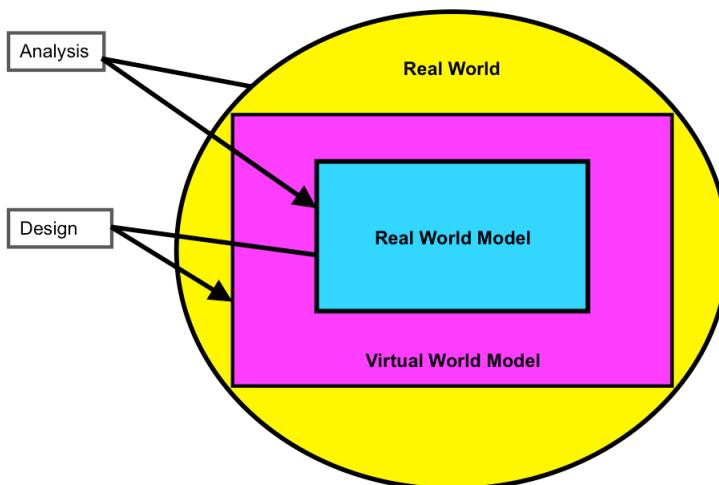


Figure 68 : Monde virtuel et monde réel

Dans la suite, le processus 6q sera introduit à l'aide d'un simple exemple. Le système en question est d'écrit par le client comme suit :

Description du système

Un piège électronique est équipé d'une barrière lumineuse (cellule photo). L'appareil dispose de quatre boutons : ARM, DISARM, RESET et TEST. De plus, il dispose d'un LED capable d'afficher les différents états et il dispose aussi d'une sirène. Il y a aussi un interrupteur principal. Le piège fonctionne de la façon suivante :

Suite à sa mise en marche, le piège se trouve dans un état de base. Après avoir ajusté la barrière lumineuse, l'appareil peut être armé à l'aide du bouton ARM. Cet état est signalé par le LED clignotant lentement. Si maintenant la barrière est offusquée, le piège commence de clignoter rapidement et il joue un son de mise en garde. Cet état doit être quittancé par le bouton RESET sous 10 secondes, autrement le piège se met en état d'alerte. Dans cet état, le LED est enclenché permanent et un son d'alerte est joué. Le bouton RESET permette de retourner en état de base depuis n'importe quel autre état. Si le piège se trouve en état de mise en garde ou en état d'alerte, il ne peut pas être déclenché. Le bouton test permette le test du LED et de la sirène mais seulement en état de base.

Analyse

Cette section décrit les différentes techniques d'analyse et les éléments UML y associés. L'analyse est en principe un mélange de quatre activités différentes :

- Mise en place d'un diagramme physique
- Analyse comportementale
- Analyse structurelle
- Consolidation des différents résultats d'analyse

Diagramme physique

Pour pouvoir s'imaginer la structure physique d'un système à analyser, il est recommandé de toujours en établir un diagramme physique. La figure suivante montre le diagramme physique du système d'écrit ci-dessus :

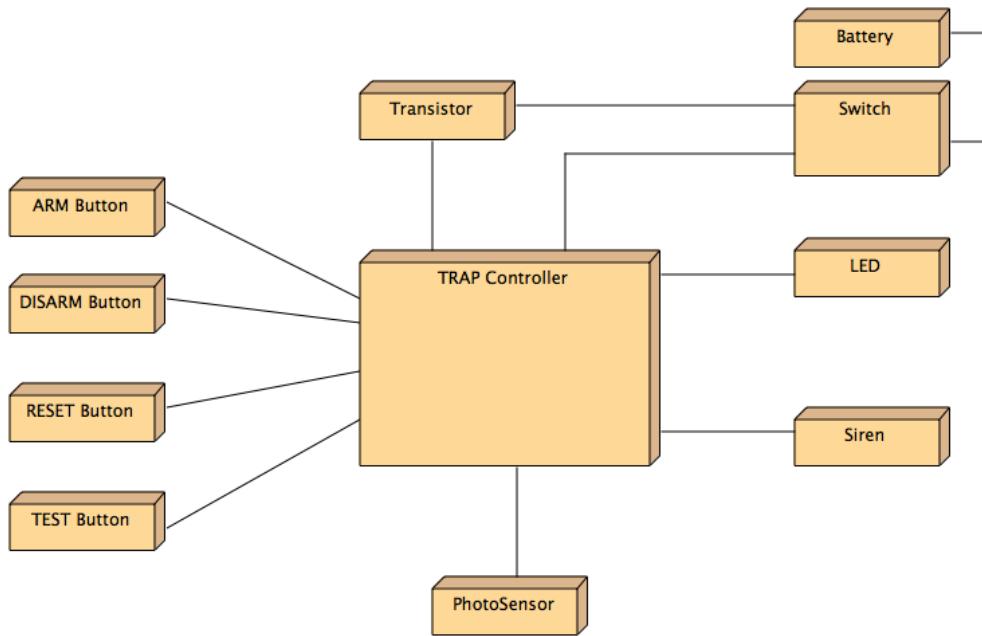


Figure 69 : Diagramme physique du système TRAP

Analyse comportementale

L'analyse comportementale spécifie le comportement d'un système. Elle utilise des cas d'utilisation (USE-case) en UML pour spécifier le comportement d'un système du point de vue d'un utilisateur. Chaque cas d'utilisation est détaillé par des scénarios. Un scénario décrit un déroulement possible et significatif d'un cas d'utilisation. Les images suivantes précisent la représentation des cas d'utilisation dans UML et leur lien avec les scénarios.

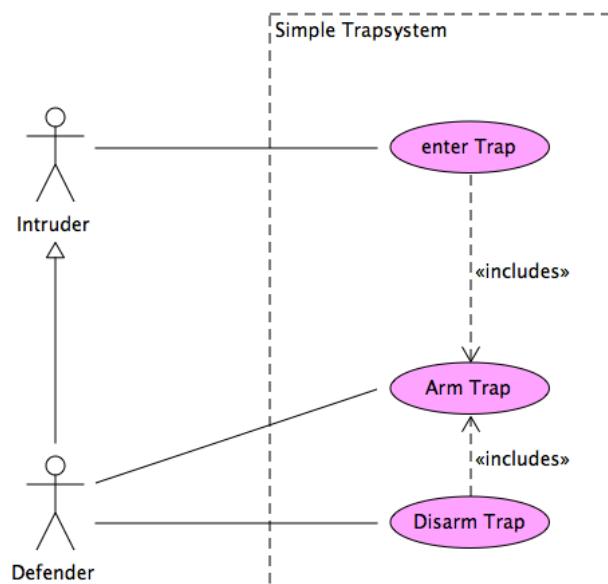


Figure 70 : Diagramme de cas d'utilisation du système TRAP

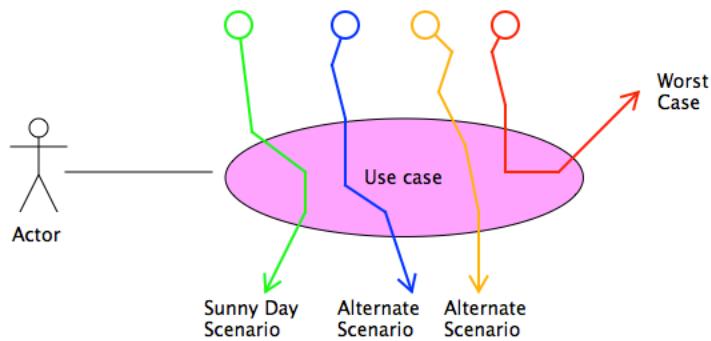


Figure 71 : Pour rappel : Scénarios d'un cas d'utilisation

Les scénarios sont décrits par une condition de départ, un traitement et une condition finale. Le traitement est représenté par des diagrammes d'objets. On utilise soit un diagramme de séquence, soit des diagrammes de collaboration. Ceux-ci montrent des objets qui effectuent une tâche spécifique grâce à l'échange de messages. Des données échangées entre les objets sont aussi des objets. Le fait que des objets échangent des messages entre eux est lié à l'existence de relations entre les classes des objets correspondants. Les images suivantes précisent la représentation des diagrammes de collaboration et de séquence en UML :

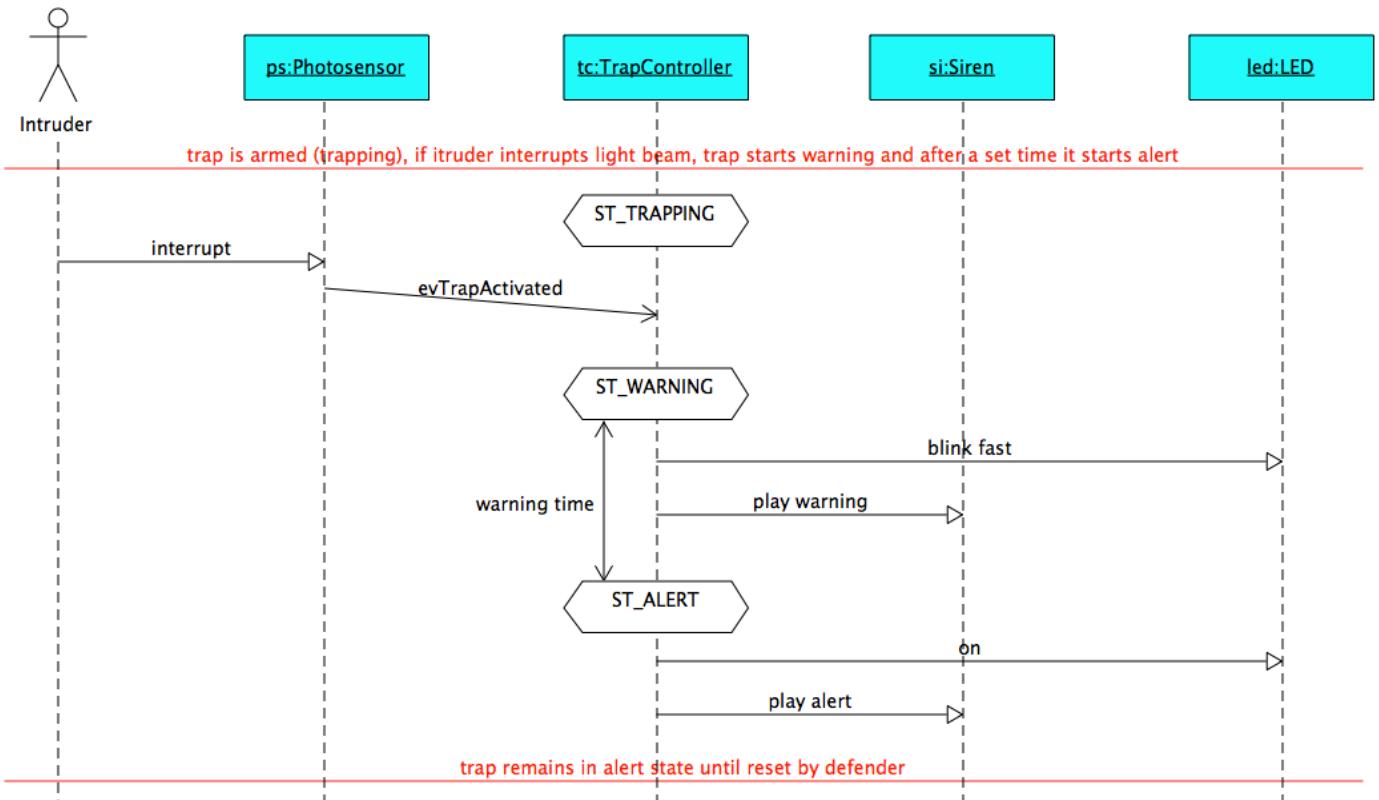


Figure 72 : Diagramme de séquence pour le scénario "activation du piège"

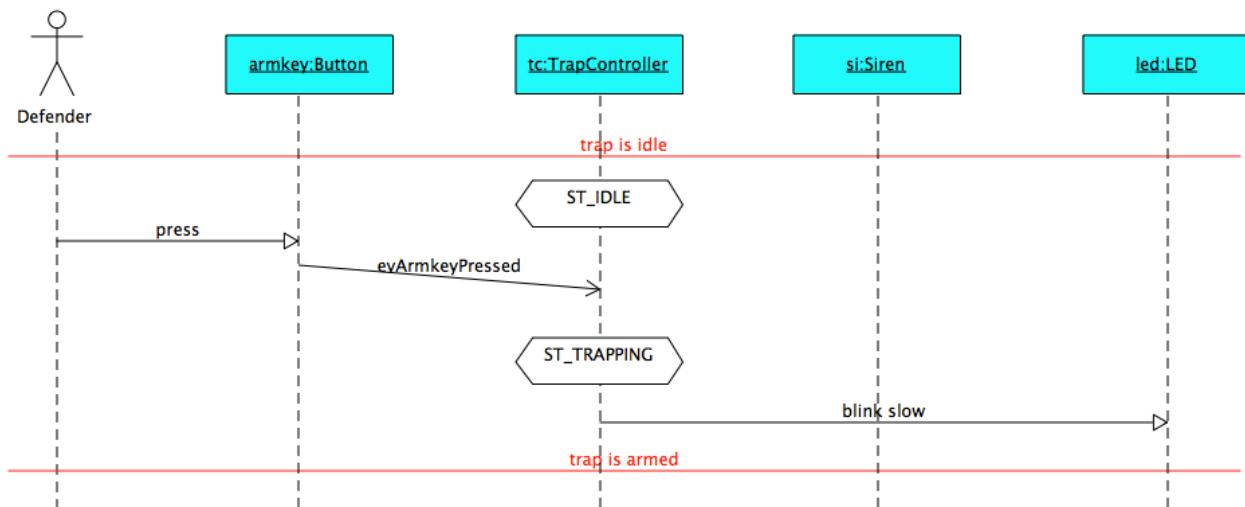


Figure 73 : Diagramme de séquence pour le scénario "Armement du piège"

Un diagramme de classes peut être dessiné comme résultat de l'analyse comportementale. Ceci montre des classes, des méthodes ainsi que des relations entre des classes. Pour pouvoir dessiner un diagramme de classes bien complet, il est nécessaire de saisir un maximum de cas d'utilisations et leurs scénarios correspondants. Dans le cadre du document présent, ceci n'est pas le cas. Né au moins, le diagramme de classes ci-dessous est composé des informations disponibles depuis les deux scénarios montrés ci-dessus :

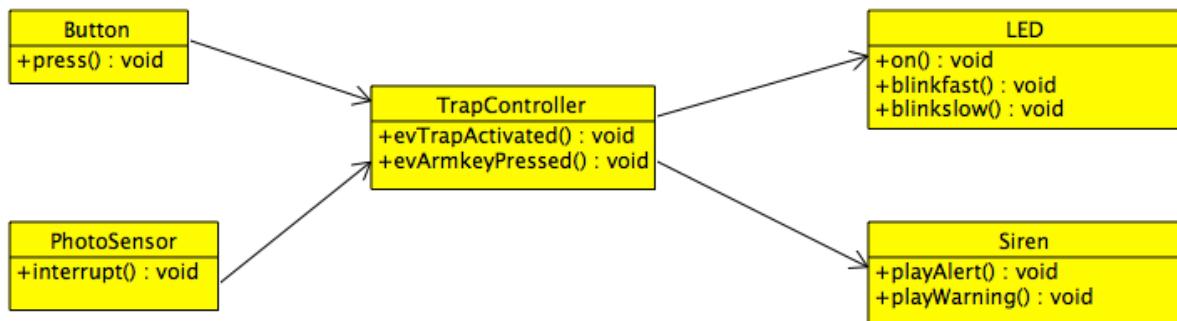


Figure 74 : Diagramme de classes dérivé des informations contenues dans les scénarios

Analyse structurelle

L'analyse structurelle se base sur l'approche qui veut comprendre un système à partir de sa composition. Le résultat de ce genre d'analyse est un ou plusieurs diagrammes de classes, qui montrent les classes, leurs méthodes et leurs attributs ainsi que les relations des classes entre elles. L'image suivante montre la représentation des classes et des relations en UML pour notre système piège :

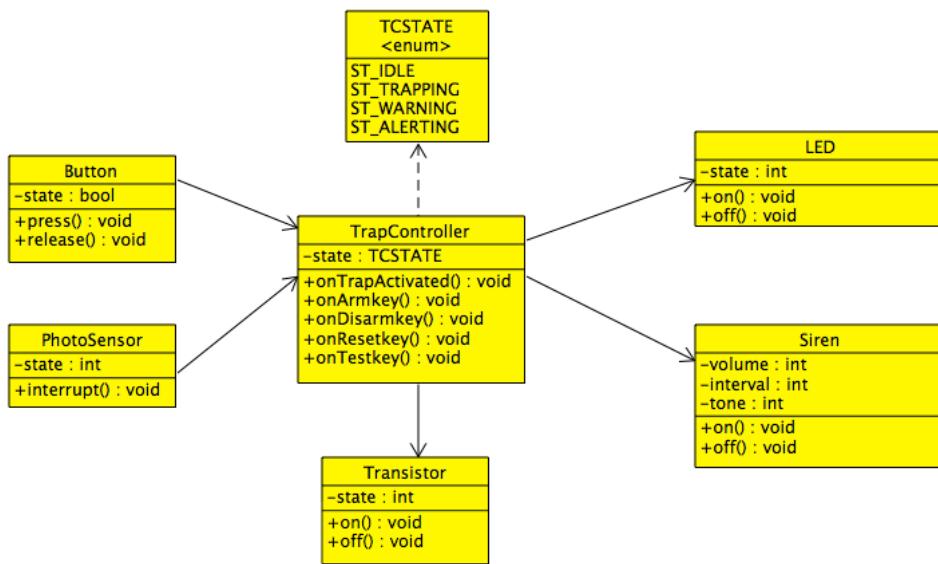


Figure 75 : Diagramme de classes

Les méthodes suivantes sont très souvent utilisées avec l'analyse structurelle :

- **Méthode textuelle** : analyse de la description du système. Des substantifs deviennent des classes, des adjectifs deviennent des attributs et des verbes deviennent des méthodes.
- **Méthode des formulaires** : analyse des formulaires existants. Des champs deviennent des attributs, les processus qui produisent un formulaire deviennent des méthodes. Les attributs et les méthodes sont regroupés d'après des critères logiques dans des classes.
- **Méthode des catalogues** : un catalogue de mots-clés aide à la reconnaissance des objets. Leurs propriétés deviennent des attributs, leurs comportements deviennent des méthodes. Des mots-clés typiques sont : Objets physiques, concepts, etc.

Consolidation des différents résultats d'analyse

Il est clair qu'une analyse comportementale ou une analyse structurelle seule ne livre pas une image complète d'un système, mais que c'est la combinaison des deux approches qui conduit à un bon résultat. La règle suivante est donc valide : La combinaison de l'analyse comportementale avec l'analyse structurelle crée le résultat de l'analyse. C'est donc l'union des deux analyses qui livre les résultats finaux de l'analyse sous forme de diagrammes de cas d'utilisation, de diagrammes de classes et de diagrammes de séquence et de collaboration mis à jour et synchronisés.

La figure suivante démontre schématiquement ce processus :

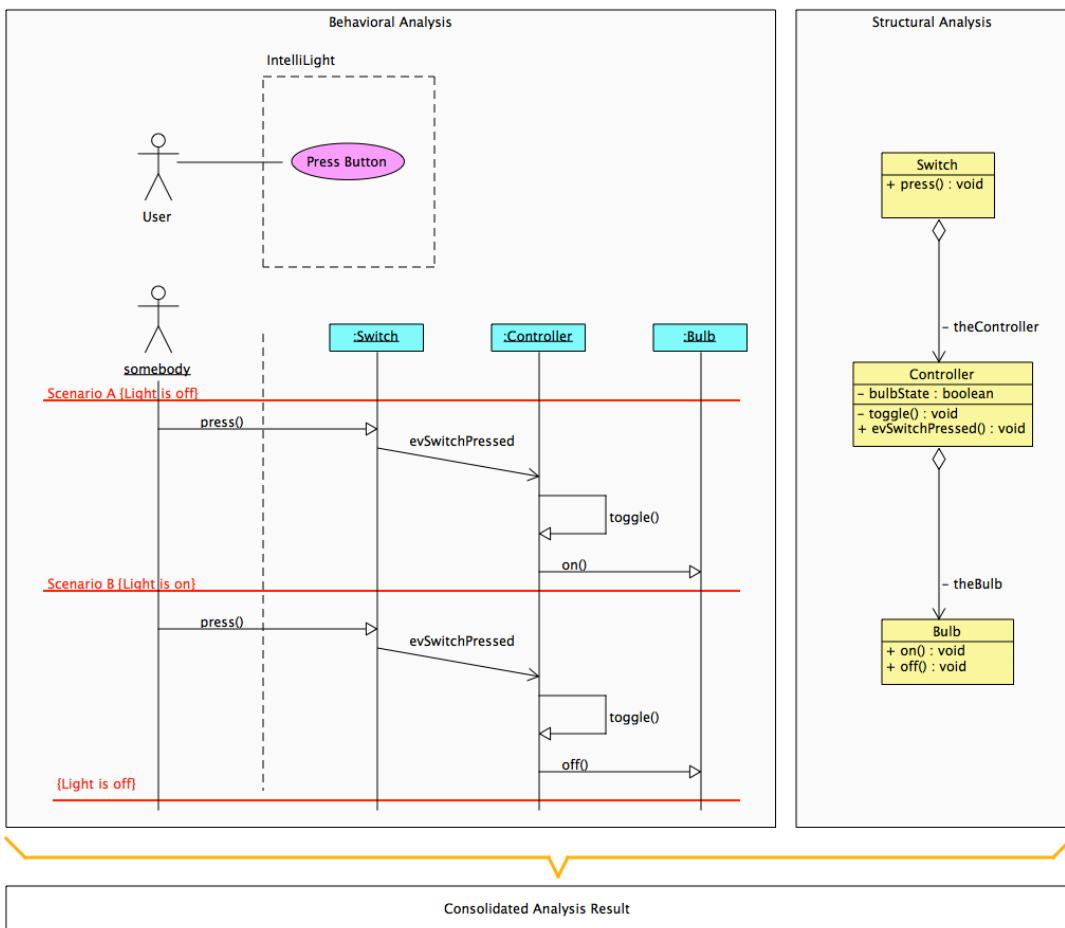


Figure 76 : La genèse du résultat de l'analyse consolidé

La consolidation des résultats d'analyse fournira un nombre de diagrammes de cas d'utilisation, de diagrammes d'interactions et de diagrammes de classes. Les figures suivantes montrent un exemple d'un résultat d'analyse consolidé :

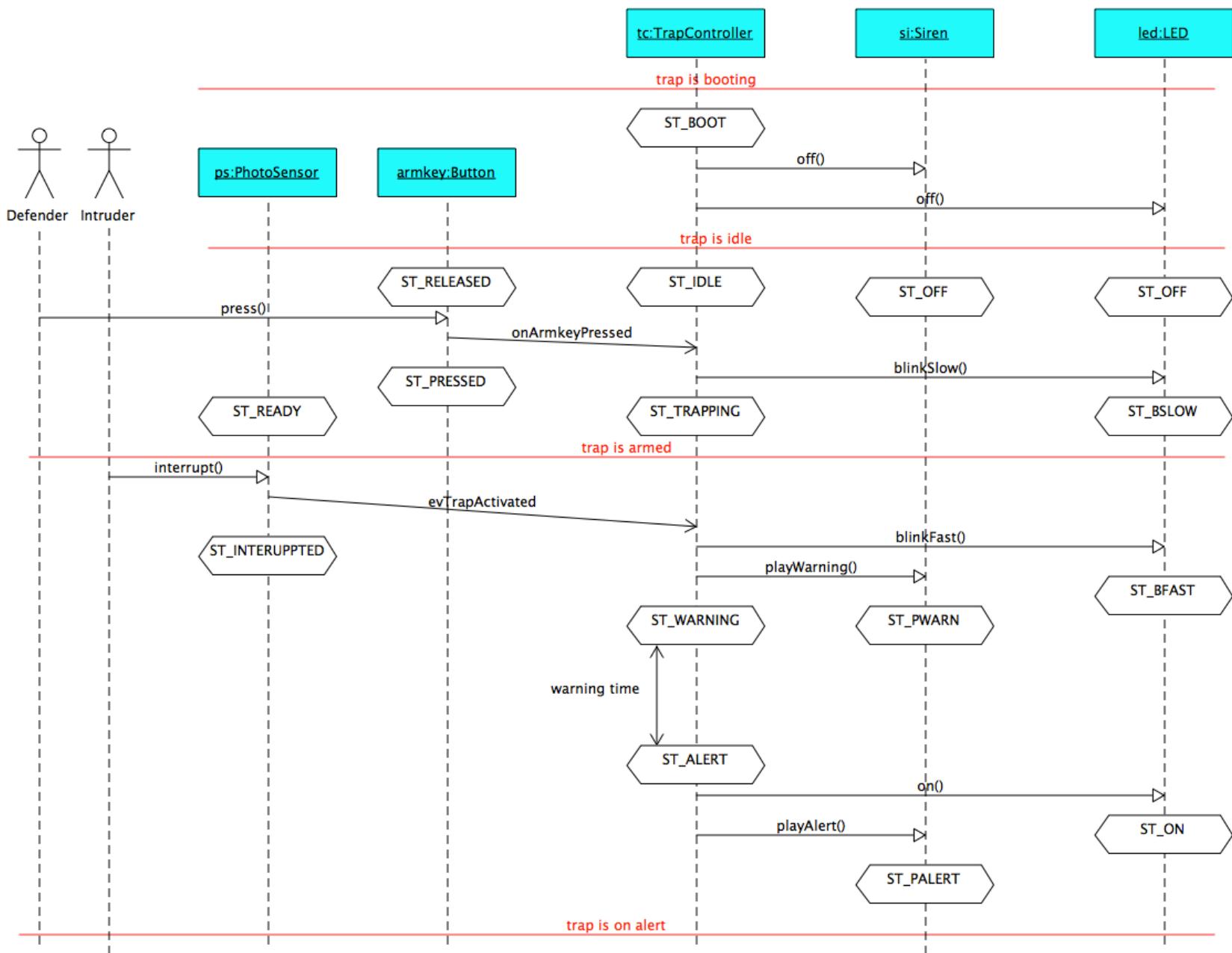


Figure 77 : Diagramme de séquence consolidé

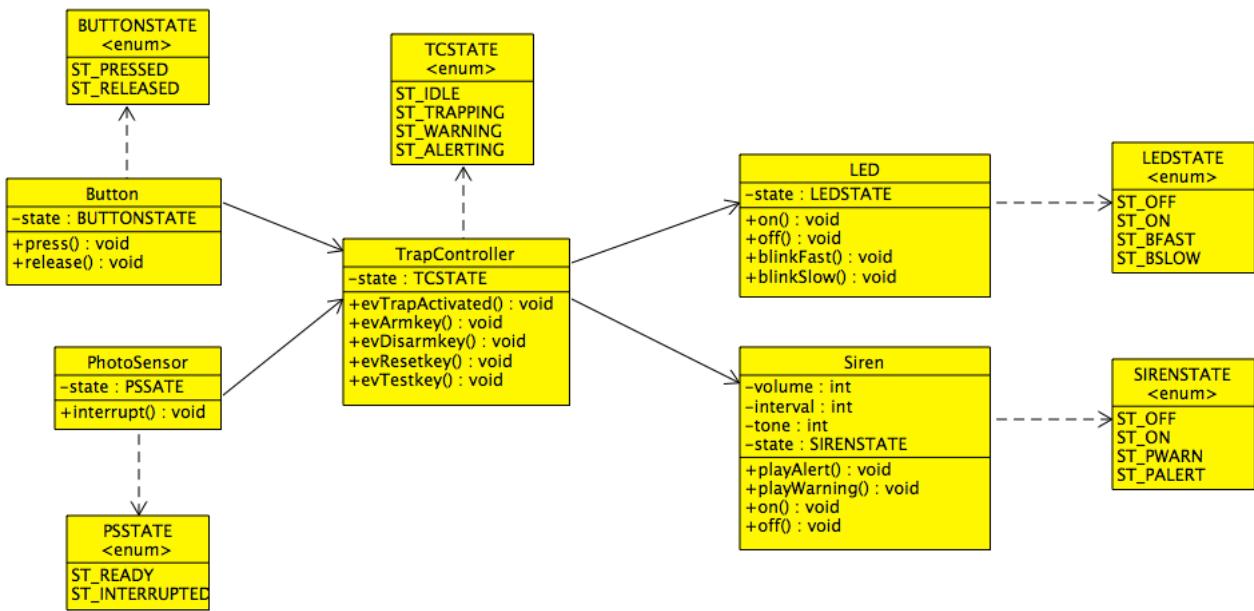


Figure 78 : Diagramme de classes consolidé

Il est clair qu'il faudrait encore ajouter un nombre de diagrammes de séquence décrivant d'autres scénarios du système TRAP. Mais ceci n'est pas réalisé dans le cadre de ce document. En outre, il faudrait aussi ajouter des diagrammes de cas d'utilisations consolidés et bien sûr aussi un diagramme physique consolidé. Dans une prochaine étape du processus 6q, le résultat d'analyse sera maintenant soumis à un procédé de conception.

Conception

Cette section décrit les étapes de la conception. La conception a lieu sous des aspects différents et n'est pas strictement ordonnée. Des étapes de conception possibles pour la plupart des systèmes sont :

- Conception au niveau des composants
- Conception au niveau des classes

Figure 79 : Etapes de conception

L'objectif de la conception est de structurer le modèle plat de l'analyse et de le rendre ainsi implémentable. L'image ci-dessous clarifie cette étape :

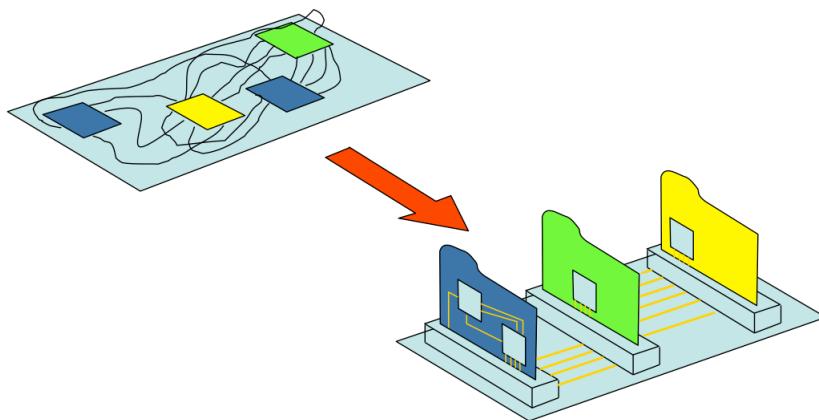


Figure 80 : But de la conception

Les différents aspects sont réalisés par différentes mesures. Les mesures les plus importantes sont détaillées dans les sections suivantes.

Conception au niveau des composants

L'objectif de la conception au niveau des composants est de rendre le modèle d'un composant (d'un logiciel ou d'une bibliothèque) insensible aux modifications apportées par l'environnement du système ou bien du système lui-même. Des telles modifications sont hautement probables un moment donné.

Application du pattern Layered Structure

Les classes sont regroupées par des critères logiques dans des paquets. L'introduction d'une hiérarchie de classes fait également partie de la structuration. Cette mesure améliore considérablement l'efficacité du code généré. Pour structurer un composant ou bien une application il existe un grand nombre de modèles prédéfinis, donc des patterns. La figure suivante montre les classes du piège après l'application du pattern Layered Structure. Considérez les nombreuses relations entre les classes dans les différents paquets. Au lieu de représenter toutes les relations entre les classes des paquets on aurait aussi pu mettre une simple relation de dépendance entre les différents paquets.

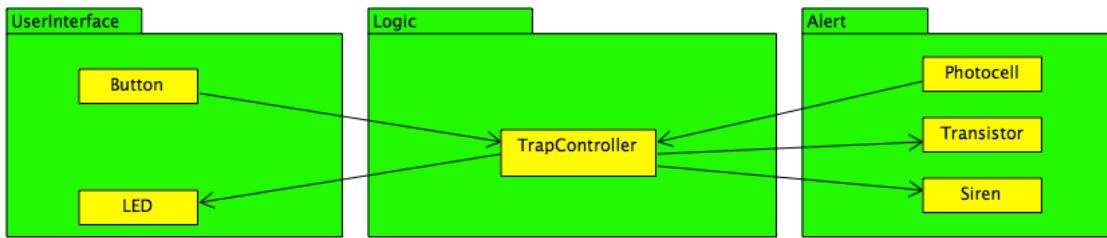


Figure 81 : Structuration au moyen de paquets

Application du Port-Pattern

Ce pattern s'occupe exclusivement de la communication dans le système. Ça commence au niveau des classes avec la communication entre classes, s'étend à la communication entre les différents paquets au moyen d'interfaces et finit par la communication entre les composants au moyen de différents types de réseaux. Généralement, les règles suivantes sont en vigueur :

- A l'intérieur d'un paquet, on communique au moyen d'événements. Avec des classes passives on communique toujours de manière synchrone, donc par des méthodes.
- Entre paquets, on doit communiquer par des appels de méthodes et non pas par des événements
- La communication entre deux paquets devrait être réduite à une liaison entre deux classes
- UML 2.0 a introduit le concept de "porte" (Port). Généralement, une "porte" est une classe qui regroupe un certain nombre d'interfaces sur la base de critères logiques.

L'illustration suivante montre la communication au moyen d'interfaces réalisées par des classes portes. Celles-ci réduisent la communication entre paquets à une seule relation et centralisent l'accès à des éléments externes.

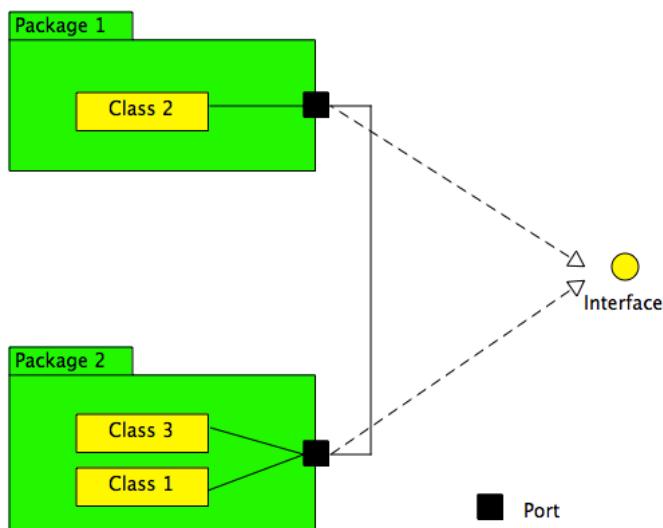


Figure 82 : Communication entre paquets au moyen d'interfaces

Considérez que ce pattern peut réduire les multiples relations des classes des différents paquets à une seule relations entre deux portes.

En plus, des portes ont été utilisées dans ce diagramme. Ce sont les petits carrés au bord des paquets, auxquels ont été attachés au moyen d'associations d'autres portes et d'autre part les classes. Des portes réalisent une ou plusieurs interfaces. Des objets des type porte sont interconnectés par des liens. Ceci signifie que leurs classes sont associées les unes avec les autres.

Les figures suivantes montrent le pattern Port-Pattern appliquée au système de piège. Des interfaces sont définies et ensuite implémentées. De cette manière, une communication bien définie entre les paquets est mise en place. De plus, des relations multiples entre des classes dans des différents paquets sont remplacées par une unique relation entre deux classes de porte.

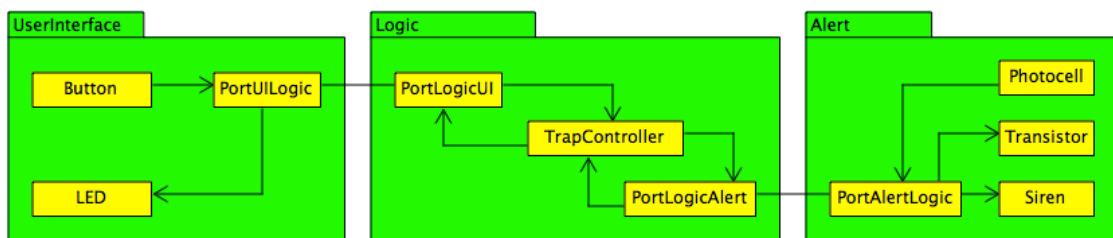


Figure 83 : Redéfinition des relations entre classes "normales" et classes "portes"

Un mécanisme important consiste dans la manière comment les classes portes implémentent une ou des interfaces de façon complémentaire. Une des classes implémente l'interface de façon "required" et l'autre l'implémente de façon "provided".

La figure suivante éclaire ce mécanisme :

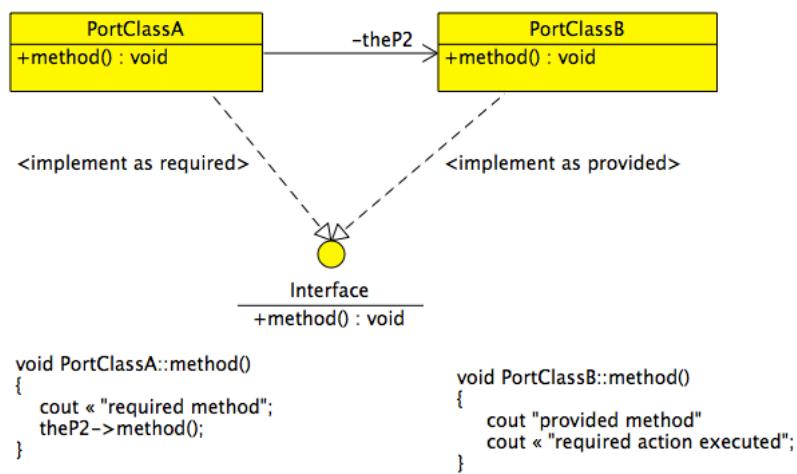


Figure 84 : Implémentation "required" et "provided"

Le pattern Port-Pattern offre donc une communication directionnelle entre deux portes. La figure suivante montre les interfaces ainsi que les classes portes du système piège :

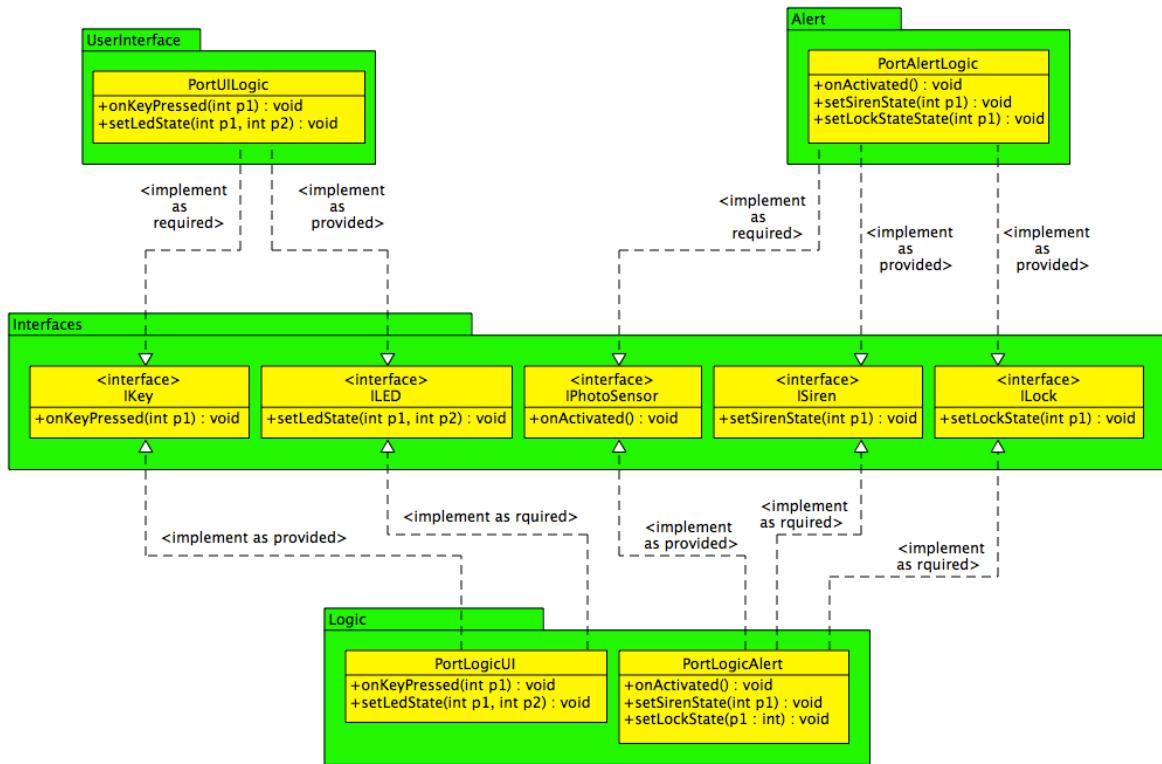


Figure 85 : Interfaces et classes portes pour le système TRAP

Conception relative aux classes

Au moyen de cette mesure, le modèle est stabilisé au niveau des classes. Les points suivants doivent être considérés :

- Visibilité des attributs et des méthodes
- Accesseurs et mutateurs pour les attributs
- Constructeurs
- Destructeurs
- Relations entre classes

Le résultat de ces travaux est un diagramme de classe amélioré et il est éventuellement nécessaire de synchroniser certains diagrammes d'interaction (diagrammes de collaboration ou de séquence). Le système piège ajusté est montré dans l'illustration suivante :

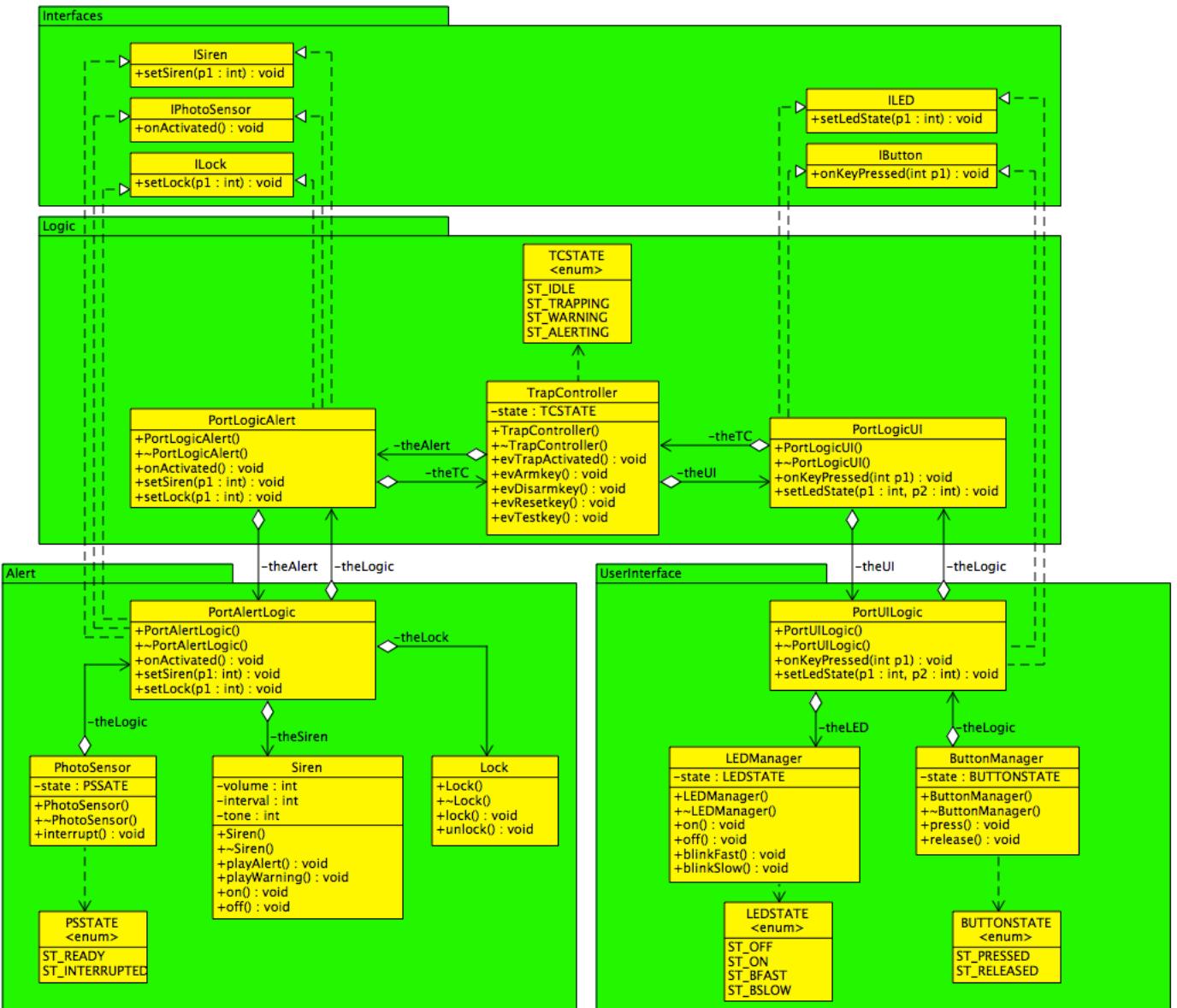


Figure 86 : Diagramme de classes résultant de l'étape de conception

Comme on peut voir dans la figure précédente, les classes ont été conçues de manière systématique : la visibilité des attributs et des méthodes a été réglée, des constructeurs ont été ajoutés et des mutateurs et des accesseurs pour les attributs ont été définis dans les cas où ça semble justifié. De plus, les relations entre les classes ont été détaillées et décorées. Avec cela, la conception au niveau des classes est effectuée de manière correcte.

Finalement, le diagramme de classes avait été optimisé une dernière fois en le connectant à un simulateur du système Trap. Le diagramme de classes ci-dessous montre les modifications qui ont été apportées lors de cette intervention. C'est ce diagramme de classes qui servira comme point de départ pour l'étape d'implémentation qui suivra maintenant.

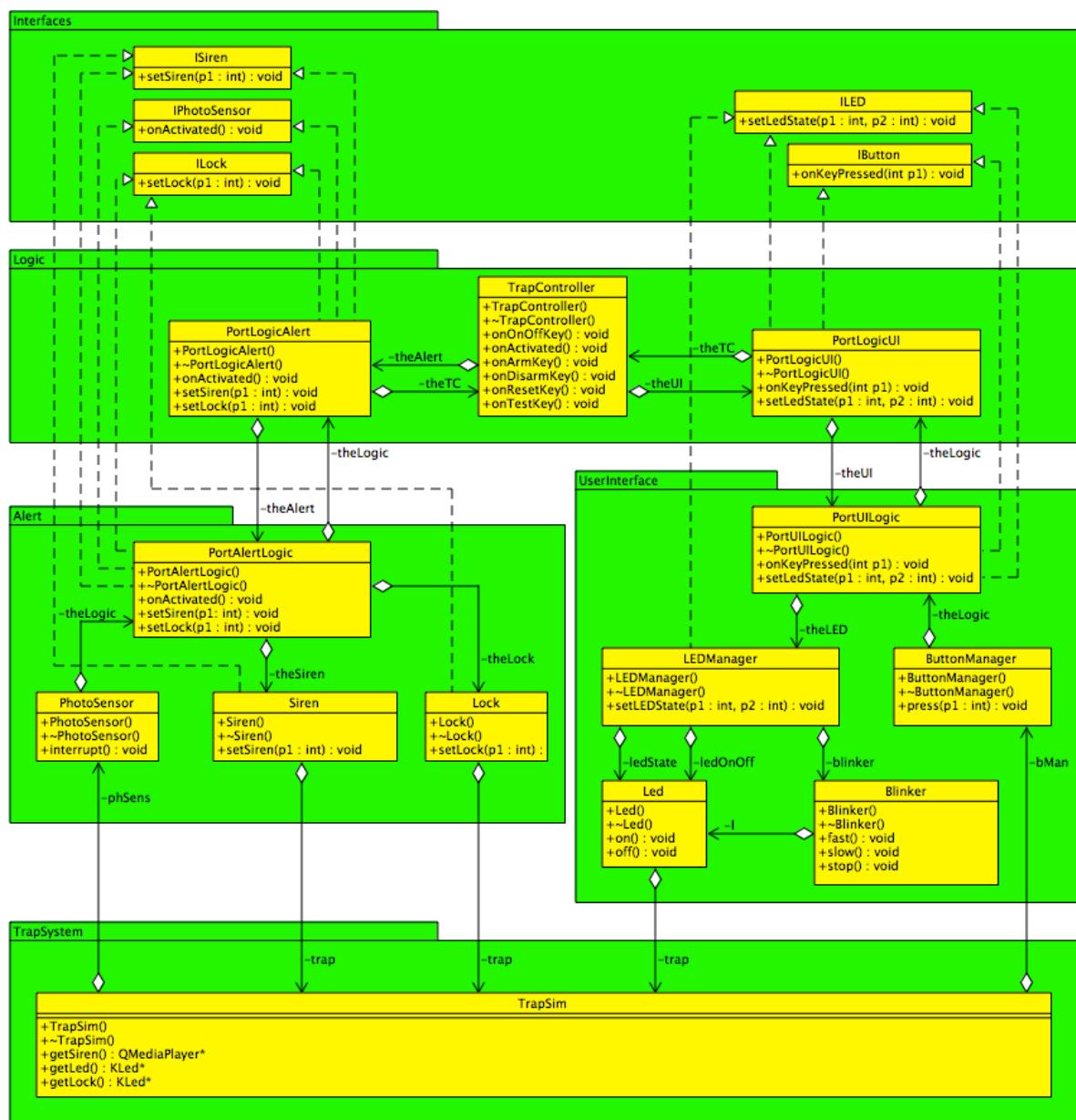


Figure 87 : Diagramme de classes final résultant de l'étape de conception

Ceci signifie aussi la finalisation de l'étape de conception. Le modèle ci-dessus sera implémenté pas à pas de suite. Comment ceci sera effectué est d'écrit dans la section "Implémentation" ci-dessous.

Implémentation

Cette section décrit les éléments les plus importants de la phase d'implémentation. L'implémentation est l'étape par laquelle le modèle universel provenant des phases d'analyse et de conception est instancié pour une cible spécifique. Les activités suivantes sont importantes :

- Implémentation des méthodes (algorithmes)
- Implémentation du comportement (classes, machines d'états transitions)
- Définition et implémentation des composants

Les sections suivantes décrivent brièvement ces activités.

Implémentation des méthodes

Les méthodes des classes implémentent des algorithmes plus ou moins complexes. Des algorithmes sont décrites au moyen de diagrammes d'activités et seront ensuite implémentés. Des méthodes pourraient aussi être décrites uniquement en utilisant du code, mais cela éloignerait considérablement de l'approche modèle qui se veut indépendante de l'implémentation. On écrit du "code-action" seulement dans les états modélisant des actions.

Le code-action est composé d'instructions simples écrites dans le langage de programmation à l'aide duquel le système sera implémenté. UML propose un Action Language, mais il n'est pas beaucoup ou pas du tout utilisé en réalité. Pendant cette étape, la décision du langage d'implémentation doit donc être prise.

Implémentation du comportement

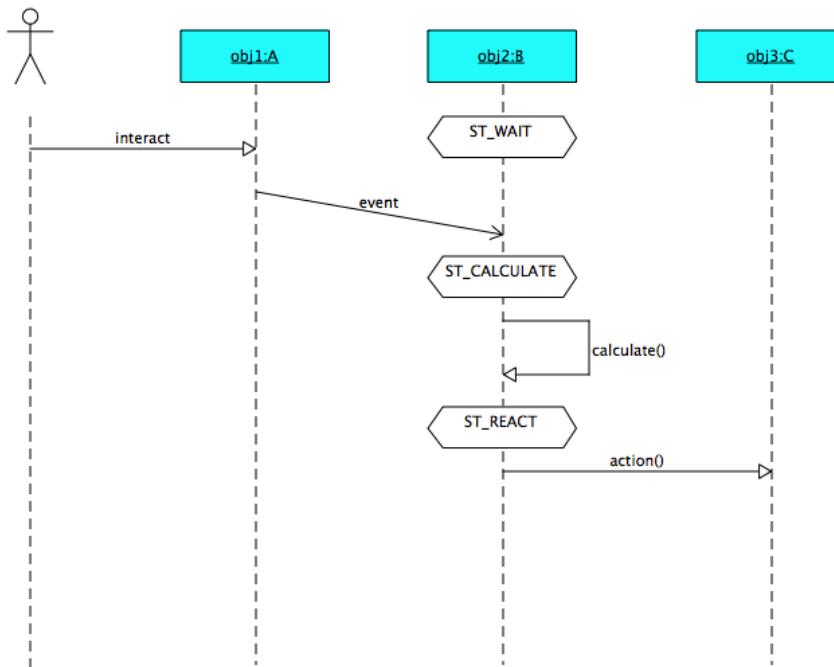
Le comportement d'un système est étudié et décrit par l'analyse. Des diagrammes d'interaction formalisent le comportement du système. Ce comportement est maintenant mis en œuvre au moyen des méthodes de classes et des diagrammes d'états-transitions.

Les classes qui contiennent des diagrammes d'états-transitions sont aussi appelées des classes réactives. Quelques règles importantes lors de la mise en œuvre des diagrammes d'états-transitions sont énumérées ci-dessous :

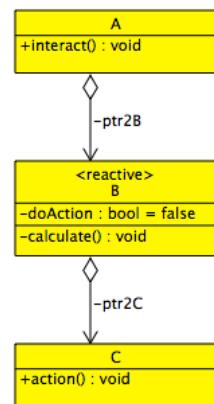
- Des états-actions ne devraient pas contenir d'instructions structurées. Les structures doivent être décrites par les diagrammes d'états-transitions eux-mêmes
- Des états-actions ne devraient contenir que des appels de méthodes. Puisque les méthodes sont décrites par des diagrammes d'activités, une alternative à l'appel de méthode existe dans la définition d'un état imbriqué (Substate)

La figure suivante montre de façon abstraite l'implémentation d'un comportement au moyen d'une machines d'états-transitions.

A) Sequence Diagram



B) Class Diagram



C) Behaviour of Class B (State Chart)

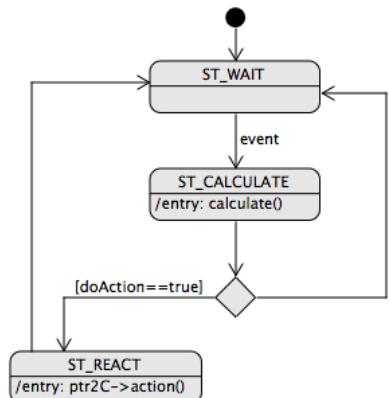


Figure 88 : Déduction du comportement d'une classe à partir des diagrammes d'interaction

Sous A), le graphique montre un scénario de communication entre trois objets. Cette communication est implémentée au moyen de la machine d'états-transitions de la classe B. La communication entre les deux premières classes (A, B) est assurée par des événements (asynchrone), entre la deuxième et la troisième classe (B, C) la communication est assurée par des appels de méthodes (synchrone). Pour celle-ci, il est alors nécessaire de pouvoir disposer d'une association entre les classes concernées. Ceci est montré par le diagramme de classes sous B). La classe B doit donc réagir à des événements, ce qui signifie qu'elle est réactive. Ceci dit aussi que son comportement doit être implémenté à l'aide d'une machine d'états-transitions. Cette dernière est montrée sous C). Remarquez bien que la décision si doAction doit être exécuté ou ne pas est reportée sur le niveau de la machine d'états-transitions est n'est pas caché à l'intérieur d'un état.

En suite de ces considérations générales, la figure suivante montre l'implémentation du comportement de la classe TrapController du système Trap à l'aide d'un diagramme d'états-transitions.

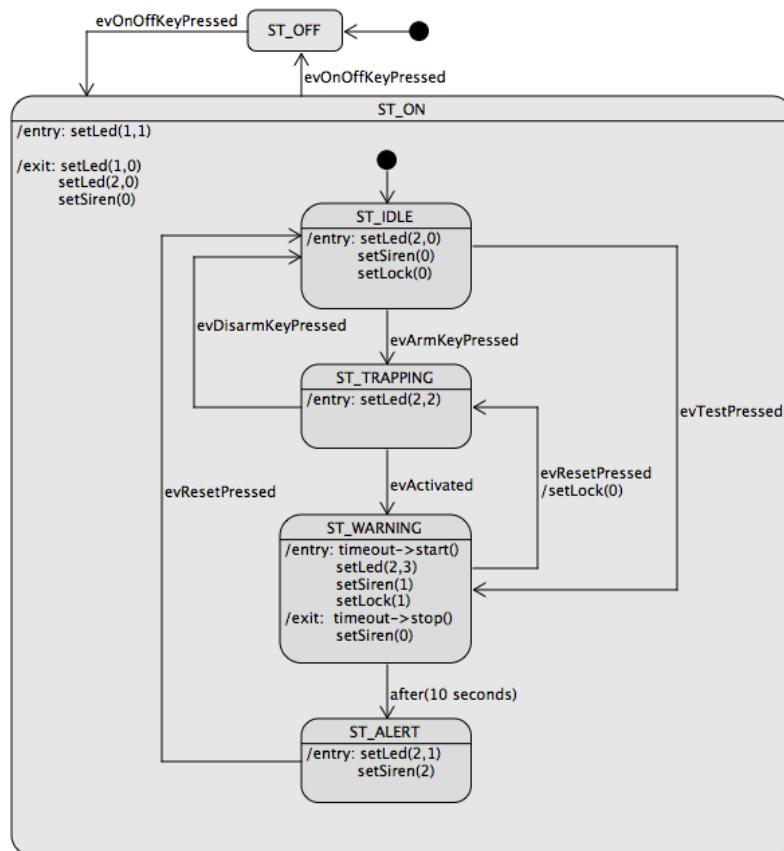


Figure 89 : Machine d'états-transitions de la classe TrapController

Il doit être dit que de dessiner une machine d'états-transitions ne signifie pas encore que cette dernière soit aussi exécutée. Ceci nécessite certaines techniques lesquelles sont énumérées en suite. Celles-ci sont les suivantes :

- **State Machine Pattern** : Ce pattern prescrit, comment un diagramme d'états-transitions est traduit en code exécutable. Un tel pattern par de l'idée qu'il existe une espèce de système d'exécution pour des machines d'états. Un tel système est aussi appelé Execution Framework. Ils y existent des nombreux patterns différents pour la traduction d'un diagramme d'états-transitions en code.
- **Execution Framework** : Un Execution Framework est capable de gérer des événements et d'ainsi faire fonctionner des machines d'états-transitions. Un Execution Framework offre aussi des minuteries qui seront utilisé dans les machines d'états-transitions pour la surveillance temporelle d'événements. Un Execution Framework est aussi brièvement appelé un XF.

Les deux éléments, soit le State Machine Pattern ou soit le Execution Framework ne seront pas décrits ici dans leur dernier détail. Il sont seulement montré à l'aide d'un diagramme de classes. En plus, la collaboration d'une machine d'états-transitions avec le XF sera illustrée à l'aide d'un nombre de diagrammes de séquences.

Le diagramme de classes ci-dessous illustre les éléments les plus importants du State Machine Pattern ainsi que d'un XF. La figure référence le système Trap.

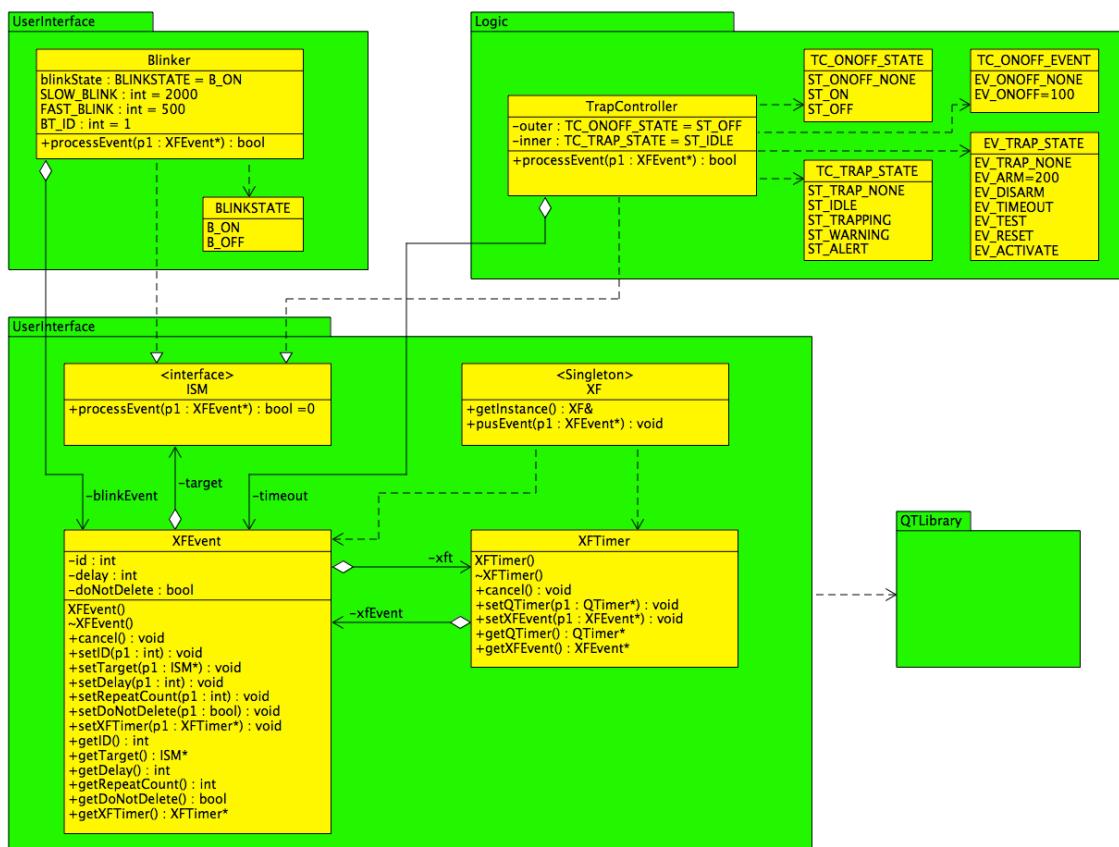


Figure 90 : Les éléments pour l'implémentation des machines d'états-transitions de la classe **TrapController** et de la classe **Blinker**

Voici quelques remarques concernant ces diagramme de classes :

Dans le système Trap il y a deux classes qui doivent réagir à des événements : La classe **TrapController** et la classe **Blinker**. Ceci dit que leur comportement est réalisé à l'aide d'une machine d'états-transitions. Ceci dit aussi que ces classes appliquent le State Machine Pattern. Ceci s'effectue de la façon suivante :

Chaque une des classes réalise l'interface **ISM**. Il est ainsi assuré que ces classes disposent d'une méthode **processEvent**. A l'intérieur de cette méthode, le comportement réactive peut être implémenté de façon plus ou moins libre. Mais dans tous le cas, au moins les éléments suivants devrait figurer dans la classe concernée :

- Une énumération pour les états
- Une énumération pour les événements
- Un attribut pour enregistrer l'état actuel

Dans le cas de la classe **Blinker** il a donc une énumération **BLINKSTATE**, l'attribut **blinkState**, un événement fixe **blinkEvent** ainsi que la méthode **processEvent**. En plus les constantes **SLOW_BLINK**, **FAST_BLINK** et **BT_ID** ont été définies.

Dans le cas de la classe TrapController, il y a les énumérations TC_ONOFF_STATE et TC_TRAP_STATE pour les états, les énumérations TC_ONOFF_EVENT et TC_TRAP_EVENT pour les événements, les attributs inner et outer pour enregistrer les états actuels, l'événement fixe timeout ainsi que la méthode processEvent.

La méthode processEvent est appelée par le XF toujours quand il doit expédier un événement à un objet d'une classe réactive. L'événement lui-même est transporté sous forme d'un paramètre. A l'intérieur de la méthode processEvent, le procéder suivant aura lieu de façon systématique :

A l'aide de l'état actuel, il est décidé si l'événement arrivant pourra être traité ou ne pas. Si oui, la machine d'états -transitions change son état, si non rien ne se passe. La méthode processEvent retourne une valeur booléenne qui indique, si l'événement avait été traité (`true`) ou s'il n'avait pas été traité (`false`). Si l'état change, toutes les actions `onExit` des anciens états et toutes les action `onEntry` des nouveaux états doivent être exécutées.

L'implémentation exacte des machines d'états transitions des classes Blinker et TrapController ne sont pas montrées ici. Ceci dépasserait le cadre de ce document.

Mais il est important de faire un nombre de remarques pour le paquet XF. Il montre les classes les quelles finalement permettent la programmation évènementielle. Il doit aussi être mentionné que ce XF à été implémenté à la base de la librairie QT.

- **ISM** : Ceci est une interface. Toutes les classes qui disposent d'une machine d'états-transitions et qui veulent collaborer avec le XF doivent implémenter cette interface. La classe XFEVENT possède un pointeur de type ISM qui adresse l'objet cible, donc l'objet avec la machine d'états-transitions. Des telles classes disposent d'une méthode processEvent laquelle est utilisé par le XF pour expédier des évènements.
- **XFEVENT** : Cette classe représente un évènement et on peut générer des objets de cette classe à n'importe quel endroit dans le code. Chaque tel objet doit disposer d'une identité unique (`id`) ainsi que d'une cible (`target`) bien identifiée. Un événement est passé au XF à l'aide de la méthode pushEvent. Le XF insère l'événement dans la liste des événements et l'expédiera au moment donnée à la cible spécifiée par `target`. La cible doit implémenter l'interface ISM pour que le XF puisse appeler la méthode processEvent et passer ainsi l'événement à l'objet cible. Après l'exécution de la méthode processEvent, le XF efface l'événement si ceci n'est pas interdit par l'attribut `doNotDelete`.
- **XFTIMER** : Des événements sont normalement expédiés par le XF à leur cible dès que possible. Mais ils y existent des situations, dans lequelles on souhaite pouvoir retarder un événement pour un certain temps. Et ils y existent même des situations, dans lesquelles on aimerait recevoir un événement de façon périodique pour un nombre limité ou illimité des fois. Ceci peut être paramétré par les propriétés `delay` et `repeatCount` dans l'événement. Si la valeur de `delay` est plus grande que zéro, le XF construira un XFTimer pour cet événement. Ce dernier créera lui-même une minuterie (QTTimer) laquelle échoué après le temps donné par `delay`. A ce moment, le XF expédiera l'événement à sa cible. Pour le cas où `repeatCount` est plus grand que zéro, le

XF répètera cette séquence. XFTimer est une classe interne au XF et elle n'est pas utilisé par les machines d'états-transitions que de façon indirecte.

- **XF** : Cette classe est quasiment le moteur du XF. Elle est essentiellement responsable du traitement des évènements. Elle dispose d'une liste d'évènements dans laquelle elle enregistre les évènements jusqu'au moment de leur expédition vers les machines d'états-transitions. Internement, elle utilise pour faire ceci le mécanisme Signal-Slot de la librarie QT. Pour passer un évènement au XF il faut utiliser la méthode pushEvent. Le XF lui-même est un singleton et peut en tout moment être atteint par l'appel de la méthode XF::getInstance.

Les diagrammes de séquence suivants illustrent l'utilisation du XF dans le cas du système Trap.

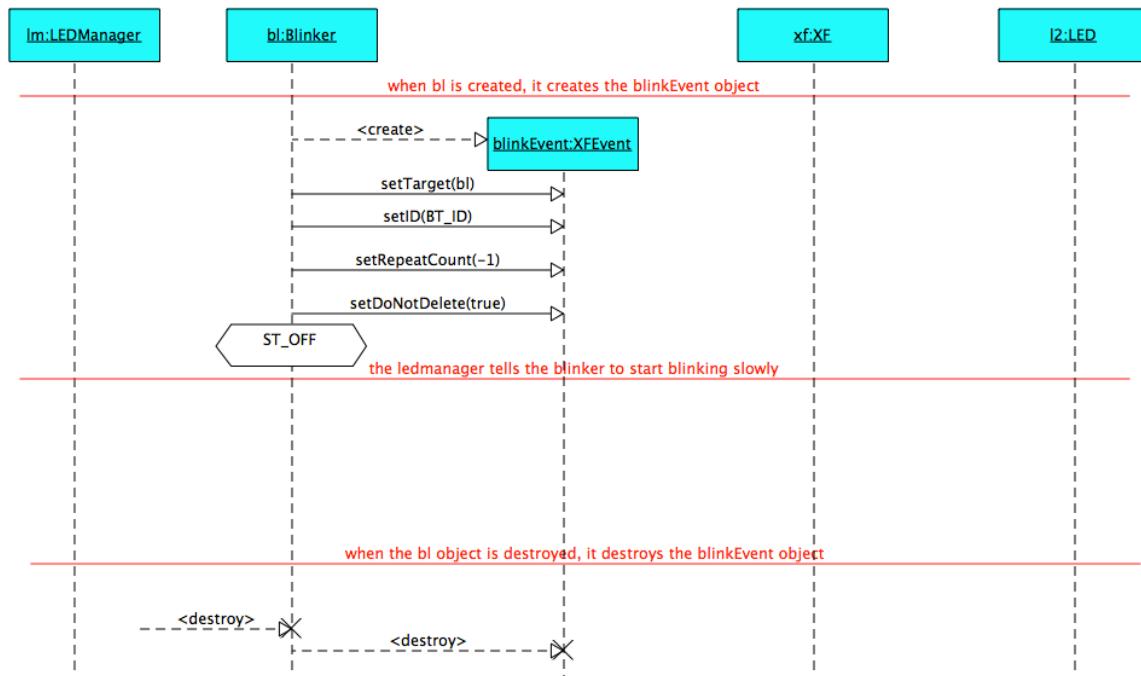


Figure 91 : Séquence d'initialisation avec le XF et un objet de la classe Blinker

On peut très bien voir ici comment l'objet **Blinker** crée un objet **XFEVENT** et comment il le paramètre de façon pour qu'il soit considéré par le **XF** comme un événement périodique et répété de façon illimitée. La période (temps de clignotement) sera réglée plus tard parce que celle-ci dépend de l'application. On peut aussi voir que cet événement ne peut pas être effacé par le **XF** mais seulement par l'objet **Blinker** lui-même.

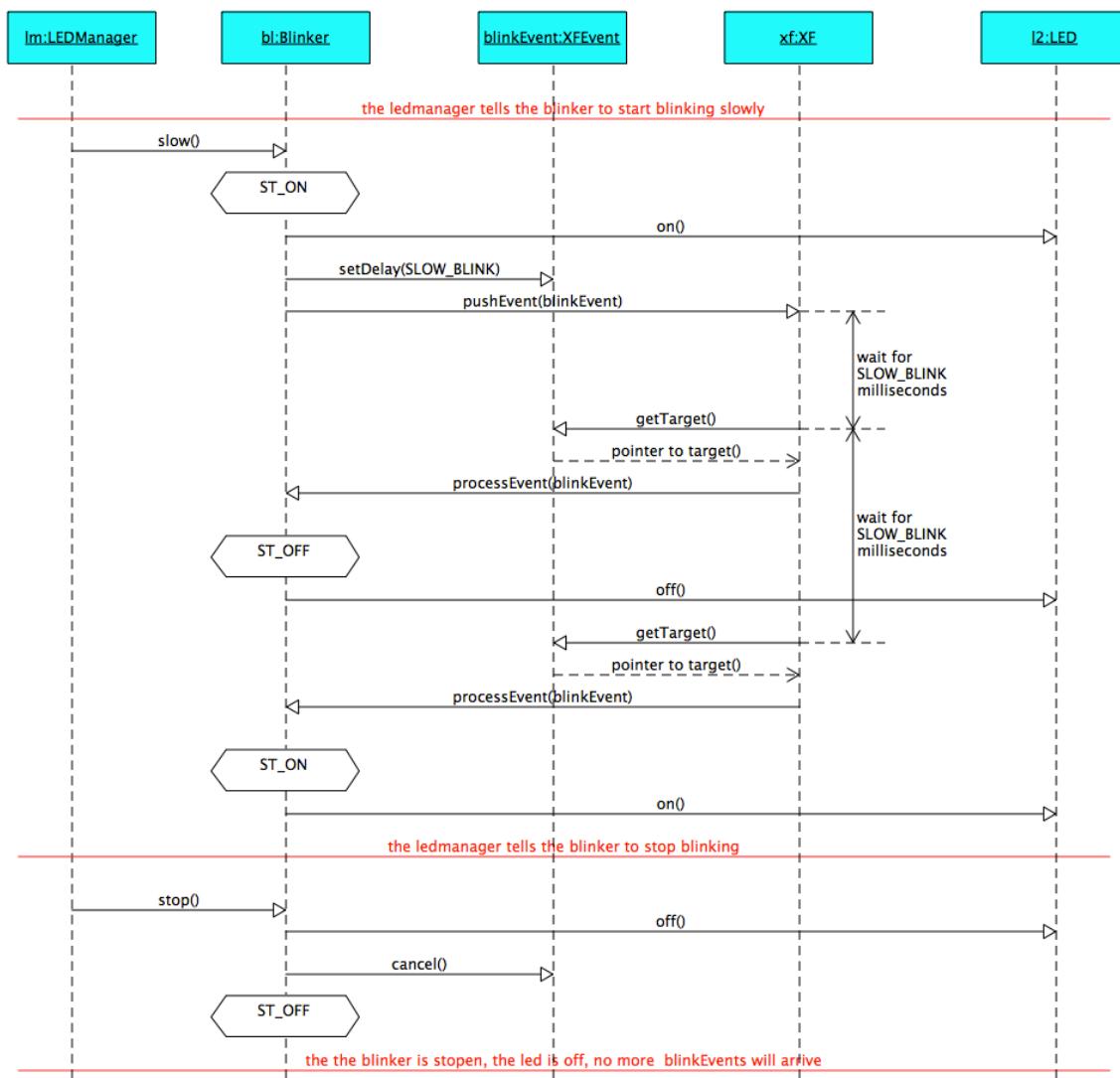


Figure 92 : Séquence de clignotement avec le XF et un objet de la classe Blinker

A l'appel de la méthode `slow` on règle d'abord la période (`delay`) de l'objet `XFEvent`. Ensuite, l'événement est passé au XF. Le XF enregistre l'événement dans sa liste d'événements et crée et démarre une minuterie correspondante. Quand cette dernière échoue, l'événement est rendu à l'objet cible qui dans notre cas est l'objet `Blinker`. L'événement a été créé de façon que le XF l'expédiera périodiquement à l'objet `Blinker` jusqu'au moment de l'appel de la méthode `cancel`. Celle-ci arrête le mécanisme de la minuterie et efface l'objet `XFTimer` y associé. Pour relancer cet événement périodique, il faudra appeler la méthode `pushEvent` de nouveau.

Implémentation des composants

Cette étape produit les diagrammes des composants ainsi que les diagrammes nécessaires pour la création et le lien de tous les objets du système Trap. Des diagrammes de déploiement ont déjà été créés pendant une étape précédente et peuvent servir comme point de départ pour les diagrammes de composants. La figure ci-dessous montre le diagramme de composants tel qu'il a été conçu pour le système Trap.

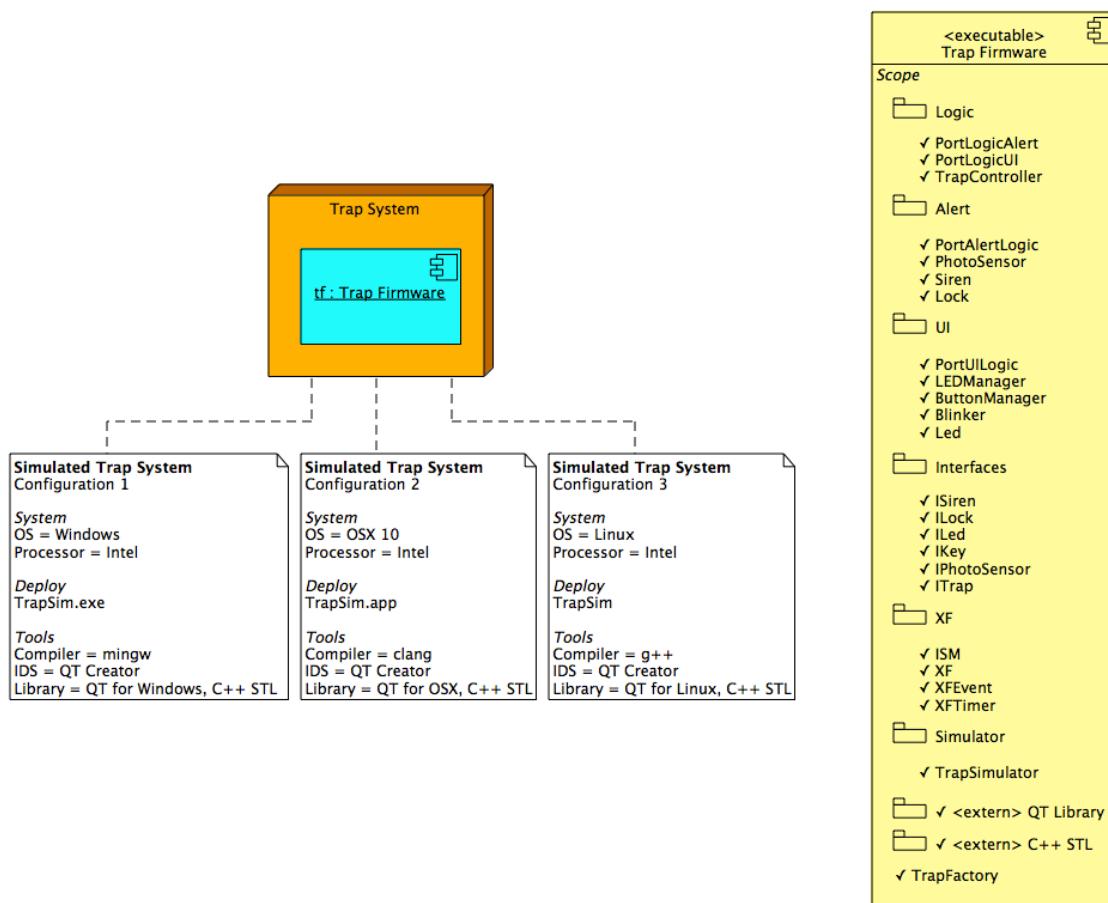


Figure 93 : Diagramme de composants avec 3 configurations pour le système Trap

A la gauche, on peut voir le nœud qui représente le système Trap de façon physique et qui est dans notre cas un PC. Ce noeud offre une instance du composant Trap-Firmware

Une autre étape importante est la définition des configurations. Chacun des composants pourra en principe être exécuté par plusieurs cibles différentes. Une cible se compose d'une architecture de processeur et d'un système d'exploitation. En principe, la cible définit aussi le langage de programmation appliqué mais ce dernier a été choisi pour des raisons pratiques déjà bien avant cette phase. Les éléments suivants doivent être définis pour chacune des différentes configurations :

- Le compilateur, le débogueur et le linker lesquels compilent le code créé, le corrigent et le lient dans une application exécutable.
- Les outils qui sont utilisés pour créer les médias de déploiement
- Le nom du résultat généré

Dans le cas du système Trap trois configurations différentes sont offertes.

En plus, on peut voir dans ce diagramme à la droite la définition du composant lui-même. On voit que le composant Trap-Firmware est un composant dont sera générée un exécutable. Le scope de composant énumère toutes les classes qui sont nécessaires pour la création de cet exécutable. Dans le cas d'outil de modélisation avancés, le code

de ces classes est généré depuis les diagrammes de façon automatique. Dans notre cas, nous devrons écrire le code nous-mêmes de façon manuelle.

Et encore, il s'agit de produire pour chaque composant une classe qui crée et lie les objets de ce dernier. Une telle classe on appelle une classe **Factory**.

Celles-ci sont des classes conteneurs qui produisent et initialisent les objets d'un composant. Par défaut, il faudrait créer une classe **Factory** pour chaque composant. Pour des raisons spécifiques aux systèmes embarqués, on peut aussi prévoir des solutions plus simples.

Les classes **Factory** sont responsables de l'initialisation correcte de tous les objets ainsi que de la création et l'installation des liens entre eux. Ceci doit être effectué selon les associations entre les différentes classes. Chaque une des classes pour cette raison obtient une méthode `initRelations` avec un nombre de paramètres utilisés pour initialiser les liens. Ensemble avec la classe **Factory**, ce système forme un espèce de pattern. Le diagramme de classes ci-dessous montre les éléments du "Factory-Pattern" pour les système Trap. Remarquez bien que les associations sont implémentés comme attributs des classes. Ces attributs sont ensuite initialisés par les paramètres des méthodes `initRelations`.

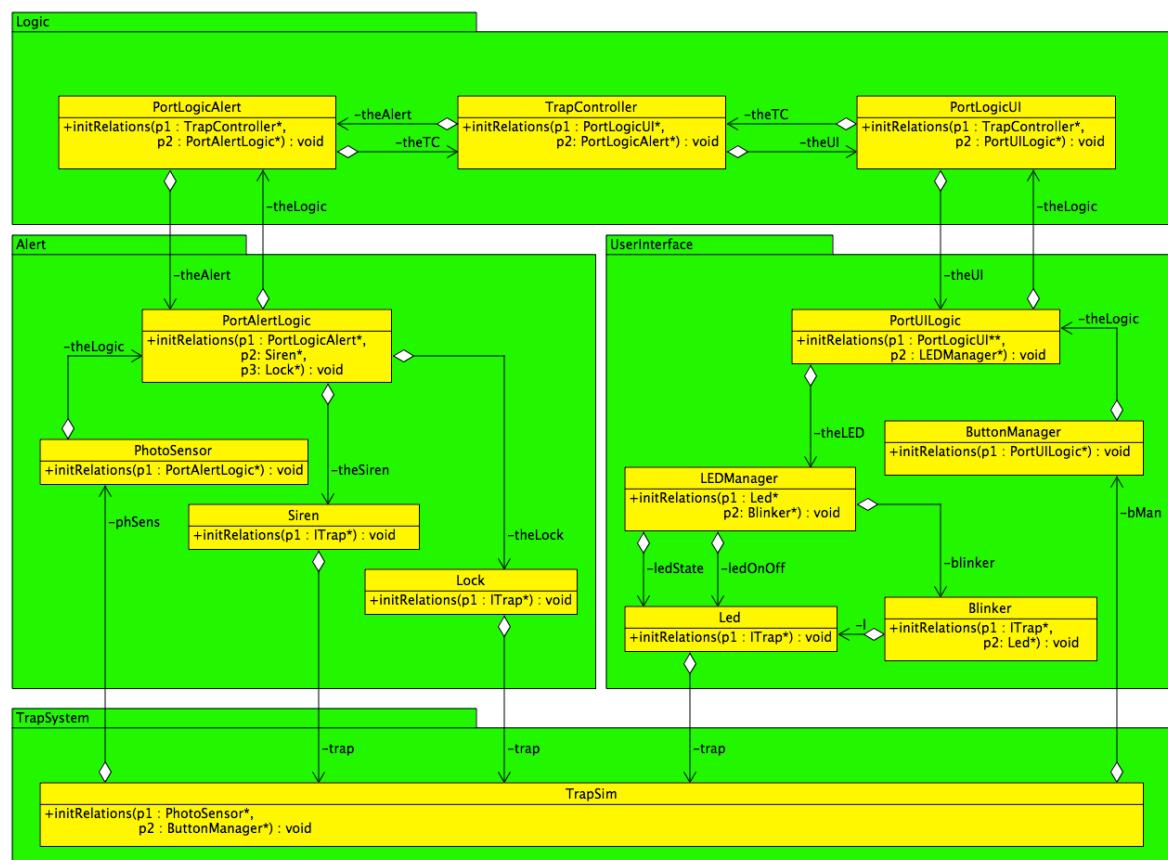


Figure 94 : Eléments Factory du système Trap

Pour la représentation de la classe **Factory** on utilise une classe UML composite. Avec une classe composite, les composants de la classe composite sont représentés sous forme d'objets. Le diagramme de classe du système Trap est illustré par la figure suivante :

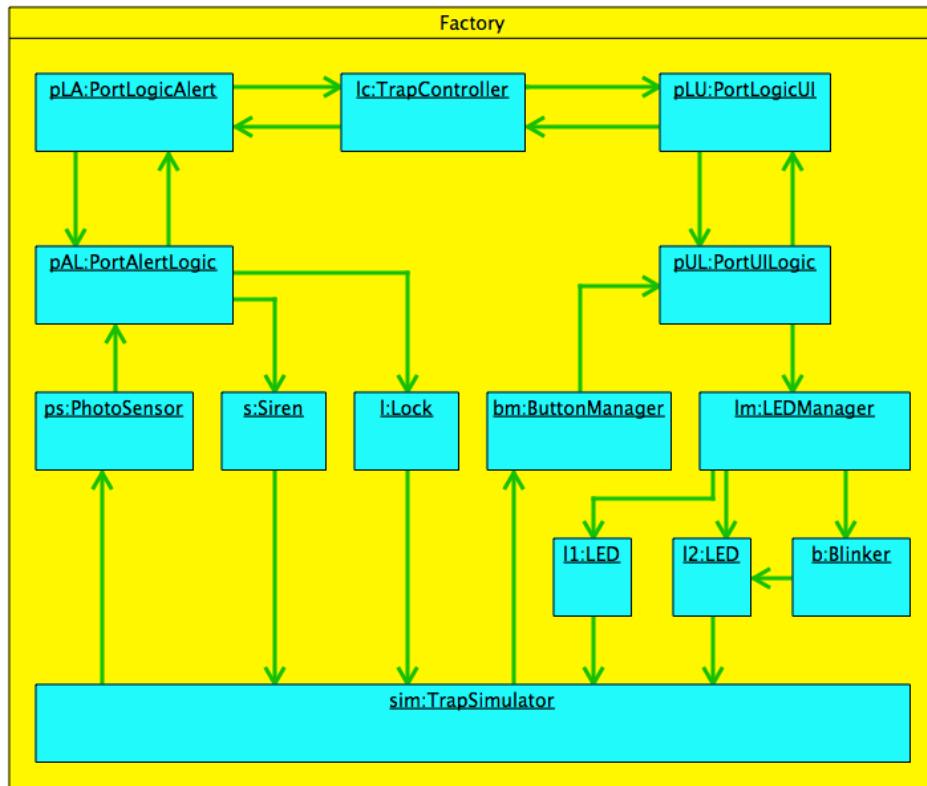


Figure 95 : la classe **Factory** du système Trap

Avec ce diagramme, tous les pas de l'étape d'implémentation sont conclus et le code des classes peut être créé selon le modèle et selon les patterns d'implémentation. Les diagrammes et les patterns suivants sont relevant :

- Le diagramme de classes résultant de l'étape de conception. Ce diagramme résulte de l'application du Layered Structure Pattern et de l'application du Port Pattern au diagramme de classes sortant de l'étape d'analyse. (figure 87)
- Les diagrammes des machines d'états-transitions ainsi que le diagramme de classes relatif à l'application du State Machine Pattern (figure 89, figure 90)
- Le diagramme de composants (figure 93)
- Les diagrammes de classes relatifs à l'application du Factory Pattern (figure 94, figure 95)

Dans la suite, des exemples de code C++ des différents éléments du système Trap sont montrés.

```
/*
 * Interface ILrd
 * This interface provides a method setLedState
 * for classes that have to deal with leds. The
 * parameters passed is the id of the led and
 * whether it has to be set on or off.
 */
#ifndef ILED_H
#define ILED_H

class ILED
{
public:
    virtual void setLedState(int p1, int p2) = 0;
};

#endif // ILED_H
```

Figure 96 : L'interface ILED

```
#ifndef FACTORY_H
#define FACTORY_H

class TrapSim;
class Lock;
class Siren;
class PhotoSensor;
class ButtonManager;
class LEDManager;
class Led;
class Blinker;
class PortAlertLogic;
class PortLogicAlert;
class PortUILogic;
class PortLogicUI;
class TrapController;

class Factory
{
public:
    Factory();
    virtual ~Factory();
    //creation method
    void create();
    //build method
    void build();
    //destroy objects
    void destroy();
private:
    //all objects of the system
    TrapSim* ts;
    ButtonManager* bm;
    LEDManager* lm;
    Led* l1;
    Led* l2;
    Blinker* b1;
    PortUILogic* pUL;
    PortLogicUI* pLU;
    TrapController* tc;
    PortLogicAlert* pLA;
    PortAlertLogic* pAL;
    Siren* si;
    Lock* lk;
    PhotoSensor* ps;
};
#endif // FACTORY_H
```

Figure 97 : la déclaration de la classe Factory

```

void Factory::create()
{
    //create all objects
    ts = new TrapSim();
    bm = new ButtonManager();
    lm = new LEDManager();
    l1 = new Led(1);
    l2 = new Led(2);
    b1 = new Blinker();
    pUL= new PortUILogic();
    pLU= new PortLogicUI();
    tc = new TrapController();
    pLA= new PortLogicAlert();
    pAL= new PortAlertLogic();
    si = new Siren();
    lk = new Lock();
    ps = new PhotoSensor();
}

void Factory::build()
{
    //link all the objects according to the system model
    pUL->initRelations(pLU, lm);
    pLU->initRelations(pUL,tc);
    bm->initRelations(pUL);
    lm->initRelations(l1,l2,b1);
    l1->initRelations(ts);
    l2->initRelations(ts);
    b1->initRelations(l2);
    tc->initRelations(pLU,pLA);
    pAL->initRelations(pLA,si,lk);
    pLA->initRelations(pAL,tc);
    ps->initRelations(pAL);
    si->initRelations(ts);
    lk->initRelations(ts);
    ts->initRelations(bm, ps);
}

```

Figure 98 : Les méthodes `create` et `build` de la classe `Factory`

```

/*
 * Class PortLogicAlert
 * This class is the gateway to the alert package.
 * It provides interfaces for the usage of the
 * siren, the lock and the photosensor.
 * It has a connection to its peer port PortAlertLogic
 * as well as a connection to the controller class
 * The controller class and its peer port are connected to it,
 * but the port does not know this.
 */

#ifndef PORTLOGICALERT_H
#define PORTLOGICALERT_H

#include "Interfaces/iphotosensor.h"
#include "Interfaces/ilogic.h"
#include "Interfaces/isiren.h"

class PortAlertLogic;
class TrapController;

class PortLogicAlert : public IPhotoSensor, public ILock, public ISiren
{
public:
    PortLogicAlert();
    virtual ~PortLogicAlert();
    virtual void onActivated();
    virtual void setSiren(int p1);
    virtual void setLock(int p1);
    void initRelations(PortAlertLogic* p1, TrapController* p2);
private:
    TrapController* theTC;
    PortAlertLogic* theAlert;
};

#endif // PORTLOGICALERT_H

```

Figure 99 : La déclaration de la classe `PortLogicAlert`

```

/*
 * Class TrapController
 * This class is the controller of the trap system.
 * It has a big statemachine doing all the logic.
 * It therefor is reactive and implements the ISM interface.
 * It has connections to the ports PortLogicUI and
 * PortLogicAlert. These ports are also connect to it, but
 * the controller does not know this.
 */

#ifndef TRAPCONTROLLER_H
#define TRAPCONTROLLER_H

#include <map>
using namespace std;

class XFEEvent;
#include "XF/ism.h"

class PortLogicAlert;
class PortLogicUI;

class TrapController : ISM
{
public:
    TrapController();
    virtual ~TrapController();
    void initRelations(PortLogicUI* p1, PortLogicAlert* p2);
    void onOnOffKey();
    void onArmKey();
    void onDisarmKey();
    void onResetKey();
    void onTestKey();
    void onActivated();

private:
    const int WARN_TIME;

    enum TC_ON_OFF_STATE
    {
        ST_ON_OFF_NONE, ST_ON=10, ST_OFF
    };
    enum TC_ON_OFF_EVENT
    {
        EV_ON_OFF_NONE, EV_ONOFF=100
    };
    enum TC_TRAP_STATE
    {
        ST_TRAP_NONE, ST_IDLE=20, ST_TRAPPING, ST_WARNING, ST_ALERT
    };
    enum TC_TRAP_EVENT
    {
        EV_TRAP_NONE, EV_ARM=200, EV_DISARM, EV_TIMEOUT, EV_TEST, EV_RESET, EV_ACTIVATE
    };

    TC_ON_OFF_STATE outer;
    TC_TRAP_STATE inner;

    map<TC_ON_OFF_STATE, map<TC_ON_OFF_EVENT, TC_ON_OFF_STATE>> outerSM;
    map<TC_TRAP_STATE, map<TC_TRAP_EVENT, TC_TRAP_STATE>> innerSM;

private:
    PortLogicAlert* theAlert;
    PortLogicUI* theUI;
    XFEEvent* timeout;

    // ISM interface
public:
    bool processEvent(XFEEvent* p1);
    bool processOuterEvent(XFEEvent* p1);
    bool processInnerEvent(XFEEvent* p1);
};

#endif // TRAPCONTROLLER_H

```

Figure 100 : la déclaration de la classe TrapController

```

bool TrapController::processOuterEvent(XFEvent *p1)
{
    //save the actual state
    TC_ON_OFF_STATE oldState = outer;
    //hash the new state from the state table
    TC_ON_OFF_STATE newState = outerSM[oldState][(TC_ON_OFF_EVENT)p1->getID()];
    //if the hash table found a new state
    if (newState != ST_ON_OFF_NONE)
    {
        //save the new state
        outer = newState;
    }
    //we do not know if the event will be processed
    bool retVal = false;

    if (outer != oldState)
    {
        //the event will be processed
        retVal = true;

        //do the eventual onexit-action of the old state
        switch (oldState)
        {
        case ST_ON:
            //we exit the on state
            //the on-off led is off
            //the state led is off
            //the siren is off
            theUI->setLedState(1,0);
            theUI->setLedState(2,0);
            theAlert->setSiren(0);
            break;
        case ST_OFF:
            //reset the inner statemachine
            inner = ST_IDLE;
            break;
        }
        //do the eventual onentry-action of the new state
        switch (outer)
        {
        case ST_ON:
            //the on-off led is on
            theUI->setLedState(1,1);
            break;
        case ST_OFF:
            //no entry action
            break;
        }
    }
    //return the result of the event processing
    return retVal;
}

```

Figure 101 : La machine d'états-transition extérieure de la classe TrapController

Les figures suivantes montrent des copies d'écran du système Trap dans de configurations différentes.



Système Trap sous OSX



Système Trap sous Linux

De cette manière, l'étape d'implémentation est terminée. Il s'agit maintenant de vérifier le comportement correct des composants créés. Aussi ce pas sera effectué selon certaines règles et va ainsi amener systématiquement un résultat acceptable.

Vérification

Cette section décrit les étapes de la phase de vérification. Elle se compose essentiellement de deux parties importantes décrites par la suite :

- Vérification du comportement
- Vérification de la stabilité

Vérification du comportement

La vérification du comportement compare le comportement spécifié pendant la phase d'analyse avec le comportement implémenté et localise les erreurs possibles. Les techniques suivantes sont appliquées :

- Tests passifs par des testeurs (test Black box)
- Tests actifs par la comparaison de diagrammes de séquence planifiés et réalisés. (test White box)

Des erreurs possibles de comportement sont mises en évidence et une nouvelle rotation du cycle de développement est enclenchée. Le test du comportement pour des systèmes embarqués est relativement complexe. Cette thématique sera précisée dans un chapitre ultérieur. L'image suivante montre la comparaison des scénarios et une itération de la spirale de développement résultant du fait que des incompatibilités ont été découvertes.

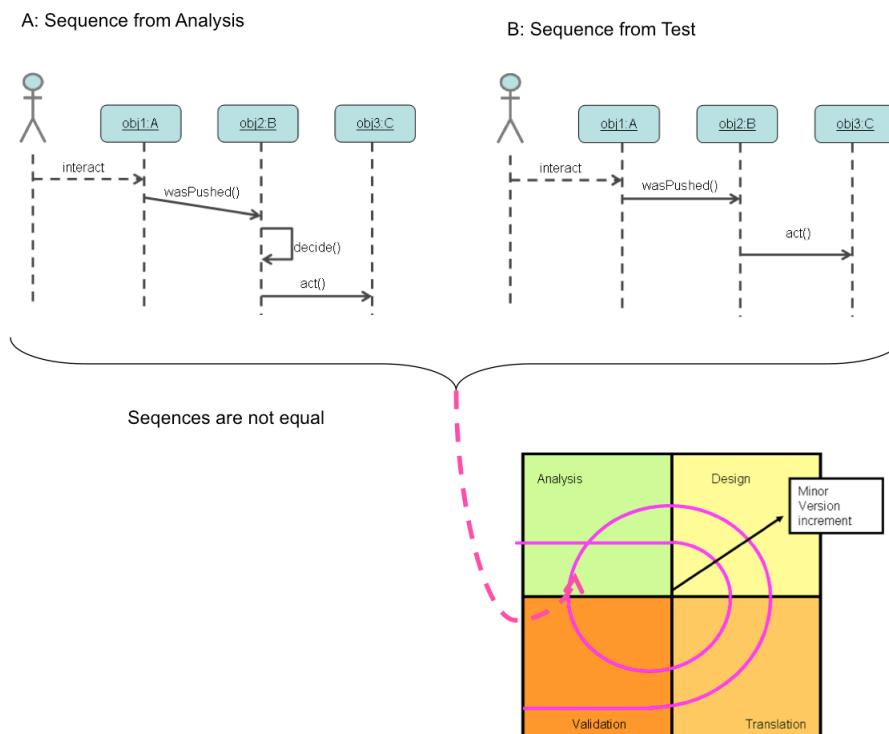


Figure 102 : Itération de la spirale de développement

Vérification de la stabilité

En ce qui concerne la vérification de la stabilité, il s'agit surtout d'une preuve d'un comportement exempt d'erreurs. Des erreurs peuvent être provoquées par beaucoup de sources différentes comme des interfaces entre le matériel et le logiciel, l'utilisation aberrante de ressources etc. Les techniques suivantes peuvent découvrir des erreurs de stabilité :

- Mise en place de métriques de logiciel qui indiquent la complexité du code
- Réduction systématique de la complexité au strict minimum
- Optimisation des générateurs de code
- Application de modèles pendant la phase de conception
- Installation d'un environnement de test proche de la réalité et complètement séparé de l'environnement de développement
- etc.

Il n'est pas possible dans ce cours de décrire en détail ces techniques, d'autant plus que chacune d'elle pourrait être le thème d'un travail propre. Dans un chapitre ultérieur, cependant quelques techniques pour des systèmes embarqués seront mentionnées et décrites plus précisément. Cependant les points suivants doivent aussi être respectés pendant le développement et surtout pendant la phase de vérification :

- Mise en œuvre d'une documentation technique
- Mise en œuvre des manuels d'utilisation et de maintenance
- Mise en œuvre des maquettes des protocoles d'acceptation
- etc.