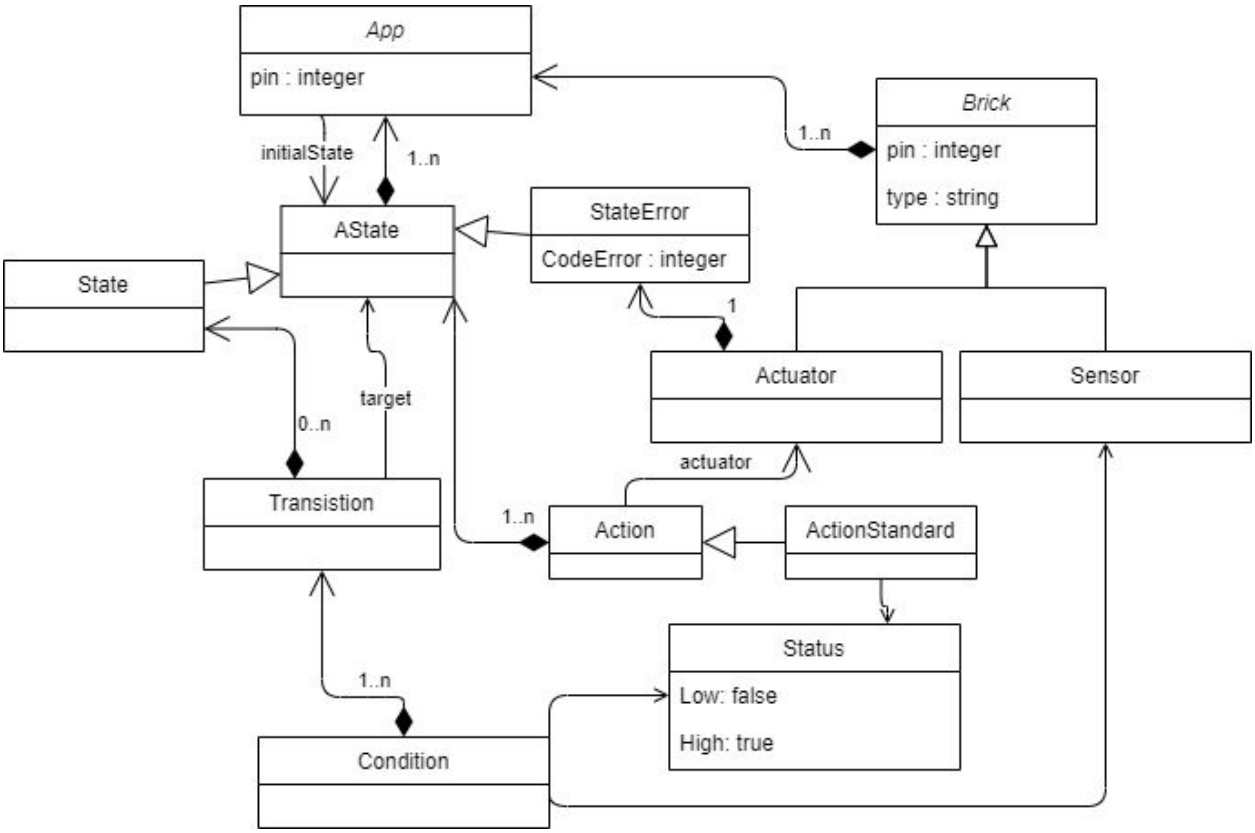
Rapport DSL

I. Plan

Équipe : Corentin Artaud, Hugo Francois et Théos Mariani

II. Description des langages developpés

A. Domain model



B. Extensions

Nous avons choisi d'implémenter les extensions suivantes :

- **Exception Throwing**

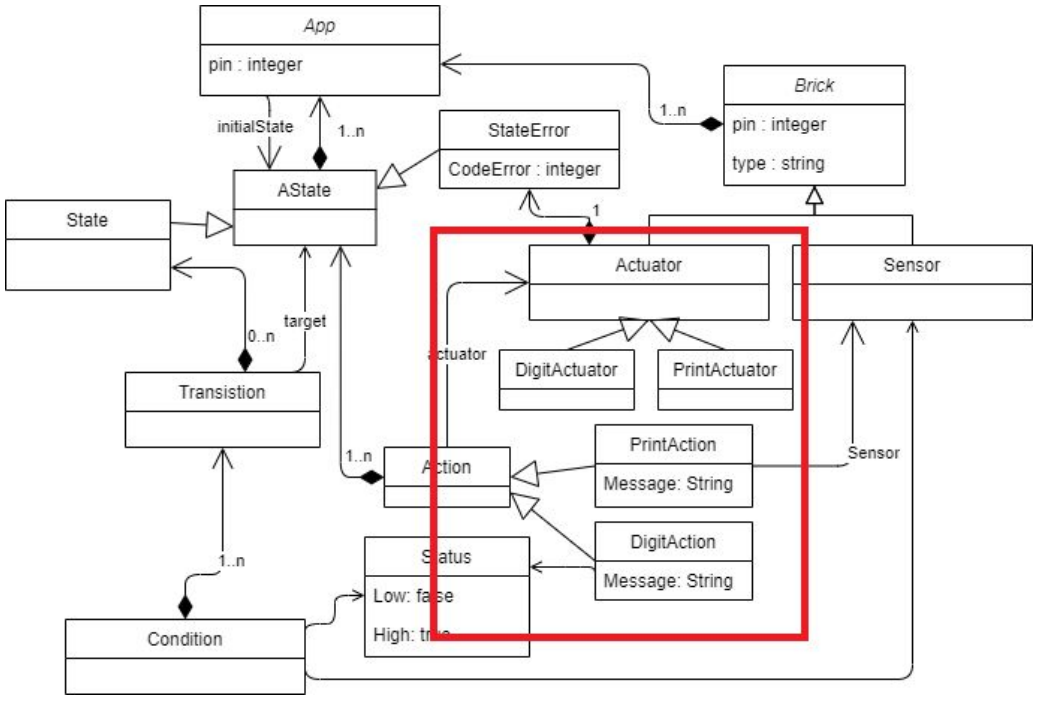
Dans un premier temps pour implémenter l'extension throwing, nous avons choisi de créer un objet *Throwing.* Celui était très similaire à l'objet transition puisqu'il était contenu dans un état et contenait une liste condition. Lorsque les conditions étaient vraies, une fonction prédéfinis était exécuté pour faire clignoter la led (contenue dans l'objet *Throwing*) périodiquement.
Cette architecture nous permettait de réaliser le scénario d'acceptance mais elle comportait les défauts suivants :

- Pas de représentation de l'état d'erreur
- Plus de règles syntaxiques

C'est pourquoi nous avons changé pour notre système actuel. Nous avons créé une abstraction de state afin de pouvoir différencier un état standard et un état d'erreur en deux sous classes. Les différences principales sont qu'un état d'erreur ne peut pas contenir de transition contrairement à un état standard et qu'il contient un *codeError* en attribut pour déterminer la fréquence de clignotement de la led.

- **Supporting the LCD screen**

Nous n'avons pas eu le temps d'implémenter l'affichage sur l'écran LCD mais nous avions déjà réfléchi aux conséquences sur notre model diagramme. Pour cela nous voulions ajouter un type d'action qui en plus de contenir un *Actuator* celui-ci aurait pu contenir un message en attribut ou *Sensor* pour afficher en directe les valeurs de celui-ci (exemple : thermomètre).



C. Règles syntaxique

1. Externe

```
app <AppName> init_state : <StateInitName> {

  bricks :
    actuator <ActuatorName>: <PinInt>
    sensor <SensorName> : <PinInt>
  state :
    <StateName> {
        <ActuatorName> become <Status>
        if <SensorName> is <Status> and <SensorName> is <Status> then
<ActuatorName>
        Error <CodeError> on <ActuatorName> when <SensorName> is <Status> and
<SensorName> is <Status>
    }

     <StateName> {
       <ActuatorName> become <Status>
        if <SensorName> is <Status> and <SensorName> is <Status> then
<ActuatorName>
        Error <CodeError> on <ActuatorName> when <SensorName> is <Status> and
<SensorName> is <Status>

    }
}
```

2. Interne

```
       Sensor <SensorName> pin <PinNumber>
       Actuator <ActuatorName> pin <PinNumber>

       State <StateName> means <ActuatorName> becomes <Status>
       State <StateName> means <ActuatorName> becomes <Status> and
<ActuatorName> becomes <Status>
       Initial <StateNale>

       From <StateName> to <StateName> when <SensorName> becomes <Status>
       error <CodeErrorInt> on <ActuatorName> from <StateName> when
<SensorName> becomes <Status> and <SensorName> becomes <Status>
```

III. Scénarios

Les scénarios ci-dessous permettent avec deux boutons et une lampe de déclencher une exception en pressant les deux boutons ou simplement allumer la lumière en appuyant sur un bouton.

A. Langage interne

```
sensor but1 pin 9
sensor but2 pin 10
actuator led1 pin 11

state iddle means led1 becomes low and led2 becomes low
state led1_on means led1 becomes high

initial iddle

from iddle to led1_on when but1 becomes high
from iddle to led1_on when but2 becomes high

from led1_on to iddle when but1 becomes low
from led1_on to iddle when but2 becomes low

error 3 on led1  from led1_on when but1 becomes high and but2 becomes high
error 3 on led1 from iddle when but1 becomes high and but1 becomes high
```

Il a comme avantage d'avoir une syntaxe simple et concise. Cependant elle pourrait être amélioré en ajoutant l'opérateur ***OR*** dans les conditions. Cela nous permettrait de diminuer par deux le nombre de déclaration de transition.

B. Langage Externe

```
app ExceptionThrowing init_state : iddle {

  bricks :
    actuator error_led : 8
    actuator green_led : 12
    sensor button1 : 9
    sensor button2 : 10
  state :
    iddle {
        error_led become low
        green_led become low
        if button1 is low and button2 is high then led_on
        if button1 is high and button2 is low then led_on
        error 3 on error_led when button1 is high and button2 is high
    }
```

```
        led_on {
            error_led become low
            green_led become high
            if button1 is low and button2 is high then iddle
            if button1 is high and button2 is low then iddle
            error 3 on error_led when button1 is high and button2 is high
        }
    }
```

Ce langage, contrairement au langage interne pour le même scénario, est un peu plus expressive. Cependant dans l'ide MPS grâce à auto-complétion il devient beaucoup simple d'écrire un scénario. Ici aussi nous n'avons pas implémenter l'opérateur **OR** car cela nous aurait coûté trop de temps de bien gérer les différentes priorités dans les opérateurs.

IV. Critique

A. Kernel

Nous avons recréé le kernel afin de mieux comprendre comment il fonctionnait. Ensuite nous avons intégrée de nouveau concept comme celui de condition afin de pouvoir en avoir plusieurs. Cela nous a permis de mieux répondre aux scénarios de base notamment pour déclencher des actuators à partir de deux sensors. De plus cela nous permettrait d'intégrer assez facilement la compatibilité avec des capteurs analogiques.

Par la suite nous avons créé d'autre concept pour les extensions. Comme le concept *StateError* qui permet de représenter un état d'erreur lorsqu'une exception est déclenché.

B. Technologies utilisées

Nous avons décidé d'utiliser l'outil MPS pour définir notre langage externe car celui-ci guide énormément ce qui permet de facilement concevoir des langages externes. De plus les scénarios devant respecter un template très stricte cela simplifie grandement leur écriture.
Cependant il a comme principale inconvénient qu'un langage réaliser avec MPS est très fortement dépendant de l'IDE. Puisqu'il ne pourra pas être ouvert par un autre IDE et que sans l'aide de celui-ci le langage devient bien moins pratique.

Pour le langage interne nous avons utilisé groovy car il est simple de l'interfacer avec du java et nous permet d'obtenir avec un peu de connaissance une syntaxe simple et fluide (pas nécessaire de déclarer une classe ou une fonction main en tant qu'utilisateur du langage)
Par contre il est assez compliqué d'obtenir un fichier de règles permettant d'être inclus comme extension dans un IDE ou éditeur de texte afin d'obtenir l'autocomplétion et un linter. De plus

l’interprétation des fichiers respectant le langage nécessite un compilateur basé sur la JVM. Ce qui impose à nos utilisateur d’installer java en plus du compilateur.

V. Responsabilité des membres du groupes

Corentin :

- Kernel
- Langage interne

Hugo :

- Langage externe (MPS)
- Testes sur l’arduino

Théos

- Scénario sur les deux langages
- Syntaxe langage externe
- Diagramme / rapport