Domain Specific Language

Rapport

Choix Technologique

Interne

Groovy est un langage inspiré de plusieurs langages comme le Python et le Rubis dans la façon d'écrire le code. Nous avons fait le choix de prendre ce langage pour 2 raisons. Étant un langage basé sur Java, le stack nous était familière. De plus le language support des écritures très souple et légère pour faire des choses assez puissantes, comme en python. avec un typage dynamique ou non , créer un DSL basé sur Groovy nous a paru assez intéressants.Enfin le langage se veut assez flexible et malléable au niveau de la syntaxe pour écrire des codes métiers très rapides. Il est adapté pour faire du Domain-Specific Languages.

Externe

JetBrains MPS (Meta Programming System) est un langage développé par JetBrains. MPS est un outil de conception de langage spécifique au domaine (DSL).

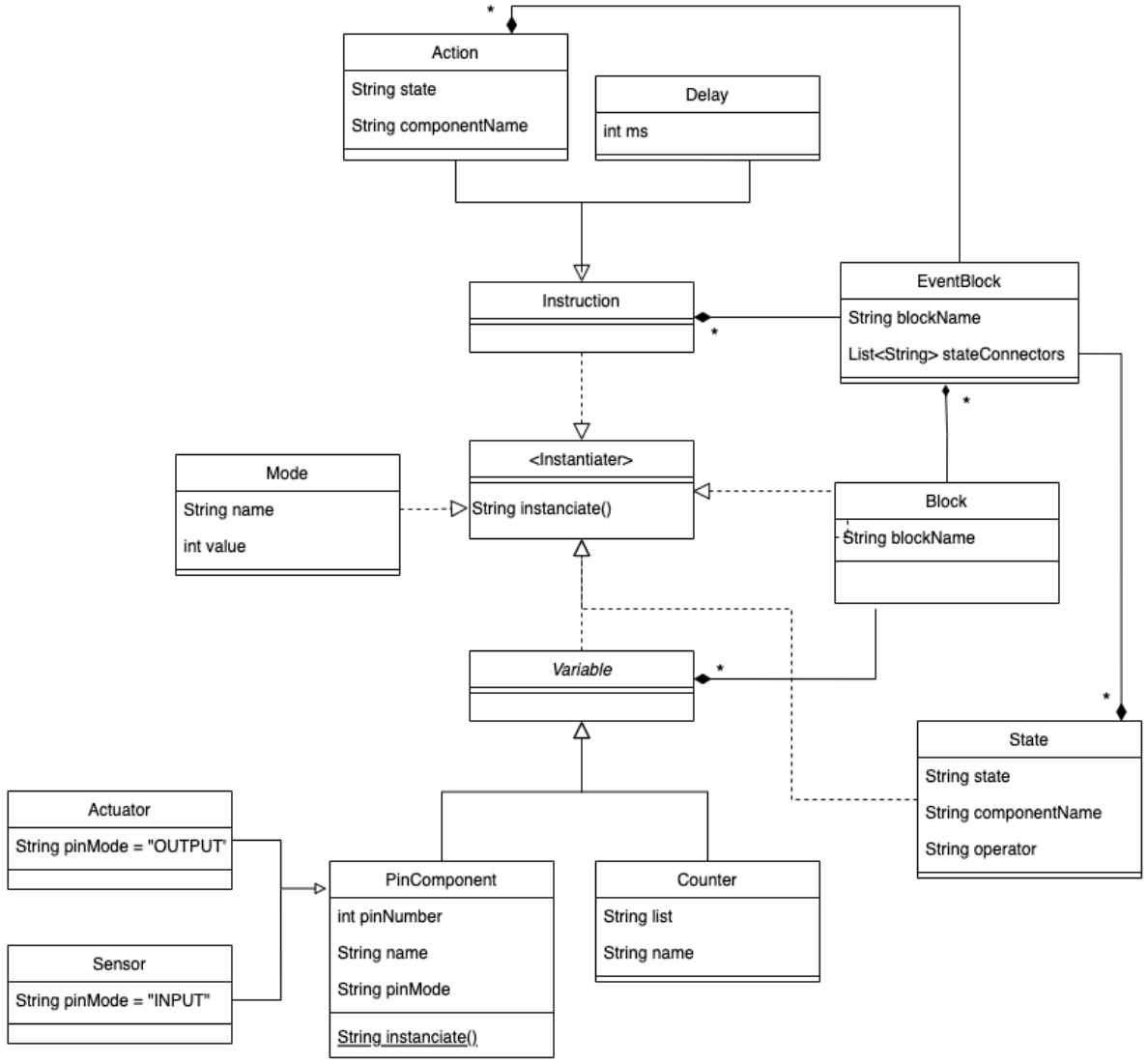
MPS est un framework qui propose déjà une abstraction compilateur/interpréteur du langage DSL que nous créons, il fournit une abstraction pour la création du code cible et du format du DSL en utilisant des "éditeurs" et "TextGen" déjà présents dans l'IDE , ce qui facilite la création du DSL, mais limite la portabilité puisque l'IDE est nécessaire à l'utilisation du DSL

Demonchaux Marine Fernandes William Gros Paul Kherraf Taha

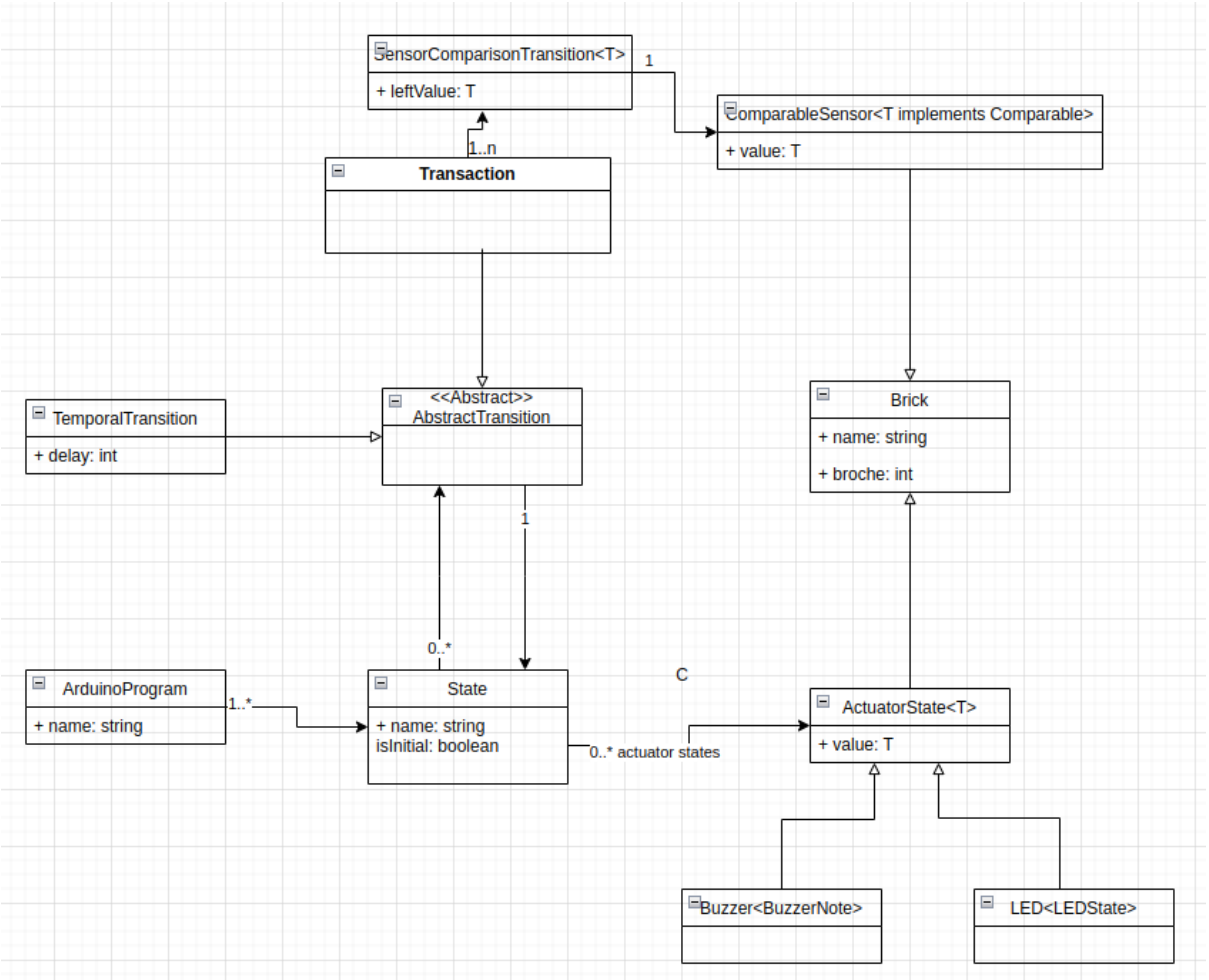
Modèles domaines

Interne

En raison de notre organisation, nos modèles de domaines diffèrent entre le DSL interne et le DSL externe. Voici, sous forme de diagramme de classe, notre modèle domaines pour le DSL interne :



Externe


SensorComparisonTransition<T>
+ leftValue: T
1
ComparableSensor<T implements Comparable>
+ value: T
1..n
Transaction
<<Abstract>>
AbstractTransition
TemporalTransition
+ delay: int
Brick
+ name: string
+ broche: int
1
0..*
ArduinoProgram
+ name: string
1..*
State
+ name: string
isInitial: boolean
C
0..* actuator states
ActuatorState<T>
+ value: T
Buzzer<BuzzerNote>
LED<LEDState>

BNF

Interne

```
#setup
<sensor> ::= sensor <sensor ref> port <port>
<actuator> ::= actuator <actuator ref> port <port>
<mode> ::= mode <mode ref> eq <number>
<actuator ref> ::= <name>
<sensor ref> ::= <name>
<mode ref> ::= <name>
<element ref> ::= <actuator ref> | <sensor ref> | <mode ref>
<elements ref> ::= <element ref>,<element ref>...
<array> ::= array <array ref> affects <element ref> | <elements ref>
<array ref> ::= <name>

<filename> ::= <name>
<name> ::= <char> <name> | NIL
<char> ::= A | B | C | ….
<port> ::= 1 | 2 | 3 | …..
<number> ::=  1 | 2 | 3 | …..

#in loop
<actuator condition> ::= <actuator ref> eq <actuator state>
<actuator state> ::= push | release

<sensor condition> ::= <sensor ref> eq <sensor state>
<sensor state> ::= high | low

<state> ::= <sensor state> | <actuator state> | <number>

<mode condition> ::= <mode ref> eq <number>
<array condition> ::= <array ref> count(<state>) eq <number>
<condition> ::= <actuator condition> | <sensor condition> | <mode condition> | <array condition>
<conditions> ::= <condition> and <condition> ...
<action> ::= set <sensor ref>| <mode ref> to <sensor state> | <number>
<delayed action> ::= on after <number> <action> | <actions>
<actions> ::= <action> and <action> …

<behavior> ::= on <condition> | <conditions> <action> | <actions> | <delayed actions>

#export file
build ::= export <filename>
```

Externe

```
<App> ::= “APP” <NAME> <SENSORS> <ACTUATORS> <START_STATE> <STATES>

<SENSORS> ::= <SENSOR><SENSORS> | NIL
<SENSOR> ::= <NAME> <SENSOR_TYPE> “ON PIN” <PIN>
<SENSOR_TYPE> ::= DIGITAL | ANALOG

<ACTUATORS> ::= <ACTUATOR> <ACTUATORS> | NIL
<ACTUATOR> ::= <NAME> “ON PIN” <PIN>

<NAME> ::= <CHAR> <NAME> | NIL
<CHAR> ::= A | B | C | ….
<PIN> ::= 1 | 2 | 3 | …..

<START_STATE> ::= <STATE>
<STATES> ::= <STATE><STATES> | NIL
<STATE> ::= “STATE” <NAME>
<ACTUATORS_STATE>
<TRANSITIONS>

<ACTUATORS_STATE> ::= <ACTUATOR_STATE> <ACTUATOR_STATE> | NIL
<ACTUATOR_STATE> ::= <HIGH_LOW_STATE> |  <TONE_STATE>

<HIGH_LOW_STATE> ::= <ACTUATOR_NAME> “=” <DIGITAL_STATE>
<ACTUATOR_NAME> ::= <NAME>
<DIGITAL_STATE> ::= “HIGH” | “LOW”

<TONE_STATE> ::= <ACTUATOR_NAME> “NOTE” <NOTE>
<NOTE> ::= 1 | 2 | ….....

<TRANSITIONS> ::= <TRANSITION> <TRANSITIONS> | NIL
<TRANSITION> ::= <DELAY_TRANSITION> | <CMP_TRANSITION>

<CMP_TRANSITION> ::= <CONDITION>  <NEXT_STATE>
<CONDITION> ::= “IF” <SENSOR_NAME> “==” <VALUE>
<NEXT_STATE> ::= “GO TO” <STATE_NAME>
<STATE_NAME> ::= <NAME>
<SENSOR_NAME> ::= <NAME>
<VALUE> ::= 0 |1 | …

<DELAY_TRANSITION> ::= <WAIT> “AND” <NEXT_STATE>
<WAIT> ::= “WAIT FOR” <DELAY>
<DELAY> ::= <VALUE>
```

Présentation de l'extension

Interne

Nous avons décidé, afin de nous préparer au prochain projet, de développer le code du DSL à partir de rien pour implémenter la syntaxe, ainsi qu'implémenter le kernel qui fait office de traducteur entre la syntaxe Groovy et la syntaxe Arduino.

Ne sachant pas exactement comment commencer, nous avons étudié le github fourni écrit en Groovy pour comprendre les possibilités que Groovy offraient.

Avant de commencer le code, nous avons étudié les scénarios que nous voulions implémenter :

- Les 4 scénarios de base.
- Les transitions temporelles.

Nous avons essayé d'écrire le script Groovy que nous voulions avoir pour réaliser ses 5 scénarios.

Notre but était d'avoir un langage très simple, et qui puisse ressembler le plus possible à une phrase. Ce qui rend la learning curve de notre DSL plus facile.

Nous avons développé le code au besoins des scénarios. Donc le DSL ainsi que le kernel étaient fait en même temps.

La partie la plus dure a été de créer le Kernel. On a pu remarquer que certains patterns sont plus adaptés pour transformer le code d'un langage en un autre, comme le pattern composite.

Externe

Pour le DSL externe on a utilisé MPS qui est un outil de conception de langages spécifiques à un domaine (DSL).

Le DSL externe que nous avons implémenté permet de décrire une machine d'état où chaque état est défini par l'état de l'actionneur et les transactions qui, sous une condition sur le capteur, transitent vers un autre état.

l’objectif de ce DSL extern est d’abstraire touts les notions de programmation et de permettre au utilisateur de décrire son code sous forme d’une logique de machine à état, loin de la complexité d’un langage de programmation ce qui est l’une des objectifs d’un DSL

l’utilisateur peut décrire son code arduino sous forme de machine à état sous en respectant la syntaxe présentée par le BNF ci dessus

Scénario Réalisable

Interne

Nous sommes en mesures de faire les 4 scénarios de bases :

```
sensor "btn" port 9
actuator "led" port 10
actuator "buzzer" port 11

on btn eq push set led to high and buzzer to high
on btn eq release set led to low and buzzer to low

export "scenario_1"
```

```
3  sensor "btn1" port 9
4  sensor "btn2" port 8
5  array "btnArray" affects "btn1","btn2"
6  actuator "led" port 10
7  actuator "buzz" port 11
8
9
10
11 on btnArray count(high) eq 2 set buzz to high
12 on btnArray count(high) dif 2 set buzz to low
13
14 export "scenario_2"
```

```
3  sensor "btn1" port 9
4  actuator "led" port 10
5  mode "mode1" eq 0
6
7  on btn1 eq push and mode1 eq 0 and led eq low set led to high and mode1 to 1
8  on btn1 eq push and mode1 eq 0 and led eq high set led to low and mode1 to 1
9  on btn1 eq release set mode1 to 0
10
11 export "scenario_3"
```

```
3  sensor "btn1" port 9
4  actuator "led" port 10
5  actuator "buzz" port 11
6  mode "state1" eq 0
7  mode "state2" eq 1
8
9  on btn1 eq push and state1 eq 0 and state2 eq 1 set buzz to high and state1 to 1
10 on btn1 eq push and state1 eq 1 and state2 eq 1 set led to high and buzz to low and state1 to 2
11 on btn1 eq push and state1 eq 2 and state2 eq 1 set led to low and state1 to 0
12 on btn1 eq release set state2 to 1
13
14 export "scenario_4"
```

Nous supportons aussi les transitions temporelles :

```
3   sensor "btn" port 9
4   actuator "led" port 10
5   actuator "buzzer" port 11
6
7
8   on btn eq push set led to high and on after 800 set led to low
9
10  export "temporal"
```

Externe

pour les scénarios DSL externes nous les avons regroupés dans le dossier « example »

Prise de recul

Interne

Avec du recul, nous pensons que notre implémentation à des qualités et des défauts lié au développement du code :

- La syntaxe à une learning curve très facile, cela ressemble beaucoup à la langue anglaise.
- Le code produit en Arduino est lisible et correctement indenté
- Le développeur peut ajouter plusieurs condition ou plusieurs actions. Nous ne nous sommes pas contentés de reproduire les scénarios.

Concernant les défauts,

- Nous n'avons pas pu synchroniser l'externe de l'interne dans l'écriture ni dans le class model.
- Le kernel est mal conçu , au fur et à mesure du développement, nous avons fait des modifications ou des refactoring sur les classes déjà existantes, ce problème vient du fait que notre diagramme de classe n'était pas bien conçu dès le départ, de plus, nous avons essayé de développer sans copier le code kernel que l'on pouvait récupérer sur un dépôt git.
- Le code arduino pourrait être optimisé
- Nous ne faisons aucune vérification sur la validité du code.

Externe

Points forts:

- abstraction du langage de programmation
- syntaxe simple et compréhensible pour les utilisateurs qui ne savent pas code
- la description du code sous forme de machine à état permet l'implémentation plusieur scénario

points faibles:

- Le code arduino généré pour la machine à état n’est pas optimisé, et peut causer un “overflow” de la stack puisque l'implémentation se base sur des appels récursifs des fonctions
- Dépendance à l'IDE intellij MPS pour utiliser le DSL
- Besoin d'une logique de machine d'état pour pouvoir implémenter des exemples plus complexes
- Pas de validation de code pour le DSL

Organisation de travail

Nous avons divisé notre équipe en deux équipes de 2, afin de développer le DSL interne et le DSL externe, cela nous a permis de développer plus vite. En revanche, chacun de nous n'a pas pu réellement faire les deux côtés. Pour pallier cela, nous avons fait des réunions pour que chaque partie du groupe explique son travail à l’autre afin de comprendre et appréhender la techno du prochain projet.

Demonchaux Marine Fernandes William Gros Paul Kherraf Taha