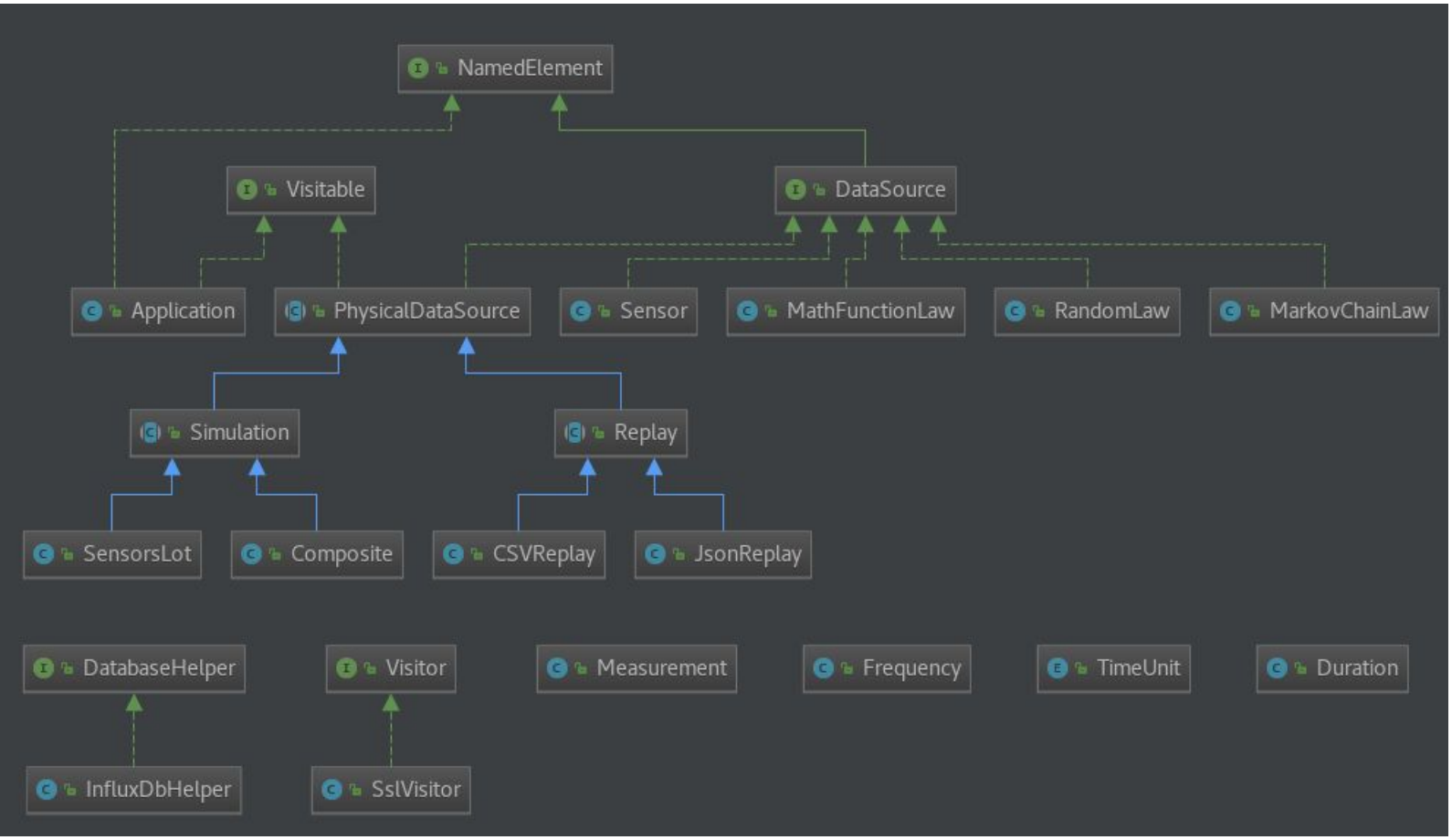
Sensor Simulation Lab

(Jeremy_Lefebvre && Florent_Pastor && Anthony_Fusco)

Model du domaine

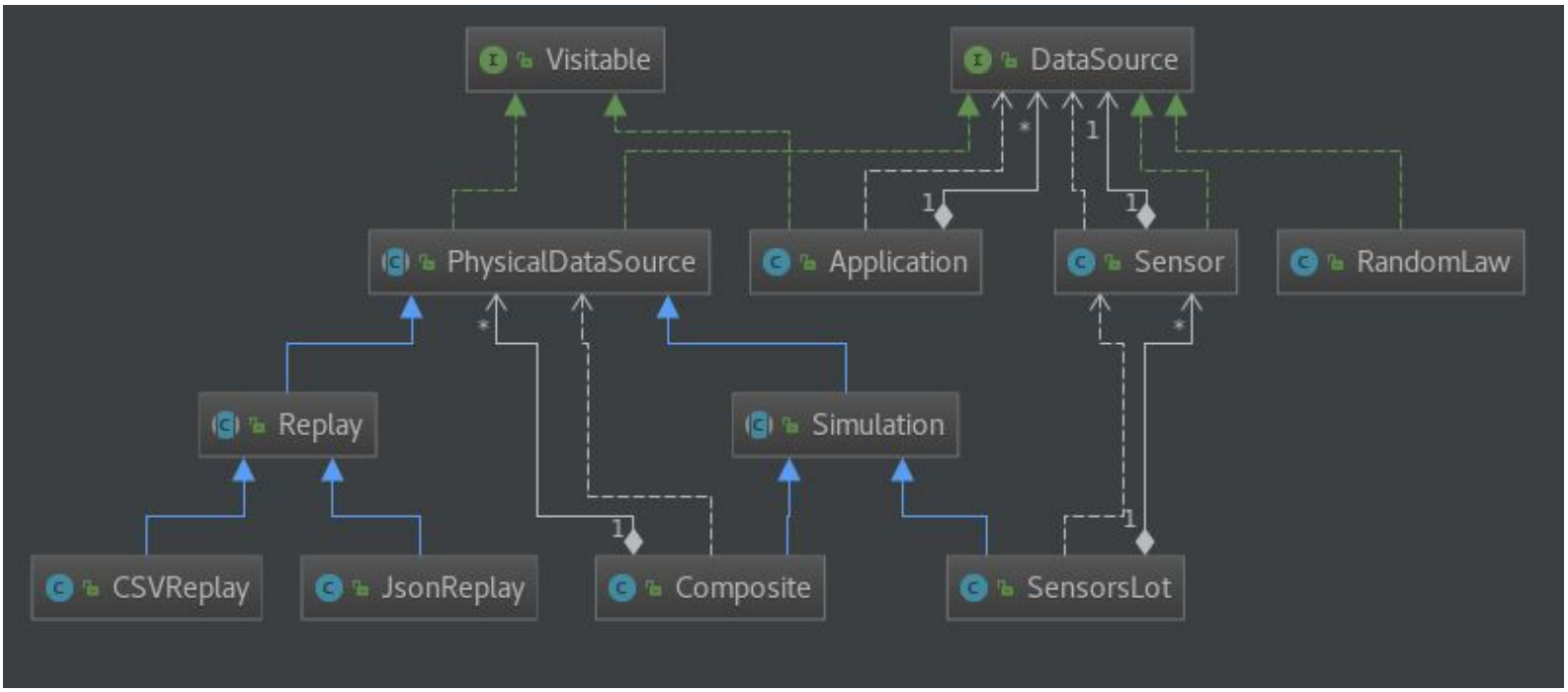
Syntaxe Abstraite :



Le diagramme au dessus représente la partie Kernel de notre DSL et est écrit entièrement en Java, le lien est fait avec la partie DSL seulement par des Builders qui construisent les objets métiers et les stockent dans l'objet Application.

L'abstraction de plus haut niveau est l'interface DataSource, implémenter cette interface signifie que l'on peut retourner une liste de mesures pouvant être envoyé au back end. Les différentes lois métiers implémente cette interface, ainsi que l'interface PhysicalDataSource qui représente les composants physiques métier, c'est à dire les sensors générant ces données.

Diagramme simplifié montrant les dépendances entres les classes :



L'abstraction PhysicalDataSource nous permet de facilement visiter les différents composant d'une simulation et récupérer les valeurs finales tout en nous donnant la possibilité de faire de la validation de type. Un composite est composé de PhysicalDataSource, sur lesquels il va déclencher un enchaînement d'appel à une méthode de l'interface DataSource qui finira par nous donner une valeur de sensor, comme pour un pattern "chain of responsability".
La séparation Replay / Simulation nous permet de gérer différemment l'envoi des données au back end pour ces deux composants lors de l'exécution du pattern visiteur.

Syntaxe Concrete

Une loi se déclare avec un mot clef définissant son type tel que markovChain ou randomLaw, puis en lui passant une closure contenant ses paramètres, par exemple :

```
markovChain {
      matrix([[0.7, 0.3], [0.3, 0.7]])
      stateFrequency 2 / h
}
```

Le même principe est utilisé pour les replays, les simulation et les composites.
Exemple d'une loi polynomiale prenant en paramètre une fonction et un noise, utilisé dans deux sensors à une fréquence de 3 données par heures :

```
def f = { t -> t * 2 + 1 }
SensorTempExt = mathFunction {
      expression f
      noise([-5, 5.9])
}

sensorsTemp = sensorLot {
      sensorsNumber 2
      law SensorTempExt
      frequency 3 / h
}
```

Ces lois/Simulation/Replays peuvent être utilisées anonymement ou être mises dans des variables.
L'exécution du script se déclenche avec le mot clef simulate, cette méthode prend plusieurs paramètres, une date de début et de fin de simulation et les simulation/replays à exécuter, par exemple :

```
simulate {
      start "10/02/2018 08:00:00"
      end "10/02/2018 19:00:00"
      play fac, sensorsTemp
}
```

Les replays de fichier se définissent comme ceci (E.g. Pour les CSV) :

```
csvReplay {
      path "datafiles/data1.csv"
      offset 1.h
      columns([t: 0, s: 1, v: 6])
      noise([0.01, 0.15])
}
```

Le paramètre columns représente la colonne à utiliser pour chaque composante.

Une description en détails de chaque fonctionnalité et syntaxe peut être trouvé dans le README.md du projet.

Extension Composite Sensors

L'extension que nous avons choisi permet de prendre des composants existant dans un script tel que les simulations et les replays et de composer leurs valeurs pour obtenir des métriques spécifiques. On voudrait par exemple pouvoir mesurer le taux de remplissage

d’un parking équipé de sensors tout au long de la journée. Pour cela il est possible de prendre les valeurs retournées par ces capteurs et de leur appliquer des transformations du type filter / map / reduce pour obtenir l’occupation du parking.

Pour cela, nous avons un mot clef composite qui prend en paramètre une liste d'exécutable (sensors, replays, composites), des closures représentant les fonctions filter/map/reduce et une fréquence d’envoi des données

Exemple d’un composite utilisant plusieurs sensors précédemment définie et une simulation anonyme créée à la volée :

```
composite {
      withSensors([
            valroseTop,
            valroseMiddle,
            sensorLot {
                  law markovChain {
                      matrix([[0.7, 0.3], [0.2, 0.8])
                      stateFrequency 1 / 10.min
                  }
            }
      ])
      reduce({ res, sensor -> res + sensor })
      frequency 2 / h
}
```

L’objet composite possède la liste d’objet exécutable et leur passe le paramètre de temps pour récupérer une valeur. Nous appliquons ensuite les fonctions filter/map/reduce à ces valeurs pour n’avoir au final qu’une seule valeur par tick.
Nos lois ne pouvant retourner que des valeurs numériques, nous n’avons pas de problème à les composers.
L’utilisation des closures de Groovy nous a permis d’implémenter facilement la fonctionnalité de filter/map/reduce. Notre architecture est aussi bien adapté à cette extension, en effet l’abstraction PhysicalDataSource (hérité par Replays, Simulation et Composite) nous permet d’avoir un check statique sur les objets que nous passons au Composite tout en gardant l’interface DataSource commune avec les lois et donc générer des valeurs de façon générique pour tous les sensors disponibles.
Au niveau métier, les mot clef filter/map/reduce peuvent être discutable, malgré tout ils sont nécessaire pour la lisibilité du langage qui est au final quand même destiné à des développeurs.

Analyse des choix

Syntaxe :

Lors des débuts d'implémentation de notre DSL, nous voulions mettre un point d'honneur à l'expressivité du langage, c'est pourquoi nous avions utilisé le chainage de méthode à la Groovy pour réaliser une fluent API afin que nos fonctionnalités soit le plus expressive possible pour l'utilisateur expert.

Au fur et à mesure de l'avancée de l'implémentation le langage perdait en expressivité ce qu'il gagnait en fonctionnalités et en longueur de phrase. De plus, des soucis de méthodes optionnelles sont venus s'ajouter avec l'apparition des possibilités de décalage sur les dates et de bruit sur les valeurs. Nous avons changer notre fusil d'épaule par la mise en place de Builders répondant à notre problème de méthode optionnelle (nous permettant au passage d'amincir le code présent dans le moteur et de le modulariser dans des classes dédiées). Nous avons perdu notre fluent api qui devenait trop massive au profit de déclaration en Nested Closures.

Cette notation nous as permis d'harmoniser toutes nos déclarations et n'as pas entraîner un grand refactor car ce pattern fonctionne toujours avec des Builders, ce qui nous as permis de garder les bénéfices des Builders tel que l'optionalité des paramètres. De plus, la mise en place de Nested Closures est grandement simplifiée en Groovy et est présentée dans la documentation officielle comme un pattern de DSL. Ce pattern nous as ensuite ouvert la voie vers un autre refactor, nous avons pu utilisé le concept de Reception Dynamique (encore une fois facilité par Groovy) pour implémenter une factory de builder au niveau de notre BaseScript et ainsi avoir un point d'extension bien définie pour notre DSL. Toute déclaration à l'intérieur d'un script SSL prend la forme d'une fonction avec une Nested Closure qui sera dynamiquement relier à un Builder qui construira un objet de notre model. Grâce à ces patterns, nous avons pu unifier notre syntaxe et définir des points d'extension clair pour une meilleur évolutivité des fonctionnalités.

Pour déclarer et utiliser des fonctions mathématiques nous avons bien sur profiter d'être dans un GPL supportant les closures pour ne pas avoir à ré-inventer la roue. Cependant, cette puissance à un coût, celui de la sécurité. En effet, il était dès lors possible pour l'utilisateur d'effectuer des boucles et récursions infinis et de créer ses propres objets groovy. Pour remédier à cela nous avons dû créer des listes de tokens, types de données et mots-clés autorisés dans les scripts du DSL. De plus, nous avons dû fabriquer un système pour interdire la récursion dans une closures qui ressemble très fortement à un "hack" mais qui est satisfaisant pour le cas d'utilisation. Encore une fois, on peut voir que Groovy nous facilite grandement la tâche, cette fois ci en nous permettant de contrôler ce qui est légal à utiliser dans nos script à travers "l'ASTSecurityCustomizer".

Nous avons décidé que le temps serait une date et une heure allant jusqu'à la seconde. Nous ne pensons pas que le domaine requiert une plus haute précision, en effet ces scripts sont destinés à jouer des simulations sur plusieurs heures et générer beaucoup de données d'un coup. Le temps joue un rôle important puisque c'est de sa gestion dont va dépendre le

réalisme des données enregistrées. Nous avons donc fournit à l'utilisateur la possibilité de configurer une durée de simulation représentée par une date de début et de fin. Il peut aussi spécifier la fréquence d'envoi des données par groupement de senseurs afin de pouvoir s'adapter au mieux à tout type de senseurs réels. Ce dernier point soulève un problème, impactant le réalisme des données obtenues (et donc l'exactitude du domaine métier ?), en effet certaines lois que suivent les senseurs calcule un nouvel état à chaque envoi de donnée qu'importe la fréquence d'envoi, une place de parking peut ainsi passer par les états libre-occupée-libre en moins d'une seconde. C'est pourquoi nous introduisons une fréquence de changement d'état pour les lois de Markov qui enverra la même valeur tant que l'intervalle minimale entre deux changement d'état n'a pas été atteint. De ce fait, les données obtenues par l'expert sont plus réalistes.

Nous avons agrémenter la classe des nombres d'objets Duration afin que l'utilisateur puisse facilement exprimé des fréquences en fonctions d'unité de temps (secondes, minutes, heures) et qu'il n'ai pas à manipuler directement des timestamps. Ceci nous permet d'être très fluent sur des points techniques.

En effet exprimé une fréquence de cette façon "frequency 1 / 30.min" est optimal pour l'utilisateur.

Support de l'IDE

Nous avons utilisé le mécanisme de description du DSL fourni par Groovy qui permet à notre IDE d'être conscient de la syntaxe et nous donne de l'autocomplétion au niveau des déclarations des lois etc et de tous les paramètre de ces déclarations.

Ce système est efficace mais difficile à maintenir, malgré tout un développeur sera heureux d'avoir cette fonctionnalitée.

Validation

Chaque Builder possèdent une méthode qui valide ses paramètres, cette méthode est appelé sur tous les builders avant de construire les objets métiers et chaque erreur détecté est logger et empêche l'exécution des simulations. Ces méthodes de validation sont lourdes et entraînent une possible répétition de code, malgré tout ce système nous as permis de rapidement ajouter de la validation pour tout nos objets métiers. Un refactor de cette partie est à envisager.

Il nous as paru important d'aider l'utilisateur en lui indiquant les lignes du script qui contiennent des erreurs, malheureusement il n'y as pas de moyen simple d'implémenter cette fonctionnalité avec notre architecture. Nous avons alors fait un autre "hack" qui nous permet d'avoir ce comportement en appliquant des transformations au script avant de l'exécuter. Le résultat nous satisfait pour les cas d'utilisation de ce DSL.

Limitation

Parmis les limitations de notre DSL nous pouvons noter que toutes nos valeurs sont des valeurs numériques. Ceci ne représente pas le domaine que nous essayons de couvrir, il est

possible que certain sensors retourne des valeurs sous forme de chaîne de caractère par exemple. Notre architecture n'est pas un frein à cette évolution, nous avons en place les bon couple abstraction/généricité pour gérer la fonctionnalité, mais ce support entraîne des questions au niveau de la syntaxe de déclaration des lois que nous ne nous sommes pas posé (e.g. comment lier une valeur à un état d'une chaîne de Markov ?) et ajouterai beaucoup de validation nécessaire pour éviter les problèmes, notamment au niveau des composites. Une autre limitation est la gestion du bruitage des valeurs qui pourrait être global aux valeurs générées et que nous avons implémenter seulement pour quelques lois.

Bien que la déclaration de composant soit unifié, la déclaration des paramètres l'est moins. En effet certain composant requiert des listes, d'autres des closures, certain paramètre requiert des parenthèses, etc. Une solution serait d'utiliser les capacités de modification de l'AST de Groovy pour essayé d'unifier la déclaration de paramètre, ou du moins la simplifier.

Nous n'avons pas utiliser cette fonctionnalité de Groovy pour ce projet, c'est donc une piste de recherche importante.

Conclusion

Dans l'ensemble nous sommes satisfait de notre design ce qui nous permet de livrer aux experts métier un moyen de créer rapidement des jeux de données tests en peu de ligne de code avec un temps réduit d'apprentissage de la syntaxe.
L'utilisation d'un DSL interne pour notre projet nous as permis de gagner du temps d'implémentation puisque nous avons pu piocher dans les librairies et mécanismes existants, ce temps à été consacré à renforcer la sécurité du DSL. L'utilisation de Groovy plus particulièrement nous as fait gagné "quelques points supplémentaires" d'expressivité grâce à sa syntaxe permissive au niveau de la ponctuation, cela nous a aussi donné accès à de puissants pattern qui nous ont permis de réaliser des points d'extensions au DSL. Ce dernier pourrait donc être enrichi par des développeurs ayant des connaissances en Groovy.

DSL dans le contexte de SSL

D'après des études récentes l'Internet des objets est un secteur en expansion rapide d'ici à 2020 plus de 26 milliards d'objets pourraient être connectés à Internet. Les données mesurées par tous ces capteurs sont déterminantes pour la mise en place de la prochaine génération d'infrastructures urbaine comme les "Smart Cities" (villes intelligentes) ou les "Smart Grids" (Infrastructure énergétique intelligentes).
Les réseaux de senseurs sont chargés de mesurer les phénomènes physique et d'envoyer les valeurs à un intergiciel consommateur qui se chargera ensuite de redistribuer les données ou d'appliquer des transformations dessus.
La mise en place de tel intergiciel est une tâche complexe et qui est difficilement testable puisque cela nécessiterait de connecter l'intergiciel à un parc/réseau de senseurs pour mesurer son bon fonctionnement ce qui entraîne de forts risques de pertes de données. Il faudrait donc pouvoir mettre en place rapidement un environnement de tests virtuel pour ces intergiciel, sans avoir à le connecter à un parc de senseurs physiques tout en récupérant des données qui se rapproche de celles pouvant être collectés dans des situations réelles.

La mise en place de cet environnement de tests, doit être rapide, est destinée à des experts du domaine de l'IoT, nous proposons donc de créer un DSL pour répondre à ce problème.

Augmenter la productivité du développement

L'utilisation d'un DSL réduit l'expressivité d'un langage de développement classique, et de ce fait il est plus difficile de se tromper et plus facile de détecter les erreurs. Dans le cas de SSl, ce comportement est en effet observé, la définition des lois de génération de données sont interne au DSL et la syntaxe s'en trouve grandement simplifié. De plus, comme on contrôle le cycle de vie du DSL, on peut fournir un système de validation qui permet à l'utilisateur de pouvoir rapidement corriger les problèmes associés au domaine ainsi qu'à l'utilisation du DSL. L'utilisation d'un "modèle" qui représente le domaine couplé au DSL qui permet de manipuler directement ces abstractions facilite la compréhension du problème que l'on souhaite modéliser, et donc facilite sa résolution.
On pourrait critiquer l'utilisation d'un DSL en parlant de son temps de développement. Cependant, dans le contexte de SSL nous pensons que ce coût est rapidement amortie par le gain de productivité qu'il apporte
En effet, le contexte des IOTs comme vu dans l'introduction requiert de très nombreux scénarii de tests. Nous pensons qu'un DSL permet de rapidement mettre en place ces scénarii et ceci avec une qualité garantie par les mécanismes du DSL fourni.
Le problème le plus évident de l'approche DSL, bien que cette critique peut être faite pour toute API, est l'apprentissage de la syntaxe. Il incombe au DSL de rester simple et adapté au domaine. Nous défendons l'idée que dans le contexte de SSL l'utilisation d'un DSL est plus pertinente qu'une simple API. En effet, l'expressivité du DSL et sa capacité à mieux représenter le domaine qu'une API le rend au moins aussi simple à comprendre et utiliser pour un développeur.

Faciliter la communication avec les experts du domaine

Les abstractions fournie par les DSL permettent qu'un concept du domaine soit directement représentable dans le code, et ce avec moins d'ambiguïté que dans un GPL. La communication avec les experts du domaines est alors plus productive et les détails du domaines ont moins de chance de se perdre dans les communications.

Coût de l'abstraction

SSL est soumis au même contrainte que n'importe quel abstraction, il faut être conscient que le DSL est toujours en évolution et ne pas hésiter à retravailler ses abstractions plutôt que de faire entrer les nouvelles fonctionnalité de force dans des abstractions non adapté. Ceci peut entrainée par exemple des modifications dans la syntaxe qui peuvent perturber les utilisateurs mais qui sont nécessaire pour une évolution saine du DSL. Ceci est vrai dans le cas de SSL qui s'attache à résoudre des problèmes d'un domaine en forte croissance et capable d'évoluer rapidement.

Bibliographie

Domain Specific Language - Martin Fowler
DSL Engineering - Markus Voelter
ArduinoML, Sensor Simulation Lab - Sebastien Mosser