MGL843 Projet d'équipe

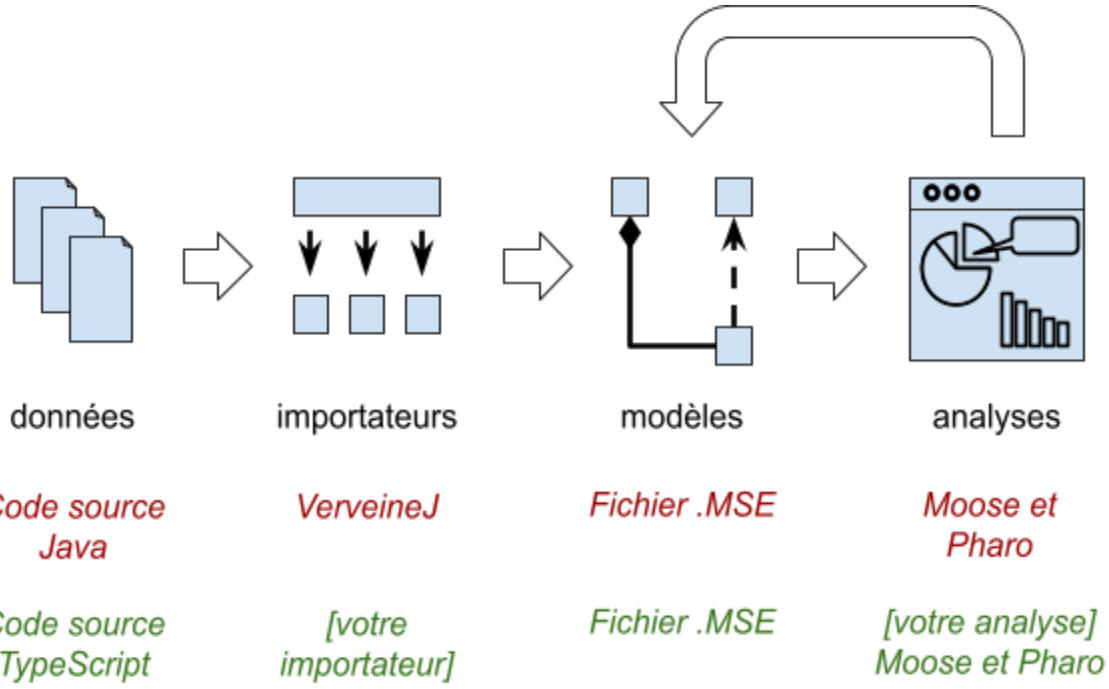
Étude empirique de projets TypeScript avec Moose

Le but de ce projet est de réaliser une étude empirique sur le volet conception de projet.s logiciel.s réalisé.s en TypeScript[1] avec la plateforme Moose.

Actuellement, il n'existe pas d'outil pour importer les modèles de code source TypeScript en Moose. Alors, l'étude empirique se fait une fois que vous avez réussi à faire un importateur de TypeScript pour Moose.

La figure suivante décrit un flux de travail pour réaliser une analyse. Elle est tirée du livre "The Moose Book".

- Les **éléments en rouge** représentent ce que l'on peut faire, par exemple, avec les outils existants pour analyser un projet écrit dans le langage Java.
- Les **éléments en vert** représentent le cadre de votre projet.



[1] ou d'autre langage moderne populaire comme Python, a discuter avec le professeur

Voici les volets du projet:

- Développer un importateur de code source (parser) pour le langage à étudier vers un modèle.
    - L'importateur doit parser du code (en TypeScript) et en générer un modèle
    - **Alternatif simple (raisonnable)**
      Utilisez le méta-modèle Famix 3.0 de Pascal Erni
        - Grâce à son API, le modèle est sérialisé en format **.mse**
        - Famix 3.0 (metamodèle pour Java) est documenté ici: https://hal.inria.fr/hal-00646884/document#page=34
        - Il est utilisé avec un vrai importateur abap2famix ici: https://github.com/pascalerni/abap2famix
    - **Volet facultatif (ambitieux)**
      Dans Pharo, définir et créer un métamodèle FAMIX pour TypeScript. Le modèle est défini dans FAMIX qui est un métamodèle défini dans le méta-métamodèle Fame (FMMM ou FM3). 😵
    - À l'aide d'un parseur TypeScript (p.ex. ts-morph dans npm), générer un fichier .MSE (à l'aide peut-être des mse-tools que j'ai créés) qui peut ensuite être importé dans Moose, selon le schéma ci-dessus.
- Définir le volet de l'analyse empirique de la conception à faire.
- Utiliser Moose et Pharo pour faire une analyse sur le plan de la conception de projet.s importé.s.

Le projet doit se faire en itérations (une itération doit se terminer à la date du livrable "Rapport avec démo intermédiaire" (voir le tableau de livrables).

Composition d'équipe

Il s'agit d'un projet en équipe de 4 à 5 personnes inscrites dans le cours.

Livrables et échéancier

Les livrables seront soumis dans Google Classroom, selon la date de remise. Voir le plan de cours pour la pondération de chaque livrable.

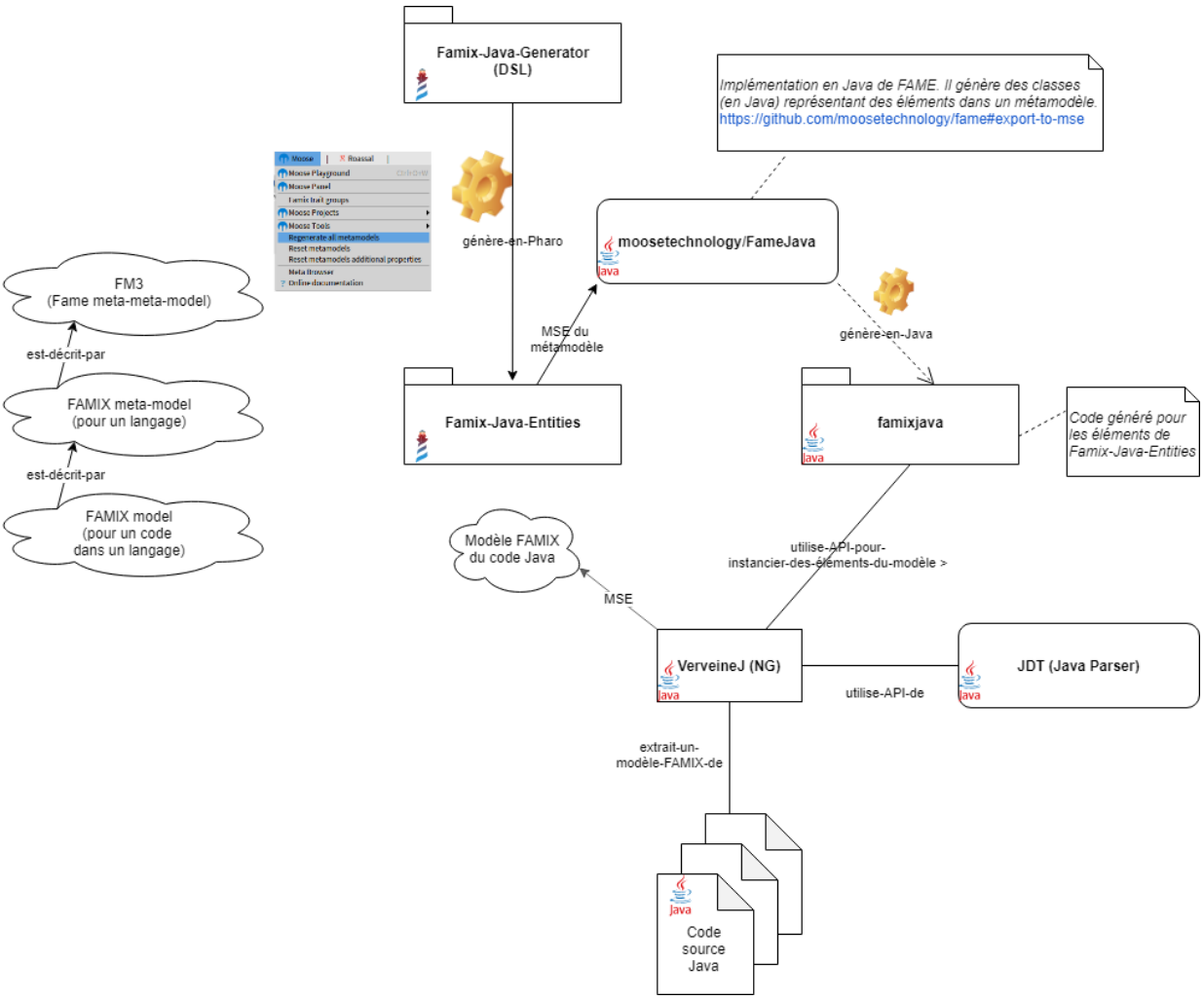
Échéancier (brouillon):

Livrable	**Séance (remise avant)**
Proposition de projet (composition d'équipe)	3
Rapport avec démo intermédiaire	7

Remise du code fonctionnel	13 (avant examen final)
Rapport de présentation finale	12 (présentation pendant la séance)

Ressources

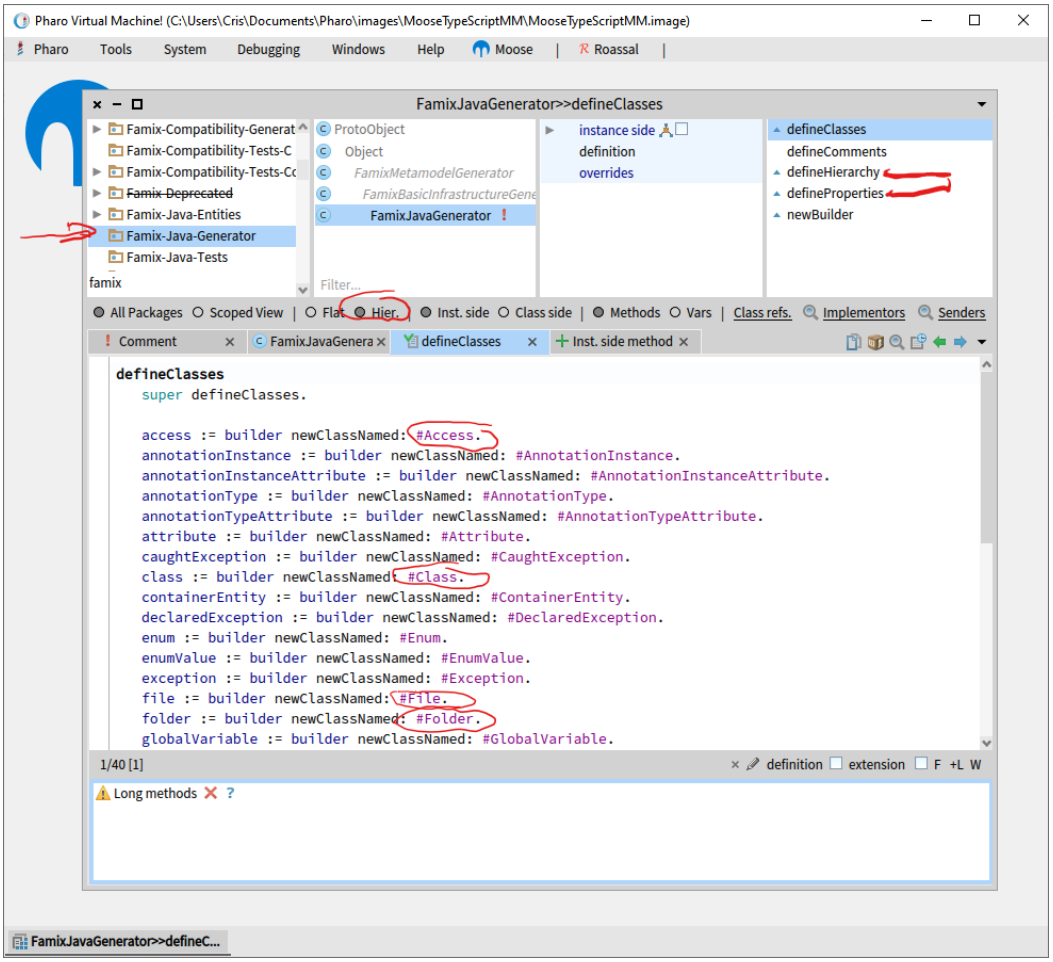
Schéma partiel de l'écosystème Famix/Moose



Métamodèles

- Contexte : [Rétro-ingénierie en informatique]
- Famix 3.0: [(PDF) MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family]

- FamixNG: [(PDF) Modular Moose: A new generation software reverse engineering environment]
- **Défi pour l'équipe ambitieuse** Comment créer un méta-modèle pour Moose: [Create a new Meta-model (FamixNG: since Moose 7)]
    - Vous pouvez vous inspirer du générateur de MM pour Java, FamixJavaGenerator qui est une sous-classe de FamixBasicInfrastructureGenerator.



Je ferai une présentation en classe pour cette partie.

    - Il faut reconnaître les différences entre Java et TypeScript. Pour commencer, incluez dans votre MM pour TypeScript les choses qui sont pareilles: Class, Method, Access, Reference (aux types), etc.

MSE

- MSE format (JSON est une possibilité) [File format | Moose wiki]
- Le projet [mse-tools] (prototype) qui a des classes de base TypeScript pour créer les objets représentant des éléments dans un fichier MSE
- Générer un MSE à partir d'un métamodèle : https://github.com/moosetechnology/fame#export-to-mse

Analyse syntaxique

- Qu'est-ce que l'analyse syntaxique (parsing)? Analyse syntaxique
- Qu'est-ce qu'un Arbre de la syntaxe abstraite (Abstract Syntax Tree, AST)?
- Visualiser l'AST d'un code TypeScript: TypeScript AST Viewer

TypeScript

- Syntaxe TypeScript : L'essentiel de la syntaxe Typescript en 10 min
- TypeScript Architectural Overview (wiki)
- Visualiser l'AST d'un code TypeScript: TypeScript AST Viewer, AST Viewers (ts-morph)
- Naviguer un AST: Navigating the AST (ts-morph), les visiteurs pourraient être utiles!
- Exemple rudimentaire en TypeScript de parcours de l'AST pour plusieurs fichiers TypeScript: https://github.com/fuhrmanator/ts2famix-exemple

Importateur

- Moose Java (comme exemple)
  - JDT2Famix
  - VerveineJ
- Comportement d'un importateur

Analyses empiriques

- Empirical Software Engineering (chapitre du Handbook of Software Engineering)

Pharo et Moose

- Tutoriel: Analyzing Java with Moose 8
- Analyses plus détaillées avec Moose: Analyze OO project
- Pharo MOOC: Live Object Programming in Pharo

Métamodélisation de TypeScript

Ici je note quelques points et des questions importantes pour la métamodélisation de TypeScript. Le but est de montrer pourquoi utiliser un métamodèle pour Java ne serait pas adéquat.
Cette section n'est pas complète. Votre travail est un problème ouvert. Il ne sera pas possible de faire un métamodèle TypeScript "complet", surtout dans le cadre du cours.

- Quelles sont les différences entre TypeScript et Java? ECMAScript 2015 joue un rôle important aussi.

    - Namespaces/Modules (TypeScript) vs Packages (Java)
        - Comprendre le problème général de “global namespace” en JavaScript
            - Pourquoi le pattern Module en JavaScript?
            - The Module Pattern - Learning JavaScript Design Patterns [Book]
        - Les Modules et les Namespaces dans TypeScript
    - D’autres types particuliers de TypeScript
    - Si vous utilisez un métamodèle Java pour analyser du TypeScript, alors il vaut mieux considérer les incompatibilités. Les liens plus bas ne contiennent pas toutes les réponses.
        - Comparing TypeScript to Java
        - …

Classes TypeScript pour son métamodèle

Le projet FameJava existe pour créer des classes Java représentant les éléments dans son métamodèle Fame-Java-Entities. C’est ce qu’utilise VerveineJ pour créer les modèles de Java. Ces classes peuvent générer la représentation en MSE de leurs instances (objets dans le modèle) automatiquement.

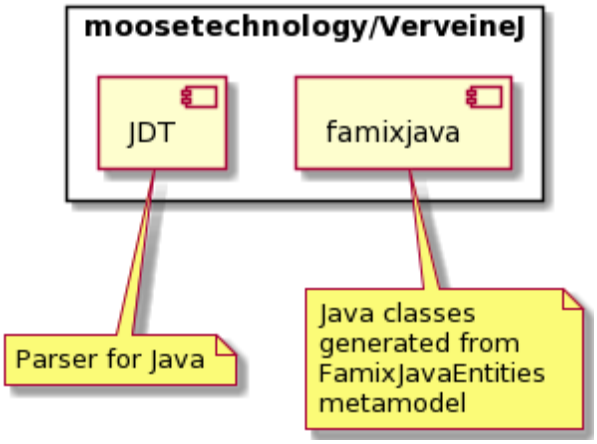
Dans le passé, un développeur (Pascal Erni) a modifié une ancienne version de FameJava pour générer les classes TypeScript qui, elles aussi, peuvent se sérialiser en MSE automatiquement. Cependant, sa solution est basée sur un métamodèle plus ancien qui ne contient pas tous les éléments de FamixNG (il n’y a pas de Traits par exemple). Il est donc incompatible avec FamixNG et un métamodèle moderne pour TypeScript.

Architecture d’un importateur

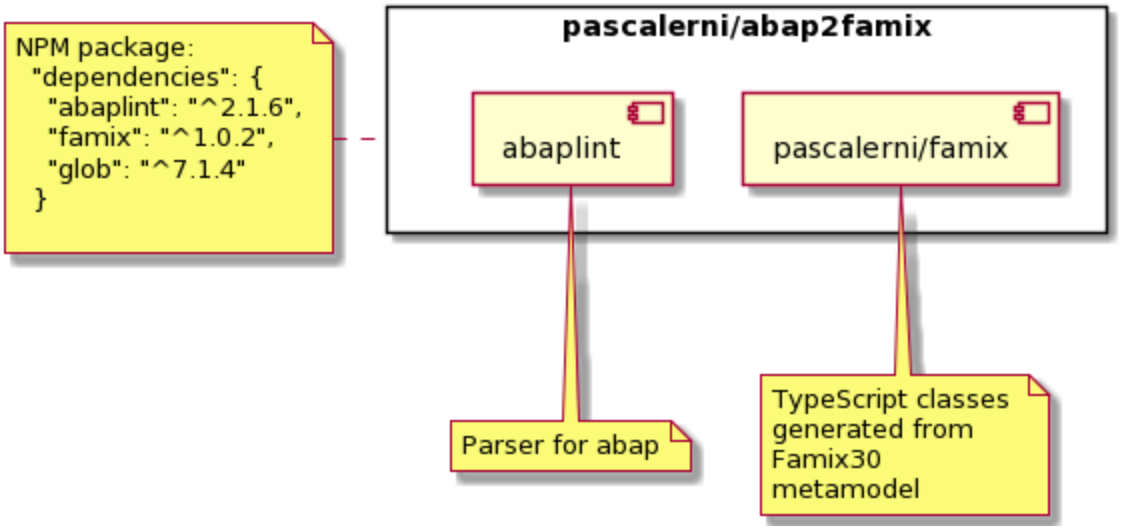
Un importateur doit

- analyser le code source (parse) et
- instancier des objets dans le métamodèle.

Voici deux exemples d’importateurs pour Moose. Le premier est VerveineJ, qui importe du Java:

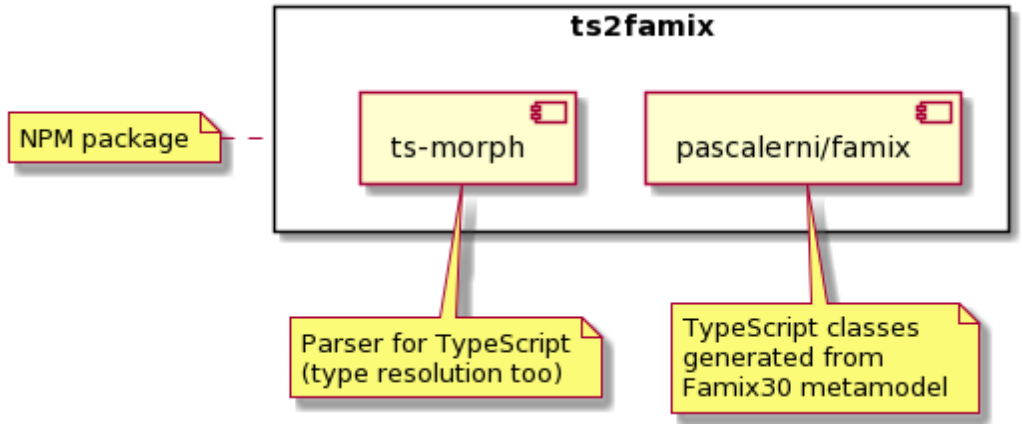


Le deuxième est abap2famix[2], qui importe du code abap (Advanced Business Application Programming, un langage orienté objet utilisé dans SAP):



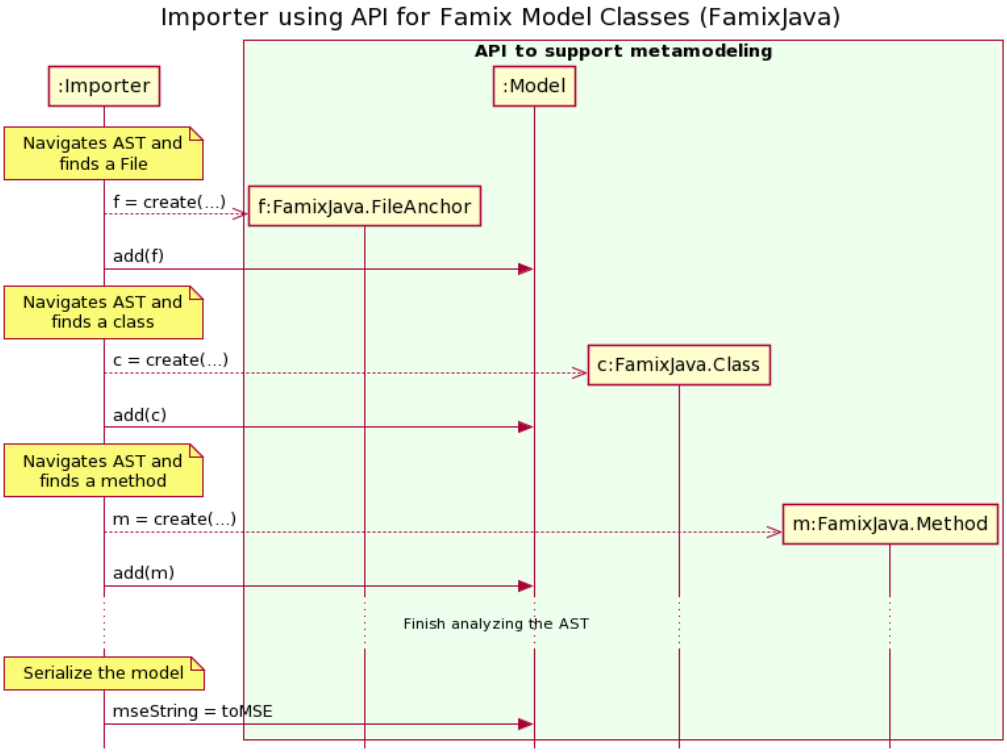
[2] "pascalerni/abap2famix: This project offers the mechanism ... - GitHub." 8 Oct. 2019, https://github.com/pascalerni/abap2famix. Accessed 9 Feb. 2021.

Voici une architecture d’une solution possible basée sur abap2famix



Comportement d’un importateur

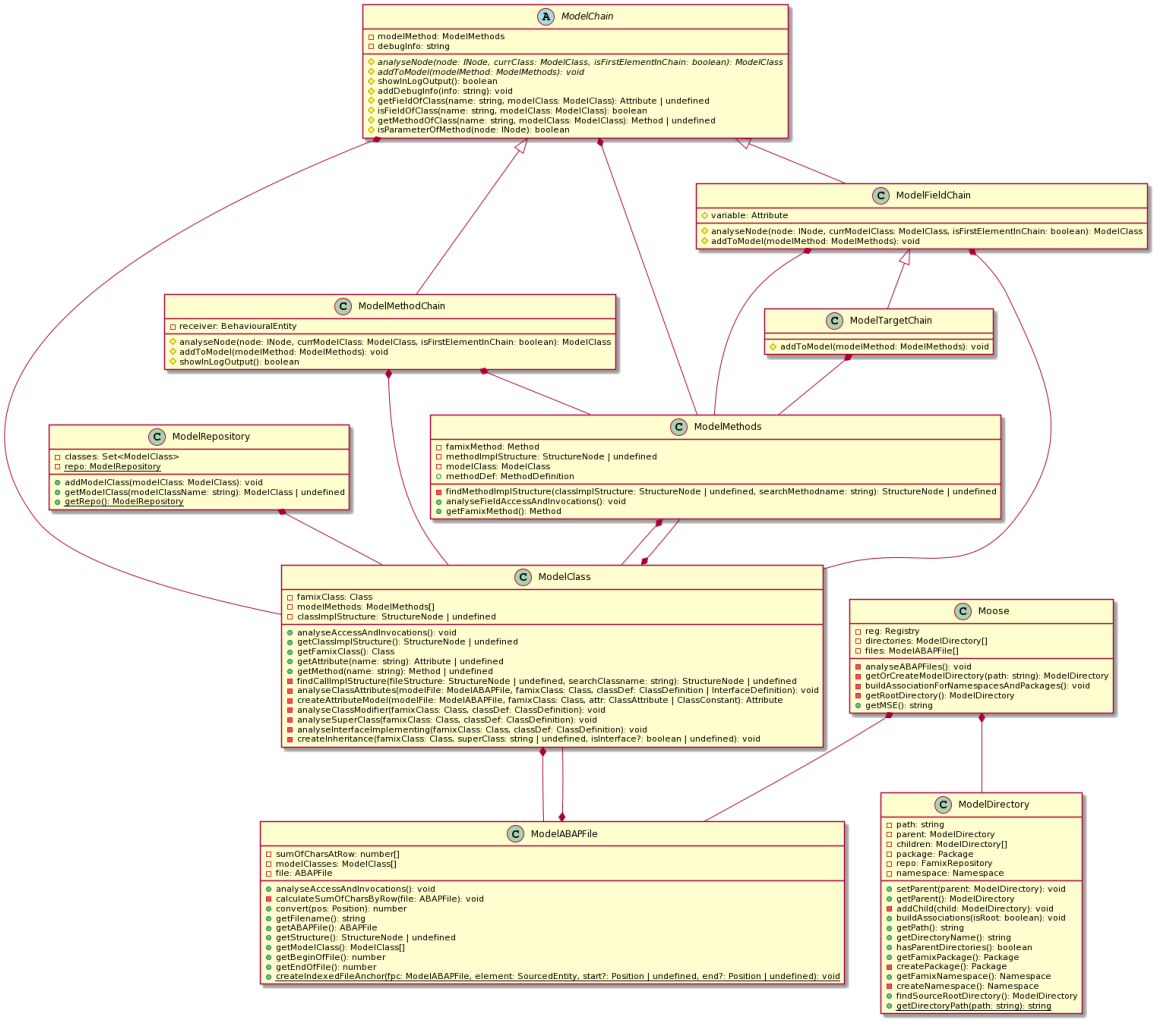
Voici un diagramme de séquence expliquant le comportement général d’un importateur. Les détails de comment un AST est généré et navigué ne sont pas considérés, car ça dépend du composant “parser”:



La solution de Pascal Erni (abap2famix) est intéressante car il est en TypeScript. Cependant, le métamodèle auquel elle correspond est une vieille version de Famix-Java (Famix 3.0)

documenté ici: https://hal.inria.fr/hal-00646884/document#page=34. Pour voir une utilisation plus concrète, il vaut mieux regarder le code de son importateur abap2famix.

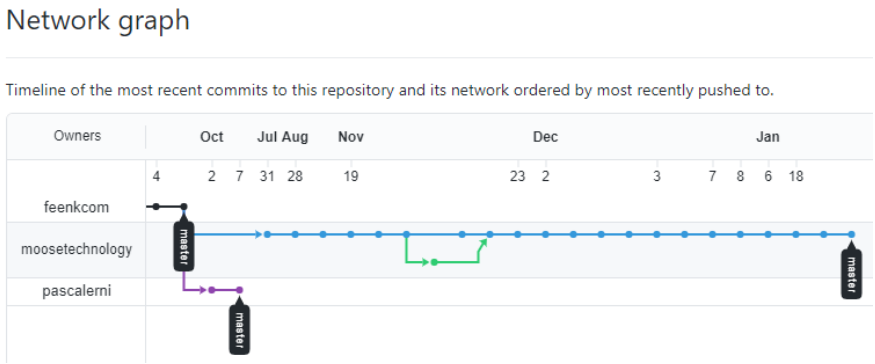
J'ai généré un diagramme de classe en PlantUML (avec l'outil bafolts/tplant: Typescript to plantuml) du code de abap2famix. Pour ceux qui vont essayer de faire un importateur basé sur cette architecture, cela peut être utile:



Questions et réponses

Q1: Est-il possible d'utiliser l'API TypeScript pour Famix (pascalerni/famix) de Pascal Erni avec un métamodèle FamixNG?

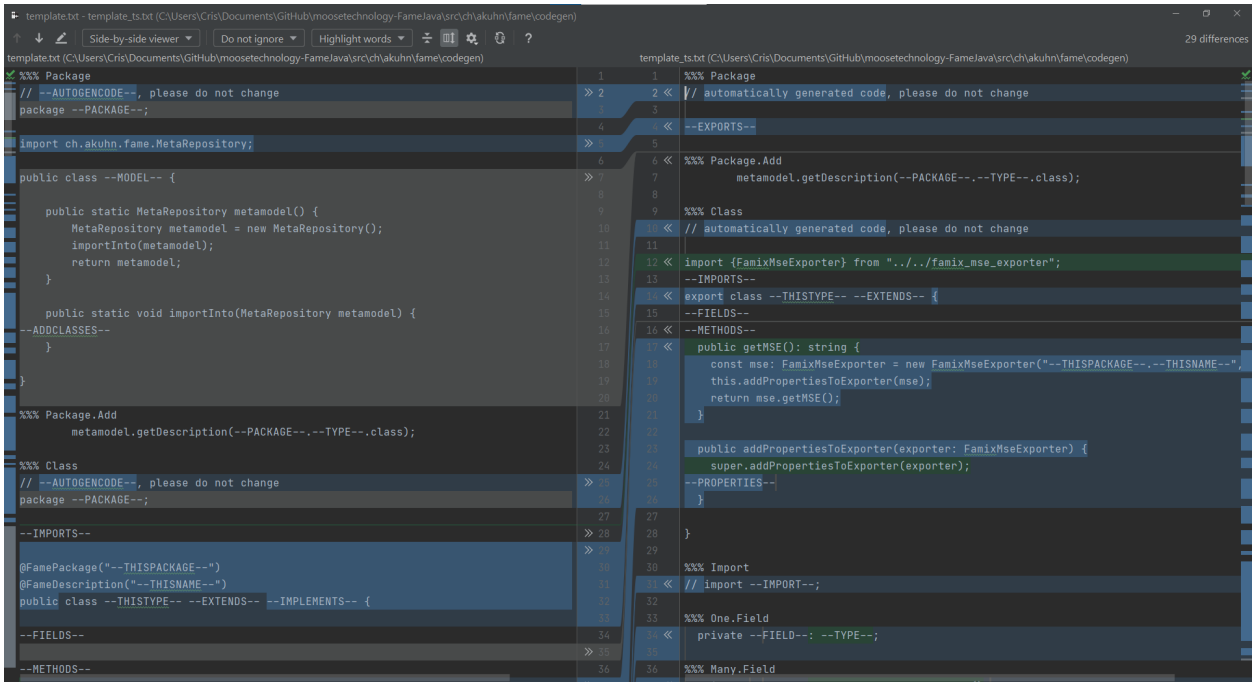
A: Non, pas dans son état actuel, car l'API a été générée par une version (fork) de FamixJava qu'il a modifiée pour TypeScript. Il y a une autre fork de FamixJava qui est utilisée par VerveineJ et qui peut générer une API avec FamixNG en Java. On peut voir ces deux forks dans l'image suivante (saisie de https://github.com/feenkcom/FameJava/network):



Il s'agirait de faire un merge des deux forks **pascalerni** (qui génère du TypeScript à partir du MM Famix 3.0) et **moosetechnology** (qui génère du Java à partir d'un MM FamixNG).

FameJava accepte (en intrant) une représentation d'un métamodèle en format MSE qui est généré par la bibliothèque Fame en Pharo.

Une approche pour supporter FamixNG serait d'essayer de reprendre la partie de génération de code de pascalerni/FameJava et de l'intégrer dans la version actuelle de moosetechnology/FameJava. La génération du code (dans les deux forks) est réalisée grâce à des gabarits (templates), par exemple ceci. Dans un outil qui compare les fichiers, on peut voir les différences entre le gabarit dans la version actuelle de moosetechnology/FameJava qui génère du Java (à gauche) et la version pascalerni/FameJava qui génère du TypeScript (à droite). Pour créer cette visualisation, j'ai utilisé IntelliJ IDEA. La saisie d'écran ne représente pas toutes les différences.



Q2: De façon générale, les concepts et entités en Java et TypeScript ne sont pas très différents. Est-ce qu'on peut ajouter la plupart du code DSL du générateur de Java (FamixJavaGenerator) dans notre générateur TypeScript?

A: Le métamodèle Famix-Java-Entities (FamixNG) est super complexe. Il n'y a pas actuellement un moyen simple d'obtenir une API générée automatiquement (voir la question plus haut). Alors, si vous essayez une solution basée sur un MM FamixNG (plutôt que la solution limitée avec Famix 3.0 pascalerni), je pense que vous avez un intérêt à appliquer le principe KISS (keep it simple stupid).

Q3: En TypeScript les namespaces sont seulement utilisés dans le sens organisationnel. Vous croyez qu'il serait pertinent de classer les modules comme packages ? Ou serait-il mieux de créer une nouvelle entité, par exemple, module ?

A: La réponse à cette question nécessite une certaine expertise en TypeScript. Il faut faire une hypothèse sur ce qu'est un Module en TypeScript (selon la documentation normalisée et non sur ce qui semble être une interprétation). Vous n'êtes pas obligés de modéliser parfaitement TypeScript (cela est difficile), mais une documentation (p.ex. Commentaires dans les entités) serait souhaitable pour qu'on comprenne les décisions.

Q4: On peut commencer à ajouter les parties du DSL de Java dans le générateur de TypeScript, puis essayer d'importer un MSE selon le même format que celui de Next Generation de Java. Croyez-vous que c'est une bonne idée ?

A: C'est très ambitieux de faire un nouveau MM en FamixNG (avec le DSL) pour TypeScript et de l'utiliser dans un importateur (parseur) comme ts2famix-exemple, car il n'est pas possible (en ce moment) de générer les classes représentant les éléments du métamodèle TypeScript (comme celle de Pascal Erni).

Pour minimiser les risques:

- Utilisez les classes de pascalerni/famix, malgré que ce soit pour un ancien métamodèle (Famix 3.0) Java.
- Documentez les limites et vos choix lorsque vous décidez de modéliser du TypeScript avec...

Cependant, si une équipe se sent plus ambitieuse, je voudrais bien l'accompagner.

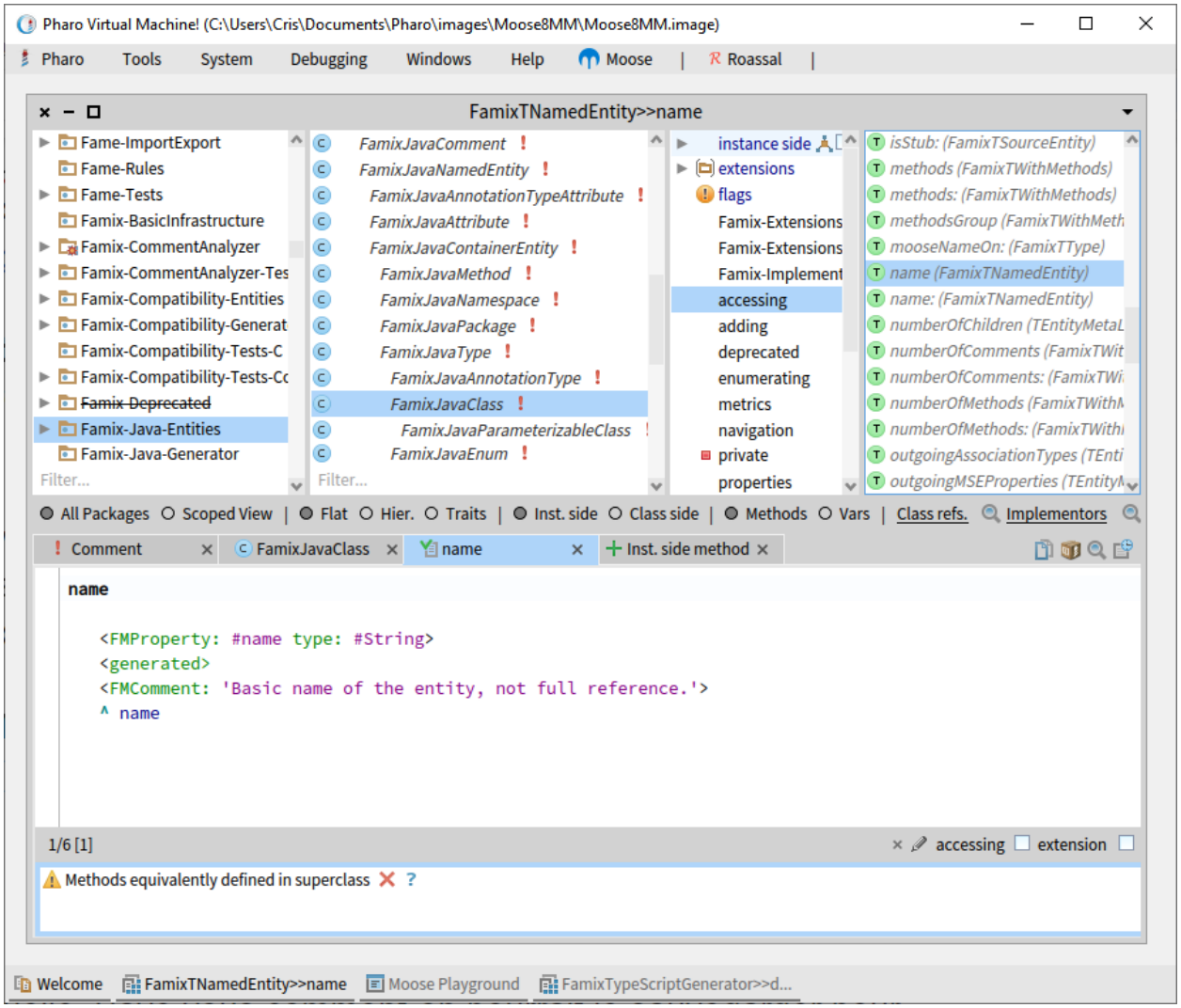
Q5: Pour générer le MSE depuis le nouveau métamodèle dans Pharo, est-ce qu'on a besoin de la nouvelle image de la machine virtuelle ? Ou s'il est possible de générer le MSE avec l'image qu'on a actuellement, pourriez-vous nous dire comment faire la génération ?

A: Ça prend la toute dernière image de Moose 8.0 (disponible via GitHub, pas encore disponible sur Jenkins du PharoLauncher normalement). Cependant, il est expliqué comment le faire en modifiant la Pharo Launcher pour télécharger l'image Moose de GitHub (release 8.0.1). https://github.com/moosetechnology/Moose#from-github-release

Q6: Dans la section du générateur (FamixNG) pour spécifier les «properties» du métamodèle, il n'y a aucune propriété comme «name». Est-ce qu'on peut déduire que cette propriété est déjà héritée de la super classe Famix?

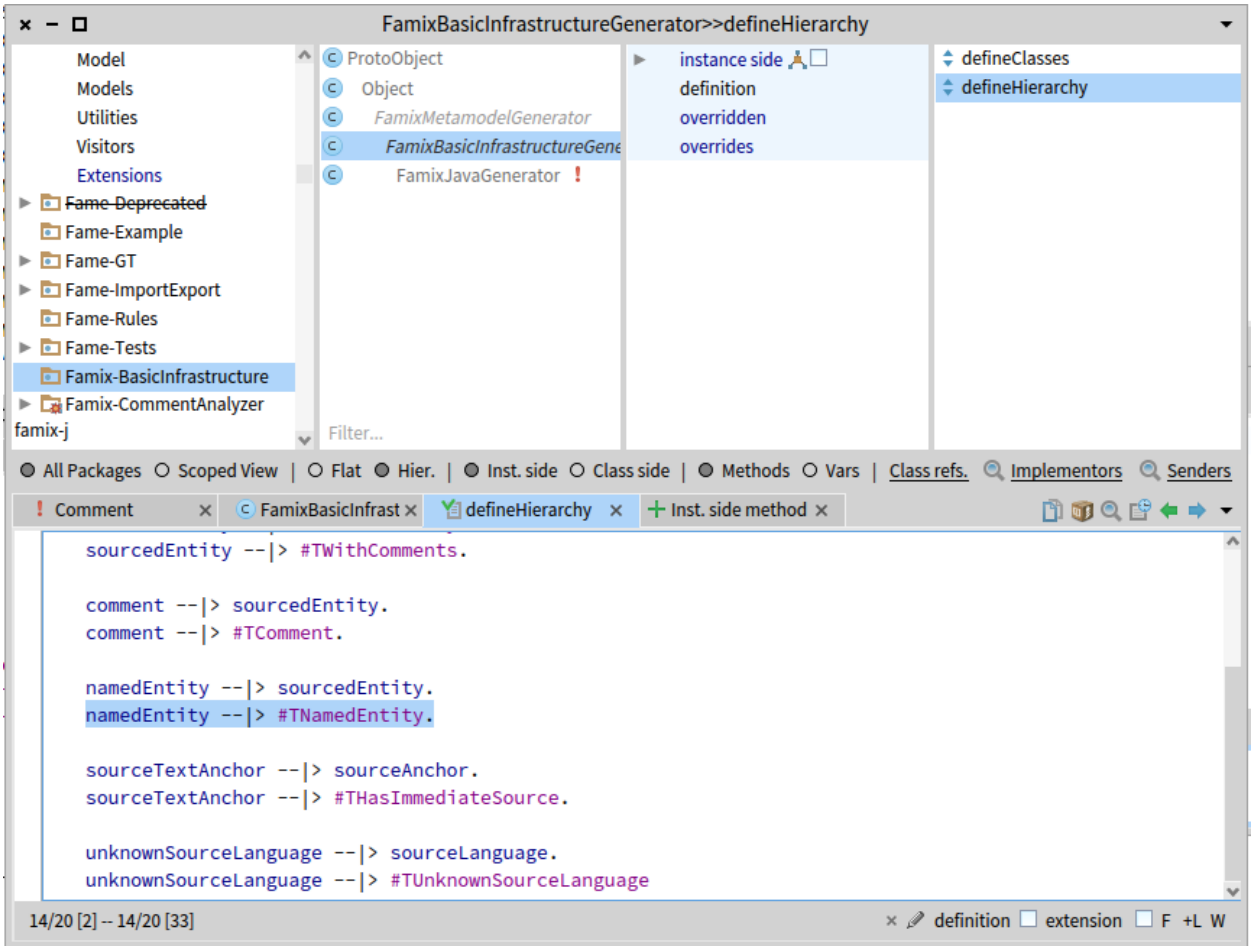
A: Le nom vient (dans Famix-Java-Entities) de FamixTNamedEntity (un trait). Le métamodèle que j'ai présenté dans le cours est super simple et n'a pas tout. Pour avoir une meilleure idée, tu peux regarder le MM de Famix-Java-Entities:

Tu navigues dans Pharo pour le MM de Java (package Famix-Java-Entities, FamixJavaClass>>#name) et tu verras son origine est un trait (T en vert). Voici une capture d'écran:



Cela veut dire que FamixJavaClass “hérite” son name. Voir https://modularmoose.org/moose-wiki/Developers/predefinedEntities#property-traits:~:text=FamixTNamedEntity pour plus de détails.

L’héritage n’est pas direct!
FamixJavaGenerator (qui génère les FamixJavaEntities) est une sous-classe de FamixBasicInfrastructureGenerator. On voit dans la hiérarchie de ce dernier que **namedEntity** est défini comme une entité qui hérite de **#TNamedEntity**:



C’est moins évident, mais on peut trouver les déclarations suivantes dans la hiérarchie de FamixJavaGenerator class>>defineHierarchy:

class --|> type.
…
type --|> containerEntity.
…
containerEntity --|> namedEntity.
…

Finalement, dans FamixBasicInfrastructureGenerator class >> defineHierarchy:

```
namedEntity --|> #TNamedEntity.
```

Q7: Je vois que ni le générateur ni le métamodèle créés pour TypeScript sont sauvegardés de façon permanente dans la machine virtuelle. Savez-vous comment on pourrait le sauvegarder pour qu'ils restent si on ferme la machine virtuelle ? Sur l'icône à côte du nom on trouve une petite boule jaune :

A: Cette icône veut dire que le package est "dirty", donc le code n'a pas été saisi dans Monticello (un outil de gestion des versions du code, comme git) et c'est normal.

Si tu as sauvegardé ton image (CTRL-SHIFT-S) tu vas avoir une copie du code dans la mémoire de l'image.

Monticello est une vieille façon de gérer les changements dans le code de Pharo. Depuis quelques années, Pharo supporte git (et GitHub) avec l'outil Iceberg. Comme tout outil git, il peut être compliqué à comprendre. Il est un peu plus compliqué en Pharo parce que les versions du code chargées dans la mémoire de l'image peuvent être différentes que celles sur disque (.git).
Si plusieurs coéquipiers dans ton projet veulent partager du code Pharo (sur un repo GitHub par exemple) c'est possible. Il y a un document à Manage Your Code with Iceberg expliquant Iceberg. Ce n'est pas une tâche triviale et ce n'est pas nécessaire pour le projet.

Q8: Concernant le Plan d'itération, on peut considérer le projet comme une seule grande itération de janvier à avril ou on considère des itérations d'un mois. Est-ce que le plan d'itération que nous devrons présenter (démonstration) concerne uniquement ce que nous allons faire du 19 février au 18 mars?

Si vous ne faites qu'une seule itération, c'est l'équivalent d'un projet en cascade, n'est-ce pas?

Tel que mentionné dans les diapos sur le "Walking skeleton", le but est de faire des itérations courtes (1 à 3 semaines, mais un mois ça va aussi je suppose) et dans chaque itération améliorer de plus en plus la solution de bout en bout après les démonstrations des choses qui fonctionnent.

Je considère pour vous que l'exemple ts2famix-exemple est une première itération, surtout si vous êtes allés jusqu'à importer le fichier MSE dans Moose (j'espère que vous avez essayé au moins ça pour le walking skeleton dans votre équipe).
Sinon, vous êtes libres de planifier vos itérations, selon les objectifs visés. Par exemple, une itération peut viser:
- intégrer l'API Famix 3.0 de Pascal Erni
- reconnaître les Classes/Méthodes/FileAnchors/etc.
- valider un comptage de ces choses dans Moose

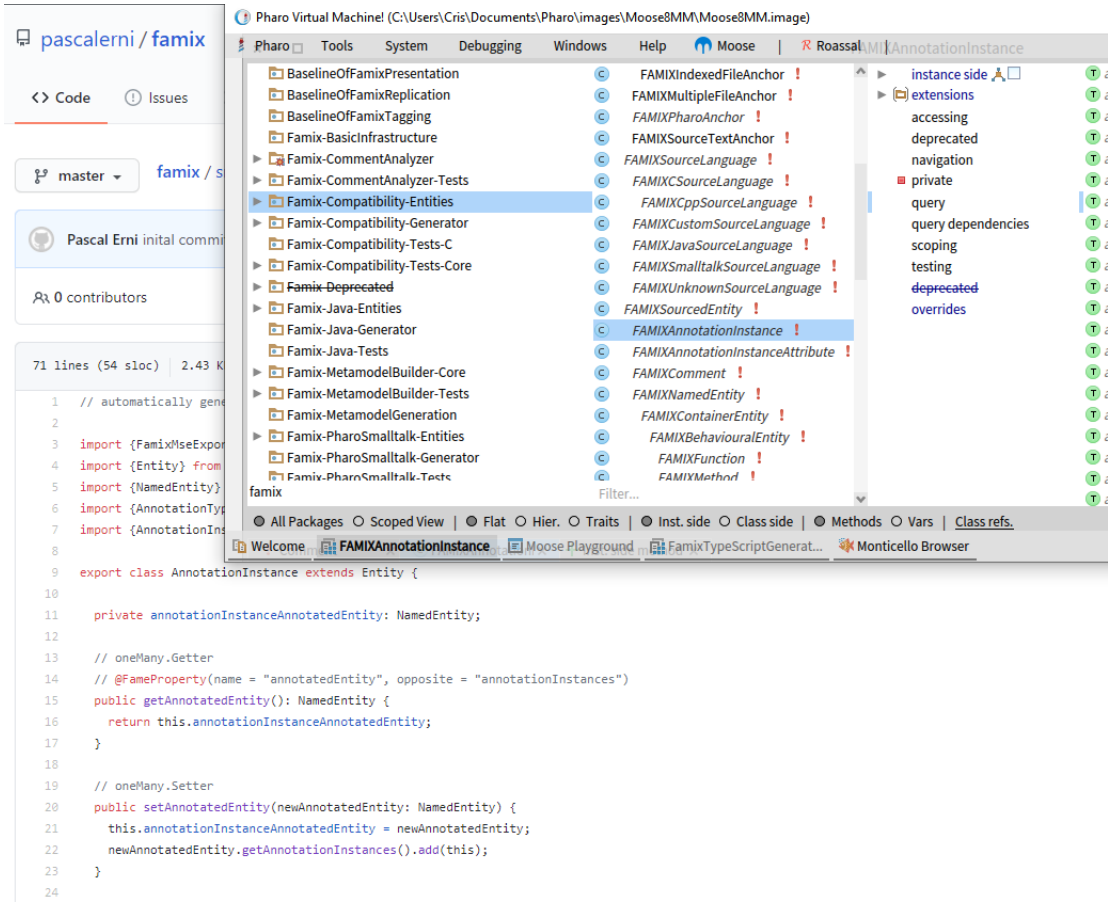
Pour la démo visé à la mi-session, il s'agit de la démo à la fin d'une itération. Alors, il faudrait synchroniser l'itération avec cette date.

Je prends pour acquis que vous avez appris à planifier les itérations pendant votre baccalauréat. Si vous avez besoin d'une mise à niveau, j'ai une liste de meilleures pratiques à https://drive.google.com/drive/folders/0B9aAbha28QkaamJDVDdSYXZLVmM?usp=sharing

Q9: Comment trouver des classes du métamodèle Famix 3.0 utilisé par Pascal Erni?

Les classes TypeScript de **pascalerni/famix** ont été générées avec Famix30.fm3.mse dans un autre dépôt dans le code ici:
https://github.com/pascalerni/FameJava/blob/f0c21dbca03558208f7ea586dcbea0c7c5c44a4a/src/ch/akuhn/fame/TypescriptGenerator.java#L12

Dans l'image de Moose 8, on peut trouver *l'équivalent* de ces classes (pour une *compatibilité* avec l'ancien Famix 3.0, en Moose 8 c'est FamixNG) dans le package **Famix-Compatibility-Entities**:



Q10: Comment générer un MSE pour un métamodèle Famix?

La réponse à cette question était dans la partie Ressources, mais je la répète ici:

- Générer un MSE à partir d'un métamodèle : https://github.com/moosetechnology/fame#export-to-mse