Contract

- Practical foundations of model management
- Model transformations
  - Model-to-Text
  - Model-to-Model
  - Metaprogramming
- Development of language interpreter / compiler
- DSLs and model management: all together (Xtext + Xtend / Langium + Typescript)

Avant de rentrer dans les détails, posons les objectifs concrets de cette section sur le Model Management.

Qu'est-ce qu'on va apprendre, manipuler, expérimenter ?

Cette partie du cours vous donne les fondations pratiques de la gestion de modèles : c'est-à-dire ce que l'on fait avec un modèle une fois qu'il est construit, et comment on exploite sa structure et son contenu.

Elle va couvrir plusieurs axes :

- Les transformations de modèles :
  - Model-to-Text (M2T) : par exemple, générer du code source (Java, TS, etc.) à partir d'un modèle.
  - Model-to-Model (M2M) : transformer un modèle dans un autre langage, ou dans une autre structure.
- La métaprogrammation :
  - manipuler des modèles pour produire du code dynamique,
  - ou personnaliser le comportement du langage lui-même.
- L'exécution des modèles :
  - soit par interprétation (exécution directe d'un modèle),
  - soit par compilation (traduction vers une cible exécutable).

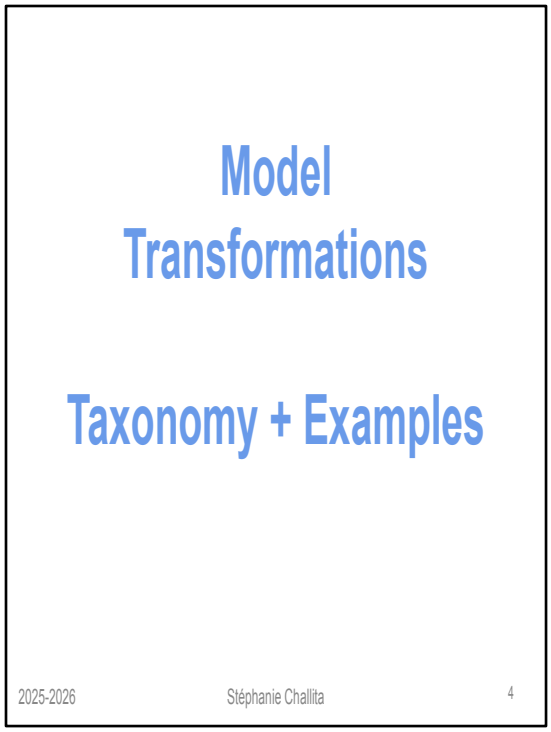
On verra également comment intégrer tout cela dans une démarche cohérente avec ce qu'on a vu précédemment sur les langages :

- Avec Xtext + Xtend, pour l'approche Java/Eclipse,

- Et avec Langium + TypeScript, pour une version web-native et LSP-compatible.

En résumé:

Cette partie vous donne les outils concrets pour faire vivre vos modèles :
les transformer, les exécuter, les relier au code — que ce soit pour générer, valider, ou interpréter.



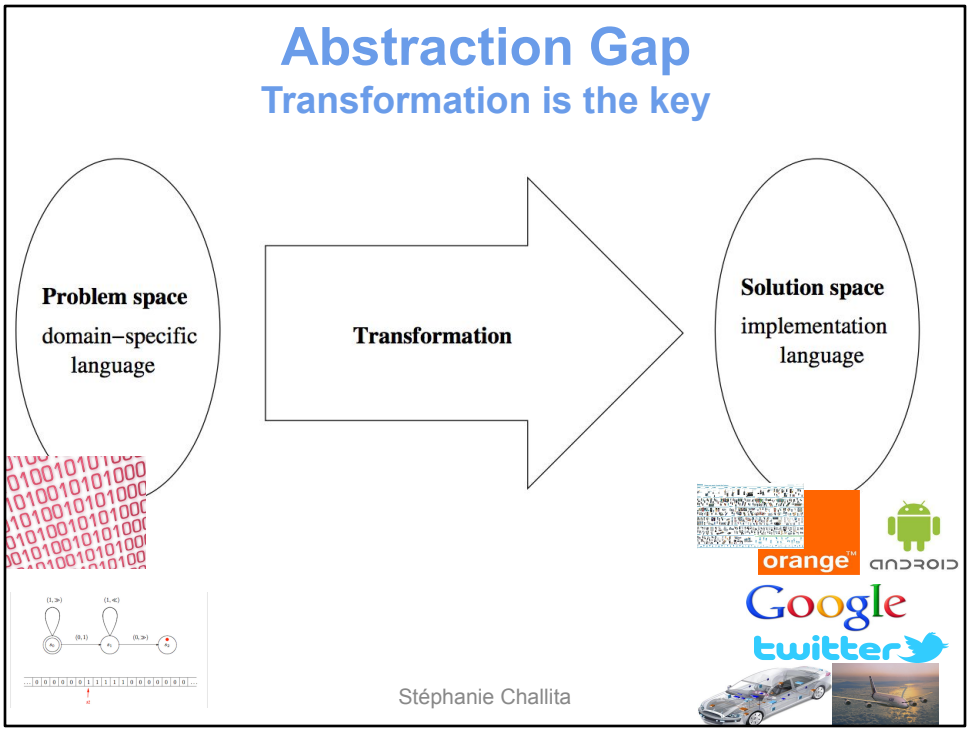
Entrons maintenant dans le cœur de la gestion de modèles :

Les transformations de modèles, ou model transformations.

C'est sans doute l'activité la plus fondamentale dans le cadre de la Model-Driven Engineering (MDE) :
prendre un modèle et en produire un autre artefact exploitable, qu'il s'agisse :

- d'un autre modèle,
- d'un programme,
- ou d'un fichier de configuration, d'interface, etc.

Mais toutes les transformations ne se valent pas :
il existe plusieurs types de transformations, qu'on peut organiser selon une taxonomie, que l'on va découvrir accompagné d'exemples.



On rappelle l'un des messages fondamentaux du cours.

Ce qu'on appelle l'abstraction gap, c'est l'écart qu'il y a entre :

- à gauche, le problem space, le domaine métier, avec ses propres concepts,
  → exprimés à travers un langage spécifique (un DSL, ou modèle abstrait).
- à droite, le solution space, c'est-à-dire la solution technique :
  → le code, les frameworks, les systèmes réels qu'on veut produire ou contrôler.

Entre les deux, il y a un fossé, une différence de niveau d'abstraction : le métier parle avec ses mots, le système s'implémente en Java, Android, sur le web, etc.

Et la clé, c'est la transformation.

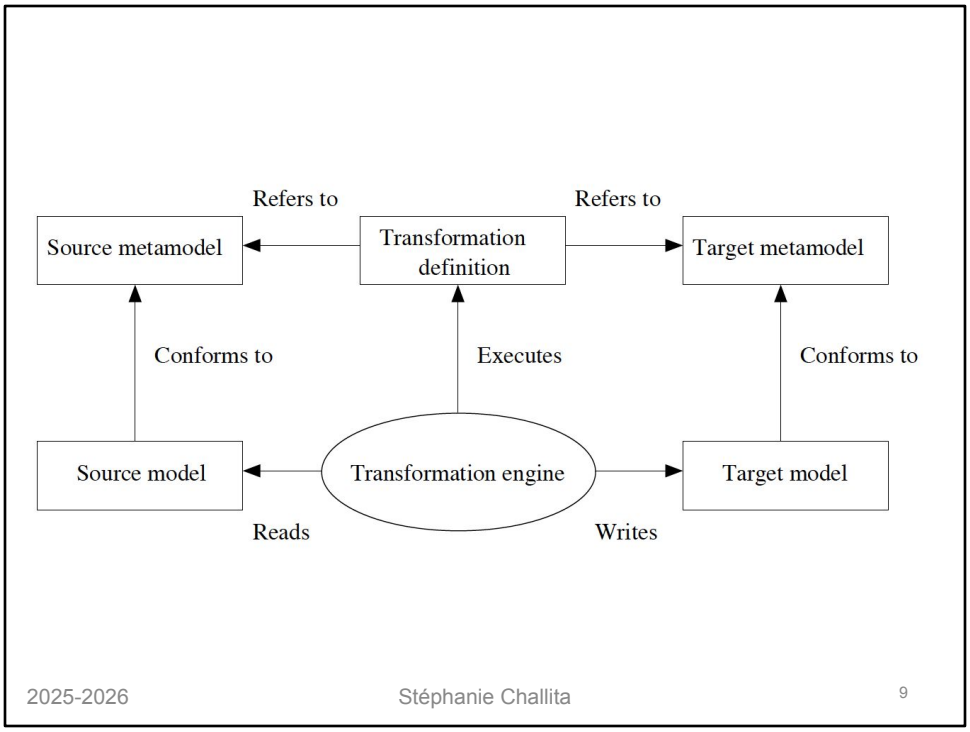
C'est elle qui permet de passer du problème à la solution, de manière automatisée, sûre et traçable.

C'est pour cela qu'on parle de Model Transformation dans l'ingénierie dirigée par les modèles :
on part d'un modèle dans le langage du métier,
et on le transforme en artefacts concrets : du code, des configurations, une UI, une simulation…

Cette idée est au cœur de tout ce qu'on a vu dans ce cours :

- La modélisation (EMF, Ecore…),
- Les DSLs (Xtext, Langium…),

- Les outils (Xtext, Langium, Eclipse Modeling…)



Ce schéma formalise le fonctionnement standard d'un moteur de transformation de modèles. Il est au cœur de l'ingénierie dirigée par les modèles (MDE) et de tout outil de transformation (comme ATL, Xtend, ou même certains usages de Langium).

🔹 À gauche : le modèle source, conforme à un métamodèle source.
C'est la description initiale, dans le langage du problème.

🔹 À droite : le modèle cible, qui doit être produit. Il est conforme à un métamodèle cible.
C'est une vue plus concrète, orientée vers l'implémentation.

Au centre : on trouve le moteur de transformation :

- Il lit le modèle source,
- exécute une définition de transformation,
- et écrit le modèle cible.

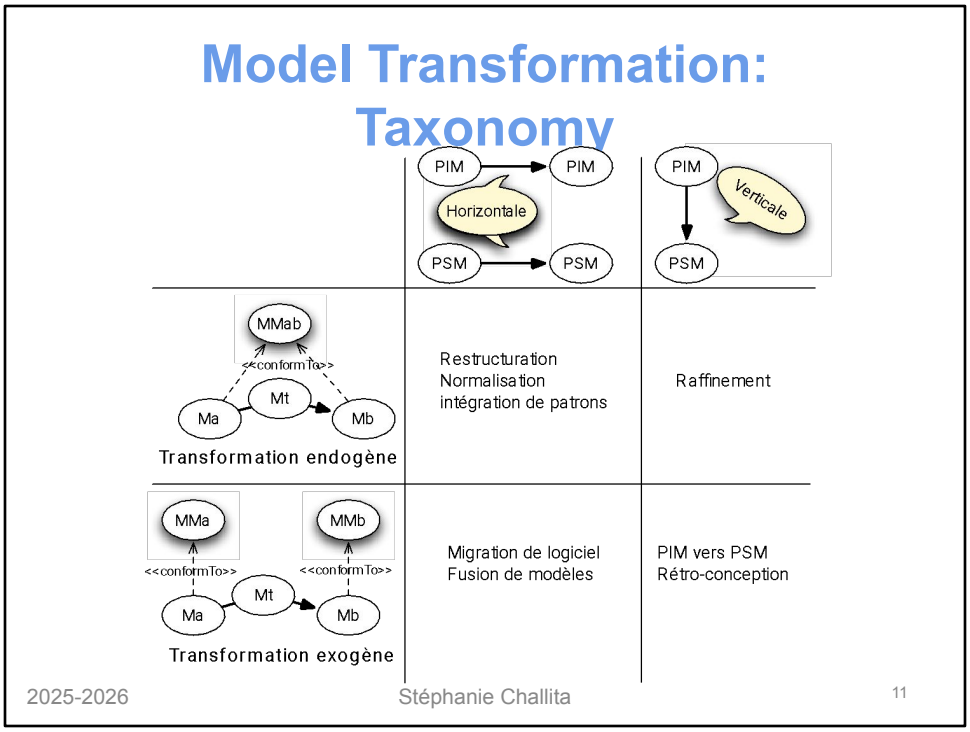
Cette définition de transformation est elle-même un programme, qui fait référence aux deux métamodèles (source et cible), et définit comment passer de l'un à l'autre.

Points à retenir :

- Le moteur n'interprète pas simplement des fichiers : il interprète des modèles conformes à un métamodèle.
- La transformation elle-même est déclarative ou impérative, selon la technologie (ATL, QVT, Xtend, etc.).
- Ce processus est générique : dès qu'on veut automatiser un passage entre deux niveaux (abstrait vers concret, problème vers solution), on peut mettre ce

- schéma en œuvre.

C’est une architecture clé pour tout ce qui suit : M2M, M2T, interprétation, génération de code.



Maintenant que l'on a vu ce qu'est une transformation de modèle, et comment on peut la catégoriser, on peut passer aux scénarios les plus fréquents qu'on rencontre dans un environnement MDE.

Ici, on introduit plusieurs cas pratiques très importants qu'on va explorer :

- Chargement de modèles :
  Lire un modèle à partir d'un fichier (souvent au format XMI), pour en obtenir une représentation manipulable en mémoire. C'est la base de tout traitement MDE.
- Sérialisation de modèles :
  C'est l'opération inverse : transformer un modèle en mémoire vers un fichier persistable, souvent au format texte ou binaire. Cela peut aussi vouloir dire "générer du code" à partir du modèle.
- Transformations modèle-à-modèle (M2M) :
  Un modèle en entrée est transformé en un autre, en suivant des règles définies — que ce soit pour raffiner une spécification, migrer un format, ou automatiser un passage d'un niveau d'abstraction à un autre.
- Transformations modèle-vers-texte (M2T) :
  On produit ici du code source, des documents, ou des fichiers de configuration à partir du modèle. C'est l'étape typique de génération d'un programme ou d'un artefact exécutable.

Ces scénarios sont au cœur des usages concrets des DSLs et de la MDE : on ne se

contente pas de dessiner des modèles — on les interprète, transforme, exécute.

Le reste de cette section du cours va s’appuyer sur ces scénarios, et montrer comment les mettre en œuvre avec Xtend ou Langium.

Model Transformation: Taxonomy

- Model-to-Model
- Model-to-Text
- Text-to-Model

Nous avons déjà vu que la transformation est l'élément central de l'approche MDE. C'est elle qui permet de passer d'un niveau d'abstraction à un autre, ou d'un langage à un autre.

On va voir une taxonomie simple mais très utile des types de transformations que l'on retrouve dans la pratique.

- Model-to-Model (M2M)

On part d'un modèle source conforme à un métamodèle donné, et on obtient un modèle cible, souvent conforme à un autre métamodèle.
C'est typiquement ce qu'on fait lorsqu'on raffine un design, migre un modèle, ou effectue une restructuration.

- Model-to-Text (M2T)

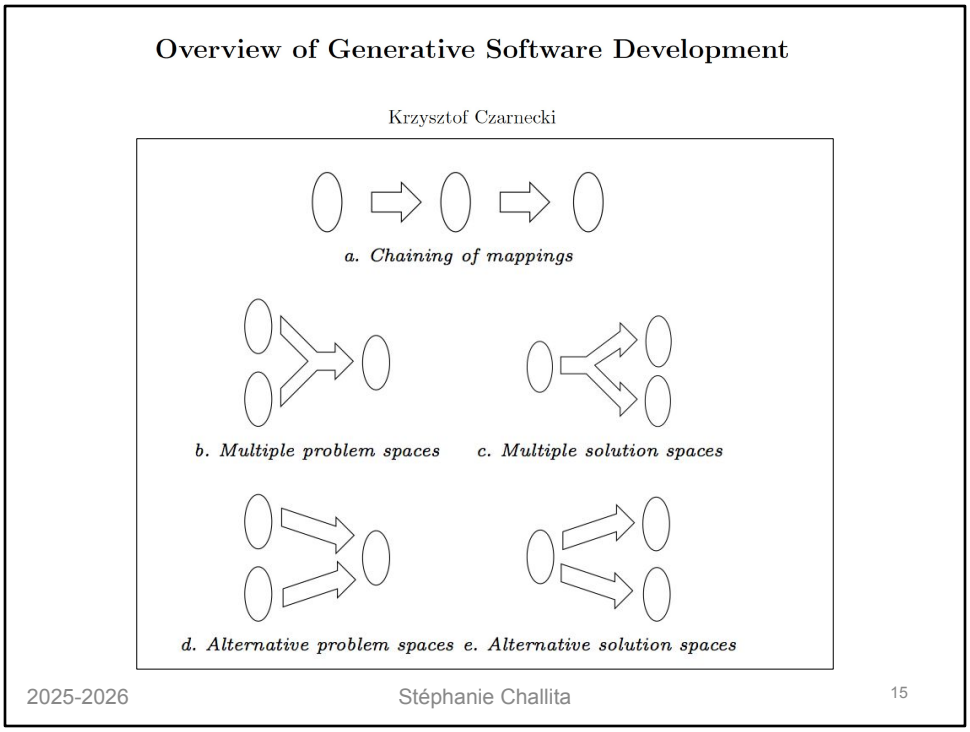
Il s'agit ici de générer du texte (souvent du code source, mais aussi des rapports ou des scripts) à partir d'un modèle.
Cette étape est cruciale dans une chaîne MDE complète : le modèle devient exécutable via une génération.

- Text-to-Model

Enfin, il y a des cas où l'on veut construire un modèle à partir d'un texte.
C'est exactement ce que permettent des outils comme Xtext ou Langium : ils analysent un fichier source, et produisent un modèle EMF ou Langium en mémoire.

Ces trois formes de transformation sont complémentaires et permettent d'instancier une chaîne complète de développement basée sur des modèles.

On va les illustrer plus en détail dans les slides qui suivent.



Voici les principales dimensions du développement génératif, selon une classification de Krzysztof Czarnecki.

Il illustre différents scénarios de transformation dans le cadre de la MDE.

- a. Chaining of mappings

Il s'agit du scénario classique de transformation en chaîne.
Un modèle est transformé en un autre, puis en un autre, dans une suite de raffinements successifs.
→ Typiquement : du DSL métier → à un modèle technique → puis à du code.

- b. Multiple problem spaces

Ici, plusieurs espaces de problèmes sont combinés dans une même solution.
→ Par exemple, un DSL pour le comportement et un autre pour l'interface utilisateur, combinés pour produire une application cohérente.

- c. Multiple solution spaces

Un même modèle peut donner lieu à plusieurs sorties.
→ Exemple : générer à la fois une application mobile, un site web, et une API backend à partir d'un même modèle métier.

- d. Alternative problem spaces

Plusieurs représentations possibles d'un même problème, selon différents points de vue.
→ On peut avoir plusieurs langages métiers (DSLs) exprimant la même intention.

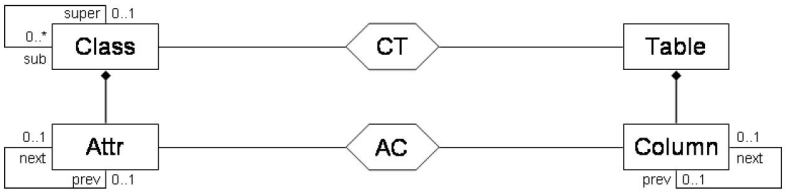
- e. Alternative solution spaces

À l’inverse, un même problème peut être transformé selon différents critères pour obtenir des solutions adaptées à des plateformes ou contraintes spécifiques.
→ Par exemple, optimisation pour Android vs optimisation pour systèmes embarqués.

Alors on peut dire que le développement génératif ne se limite pas à une simple transformation linéaire, mais qu’il s’inscrit dans un espace plus large, où les langages, les plateformes, et les usages se combinent de façon riche et flexible.

Andy Schürr, Felix Klar “15 Years of Triple Graph Grammars.” ICGT 2008

(declarative; bi-directionnal; model-to-model)



Ici, on introduit un formalisme central pour les transformations modèle-à-modèle bidirectionnelles : les Triple Graph Grammars, ou TGG.

🔹 Qu’est-ce que c’est ?
Les TGG sont un formalisme déclaratif permettant de spécifier une correspondance entre deux modèles — typiquement issus de deux métamodèles différents — et de maintenir cette correspondance dans les deux sens.

C’est-à-dire :

- Si je modifie le modèle source, je peux mettre à jour automatiquement le modèle cible.
- Mais aussi : si je modifie le modèle cible, je peux rétro-propager la mise à jour dans le modèle source.

🔹 Le schéma présenté ici
On voit trois parties :

- À gauche, le modèle source, avec un métamodèle orienté objet : `Class`, `Attr`, etc.
- À droite, le modèle cible, typiquement un schéma relationnel : `Table`, `Column`, etc.
- Au centre, les correspondances (`CT` et `AC`) entre les éléments des deux modèles.

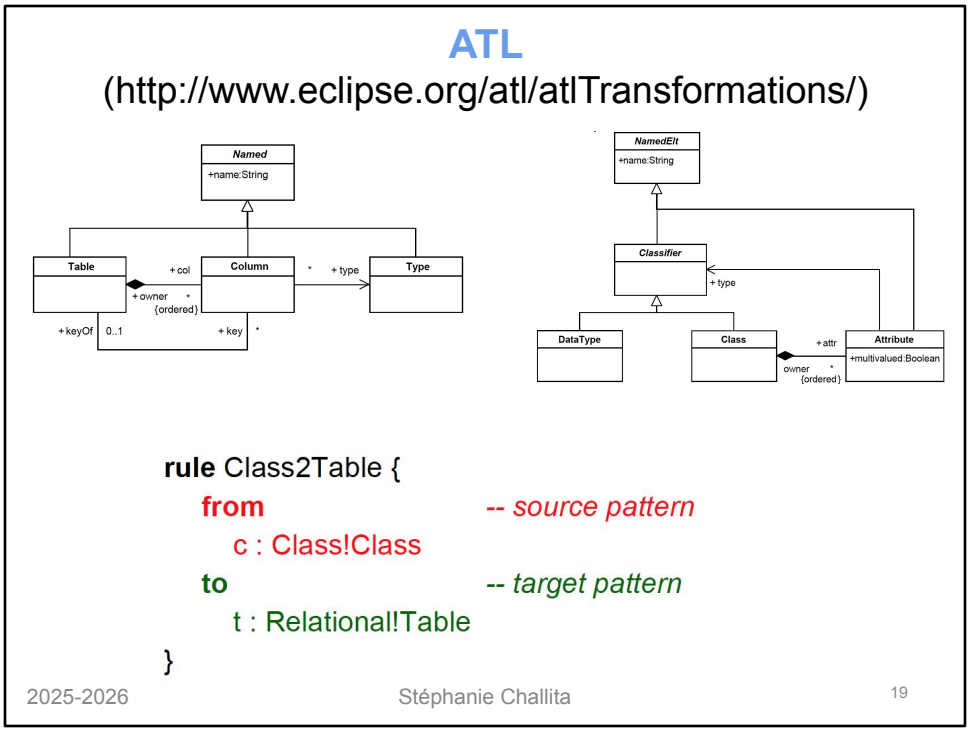
Chaque triplet (`Class`, `CT`, `Table` ou `Attr`, `AC`, `Column`) représente une règle de transformation déclarative.

🔹 Pourquoi c’est important ?

- Ce type de transformation permet d'assurer la synchronisation entre deux représentations.
- C’est un outil clé dans des cas d'ingénierie bidirectionnelle, comme l’alignement UML / base de données, ou pour des processus de co-ingénierie modèle/code.

🔹 À retenir :

- Les TGG sont modèle-à-modèle, bidirectionnels, et déclaratifs.
- Ils s’appuient sur des triples graphe (modèle source, correspondance, modèle cible).
- Ils sont à la base de frameworks comme eMoflon pour la synchronisation de modèles.



Ici, on introduit ATL (Atlas Transformation Language), un langage de transformation modèle-à-modèle très utilisé dans l'écosystème Eclipse.

🔹 À droite, on a le métamodèle source, orienté objet : on y retrouve les concepts classiques de la modélisation UML, avec des classes, des attributs, des types, etc.

🔹 À gauche, le métamodèle cible, de type relationnel : on y trouve des tables, des colonnes, des types SQL, etc.

En dessous, on voit une règle ATL :

```
rule Class2Table {
 from
   c : Class!Class   -- pattern source
 to
   t : Relational!Table  -- pattern cible
}
```

Cette règle dit simplement :

> "Pour chaque élément de type `Class` dans le modèle source, on crée un élément `Table` dans le modèle cible."

Ce qu'il faut retenir :

- ATL permet de définir des règles de transformation très lisibles, qui associent des patterns source à des patterns cible
- Cela s'intègre dans une exécution automatique, au sein d'un moteur ATL.
- Et ce type de transformation est essentiel pour passer d'un modèle conceptuel

- métamodèles de part et d’autre.

C’est un exemple typique de transformation exogène et verticale, comme vu précédemment dans la taxonomie.

Quiz Time

Characterize the following model transformations

Endogeneous? Exogeneous?
Vertical? Horizontal?
Model-to-text? Model-to-Model?

On va faire un petit quiz pour vérifier que vous avez bien compris les notions fondamentales autour des **transformations de modèles**.

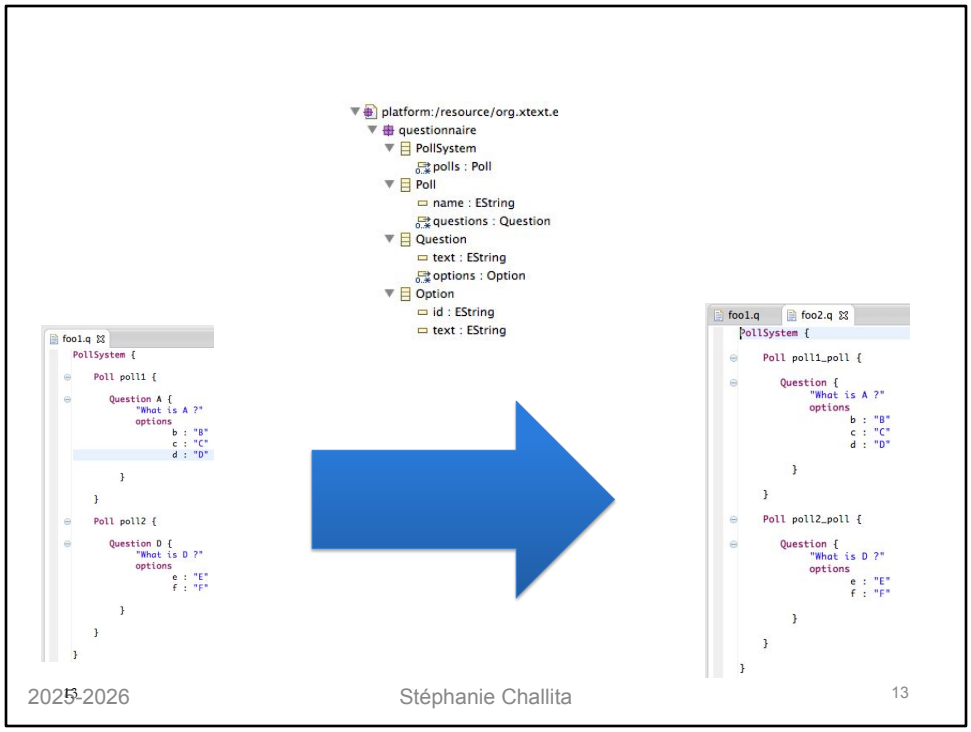
🔍 Pour chaque transformation qu'on va vous présenter, vous devrez répondre à trois questions simples :

1. **Est-ce une transformation endogène ou exogène ?**
   Autrement dit : est-ce qu'on reste dans le même métamodèle (endogène), ou est-ce qu'on change de métamodèle (exogène) ?
2. **Est-elle verticale ou horizontale ?**
   Est-ce qu'on affine un modèle (verticale), ou est-ce qu'on le restructure ou normalise à même niveau d'abstraction (horizontale) ?
3. **Quel est son type technique ?**
   Transformation **modèle-vers-modèle** ou **modèle-vers-texte** ?

💡 L'idée, c'est de mobiliser toutes les notions vues jusqu'ici :

- la distinction entre PIM et PSM,
- la notion de métamodèle,
- la structure d'une transformation,
- et les différents types d'opérations associées (raffinement, restructuration, génération de code…).

On vous laisse quelques instants pour réfléchir sur chaque exemple, puis on corrige ensemble.



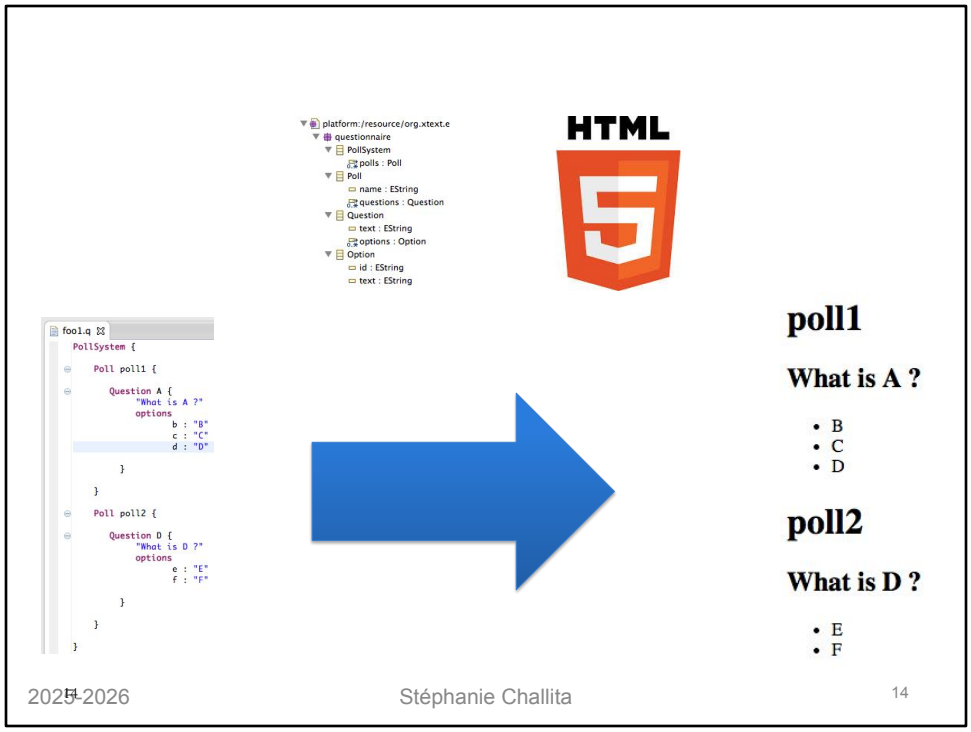
On reprend notre modèle PollSystem, défini à partir d'un fichier `.q`, et on applique une transformation qui modifie légèrement ce modèle — ici, en ajoutant simplement le suffixe "`_poll`" aux noms de tous les sondages.

Analysons ça :

1. Endogène ou Exogène ?
   → Endogène :
   On transforme un modèle vers un autre dans le même métamodèle (le métamodèle `PollSystem`). On ne change pas de structure globale ou de langage cible.

2. Verticale ou Horizontale ?
   → Horizontale :
   Le niveau d'abstraction reste le même, c'est une transformation au même niveau (de M1 à M1). Ce n'est pas un raffinement, ni une génération de code. On restructure légèrement le modèle.
3. Model-to-Model ou Model-to-Text ?
   → Model-to-Model :
   On part d'un modèle `.q`, on le charge comme un modèle, on le transforme en mémoire, et on le sauvegarde comme un autre `.q`. Même s'il est re-serialisé en texte, c'est une transformation de modèle.

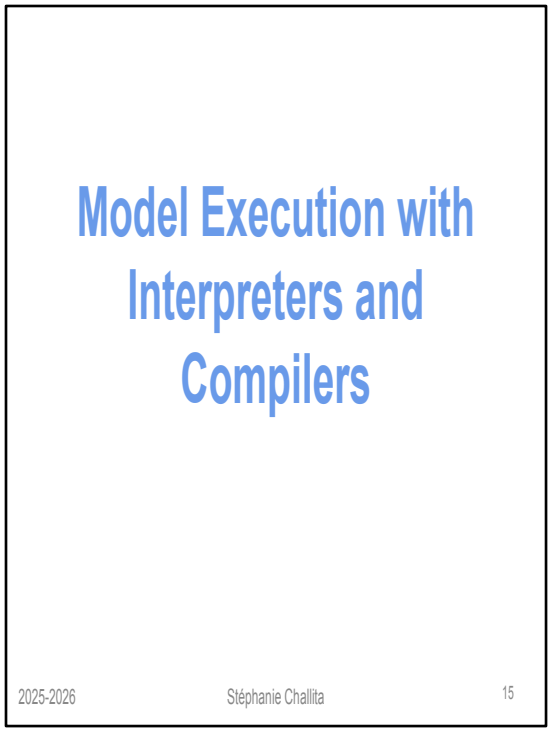
On est dans un cas très typique de nettoyage ou transformation structurelle douce.



Cette fois, on prend un modèle de type PollSystem — par exemple un fichier `.q` contenant des sondages — et on génère un fichier HTML pour que le sondage puisse être affiché dans un navigateur.

Analysons ça :

1. Endogène ou Exogène ?
   → Exogène :
   On part d'un modèle du langage `PollSystem`, et on génère du HTML — un langage différent avec un autre métamodèle (celui du DOM HTML ou du texte structuré).
2. Verticale ou Horizontale ?
   → Verticale :
   On passe d'un modèle abstrait à une représentation plus concrète — ici un fichier HTML qui s'exécute dans un navigateur. C'est un pas vers l'implémentation.
3. Model-to-Model ou Model-to-Text ?
   → Model-to-Text :
   Le résultat n'est pas un modèle EMF ou une structure manipulable en mémoire, mais bien un fichier texte, ici du HTML.



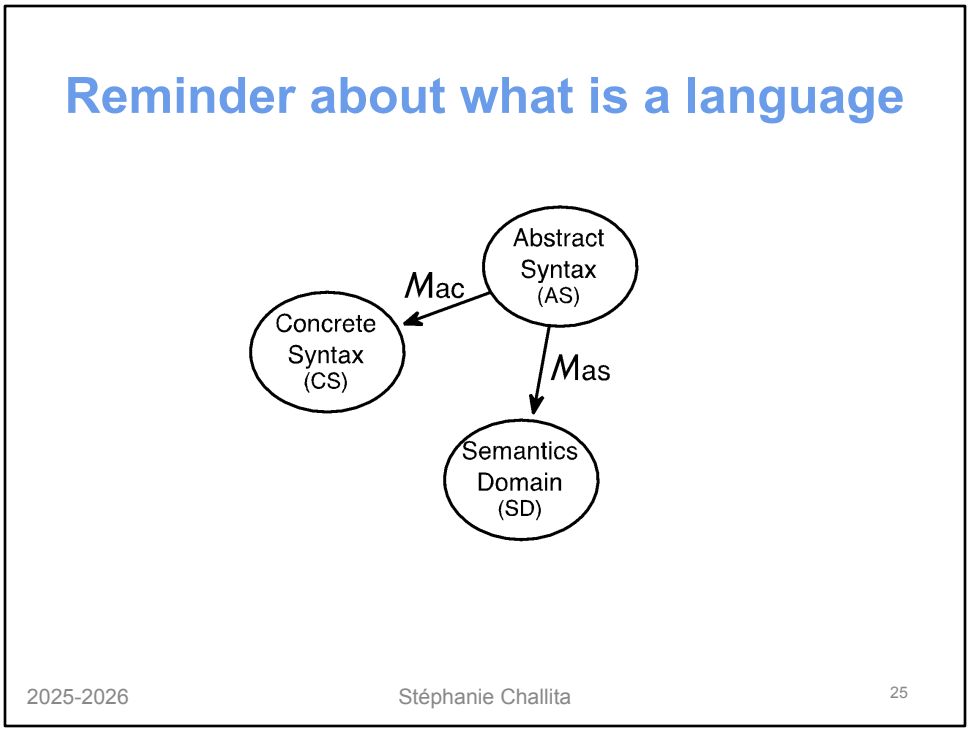
Entrons maintenant dans le cœur de la gestion de modèles :

Les transformations de modèles, ou model transformations.

C'est sans doute l'activité la plus fondamentale dans le cadre de la Model-Driven Engineering (MDE) :
prendre un modèle et en produire un autre artefact exploitable, qu'il s'agisse :

- d'un autre modèle,
- d'un programme,
- ou d'un fichier de configuration, d'interface, etc.

Mais toutes les transformations ne se valent pas :
il existe plusieurs types de transformations, qu'on peut organiser selon une taxonomie, que l'on va découvrir accompagné d'exemples.



Prenons un moment pour revenir à l'essence même de ce qu'est un langage.

Un langage, c'est plus qu'une simple grammaire ou une syntaxe colorée. Il se structure en trois composantes fondamentales :

- Concrete Syntax (CS) — la syntaxe concrète
  C'est ce que vous écrivez. Le texte, les mots-clés, la ponctuation.
  C'est ce qu'on voit, ce qu'on édite dans un fichier `.q`, `.java`, `.dsl`, etc.
  C'est souvent ce qui est défini par un fichier Xtext ou Langium grammar.

- Abstract Syntax (AS) — la structure abstraite
  C'est ce qui est compris par la machine.
  Une fois le texte analysé (parsé), on obtient une structure de données en mémoire, un modèle, souvent un arbre syntaxique (AST), ou une instance d'un métamodèle Ecore.
  C'est ce qu'utilisent les transformations, les validations, les navigateurs de modèle.

- Semantic Domain (SD) — le domaine sémantique
  C'est ce que signifie le modèle.
  Par exemple : exécuter une machine à états, calculer une valeur, afficher une interface…
  C'est ici qu'on implémente le comportement associé au modèle — via un interpréteur, un générateur de code ou une transformation.

Les relations entre ces éléments :

- `Mac` : le mapping concret → abstrait

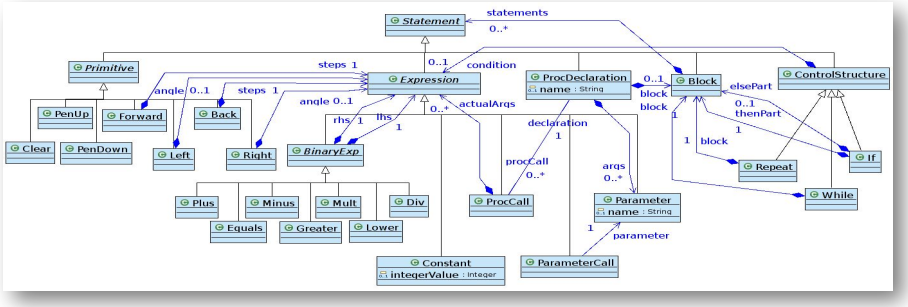
- On passe de la syntaxe concrète (CS) à la structure abstraite (AS).
  → C'est le rôle du parseur.
- `Mas` : le mapping abstrait → sémantique
  On donne du sens au modèle abstrait, en le connectant à une sémantique.
  → C'est le rôle de l'interprétation, des validations ou de la traduction.

Ce qu'il faut retenir :
Un langage, ce n'est pas juste un fichier texte. C'est une articulation entre :

- une syntaxe lisible par l'humain,
- une structure exploitable par la machine,
- et une sémantique qui permet d'agir.

C'est exactement ce que cherchent à structurer les outils comme Xtext, Langium, EMF ou encore Xtend.

Reminder about what is an abstract syntax



Maintenant qu'on a rappelé ce qu'est un langage, on zoome ici sur l'abstraction syntaxique — autrement dit, sur ce que comprend la machine une fois le texte analysé.

🔹 Ce que vous voyez ici, c'est un métamodèle, ou structure abstraite d'un langage visuel inspiré de LOGO ou d'un petit langage impératif.

Chaque boîte bleue représente une classe du métamodèle, c'est-à-dire un concept du langage :

- `Statement` : instruction générique.
- `Expression` : pour des formules arithmétiques ou booléennes.
- `ProcDeclaration` et `ProcCall` : pour déclarer et appeler des procédures.
- `If`, `Repeat`, `While` : structures de contrôle classiques.
- `BinaryExp` avec `Plus`, `Mult`, `Equals`, etc. : opérations binaires.

Les flèches indiquent :

- les relations de composition (par exemple : un bloc contient des statements),
- ou les liens typés (par exemple : `BinaryExp` a un `lhs` et un `rhs` de type `Expression`).

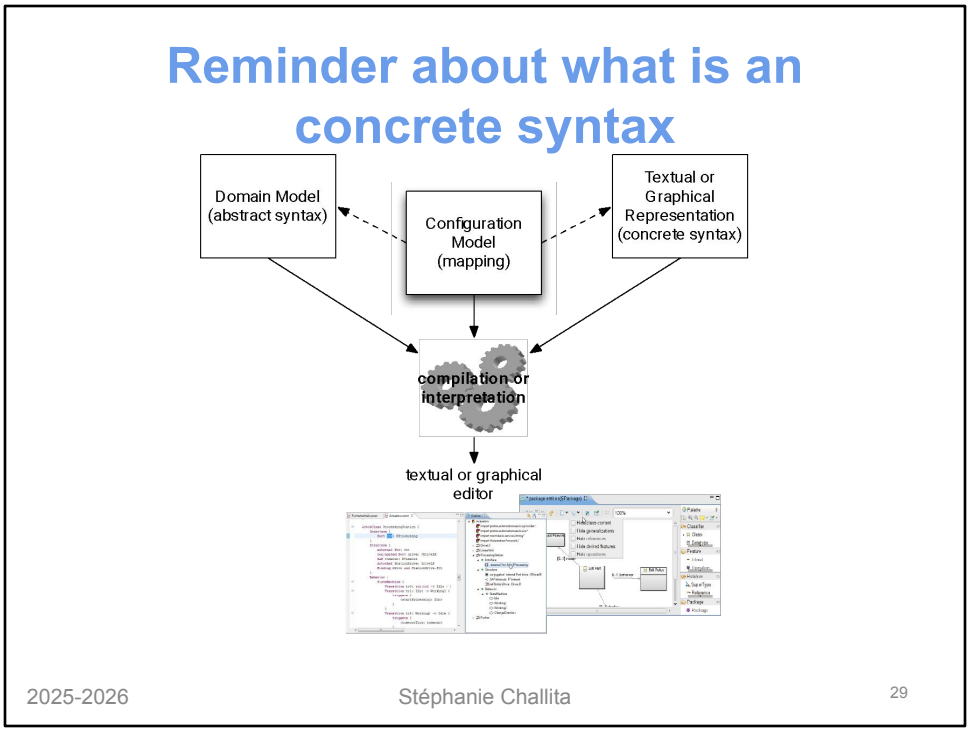
Ce graphe représente le monde logique du langage — ce que le parseur va construire à partir d'un code source.

C'est ce qu'on appelle un modèle conforme à un métamodèle.

À quoi ça sert ?

- À transformer ce modèle (génération de code, optimisation…),
- Ou encore à l'exécuter dans un moteur d'interprétation.

Et comme on l'a vu avec Xtext ou Langium :
La grammaire qu'on définit permet justement de générer automatiquement cette structure abstraite.
Ce que vous voyez là, c'est le cœur du traitement d'un DSL.



Maintenant qu'on a vu l'abstraction, l'abstract syntax, on s'intéresse à l'interface concrète, celle avec laquelle les utilisateurs interagissent : c'est la concrete syntax.

Qu'est-ce qu'on entend par concrete syntax ?

C'est la forme visible ou lisible d'un modèle :

- soit textuelle : comme un fichier `.q`, du code Xtend ou Langium,
- soit graphique : un diagramme UML, une boîte à flèches, etc.

Mais attention : cette forme concrète ne vit pas seule.

Elle est liée à un modèle abstrait (à gauche sur le schéma), qui définit la structure logique.

Au centre : un modèle de configuration ou de mapping.
C'est lui qui fait le lien entre :

- ce qu'on voit (la concrete syntax),
- et ce qu'on interprète ou compile (le domaine abstrait).

En bas, vous voyez deux exemples de DSLs dans Eclipse :

- à gauche, un éditeur textuel,
- à droite, un éditeur graphique.

Ces deux interfaces sont deux vues concrètes d'un même modèle abstrait.

Ce qu'il faut retenir ici :

Et c’est justement tout l’objectif d’outils comme Xtext ou Langium : automatiser cette connexion entre forme visible et structure modélisée.

Reminder about what is a semantic

- Any “meaning” given to the domain model
  - compiler, interpreter, analysis tool, refactoring tool, etc.
- Thanks to model transformations
  - program = data + algorithms ☺
- In practices?
  - It requires to “traverse” the domain model, and… do something!
  - Various languages, and underlying paradigms:
    - Declarative (rule-based): mostly for pattern matching (e.g., analysis, refactoring)
    - Imperative (visitor-based):
      - interpreter pattern: mostly for model interpretation (e.g., execution, simulation)
      - template: mostly for text generation (e.g., code/test/doc generators)

Maintenant qu’on a bien identifié la syntaxe concrète et la syntaxe abstraite, il reste une question centrale :
Mais que signifie ce qu’on modélise ? C’est ici qu’intervient la sémantique.

La sémantique, c’est tout simplement : Le sens ou le comportement associé à un modèle.

Cela peut être :

- un compilateur qui transforme le modèle en code,
- un interpréteur qui exécute le modèle,
- un analyseur qui vérifie des propriétés,
- un outil de refactoring qui restructure le modèle intelligemment.

Et comment donne-t-on ce sens ? Grâce à des transformations.
Un programme, finalement, c’est :

des données (le modèle) + des algorithmes (la sémantique associée)

En pratique, on traverse le modèle et on agit.

Deux grands paradigmes :

1. Déclaratif (par règles) : idéal pour faire de l’analyse, du pattern matching, de la transformation structurelle.
   - Ex : ATL, QVTo, ou outils de refactoring.
2. Impératif (visiteur) : on suit un interpreter pattern, souvent utilisé pour simuler, exécuter, ou générer du texte.

À retenir : La sémantique, c’est ce qu’on fait avec le modèle. Et pour cela, on a besoin de parcourir, transformer, exécuter.
La force des DSLs, c’est justement de relier une syntaxe à un comportement concret

Definition of the Behavioral Semantics of DSL

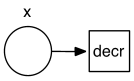
```
int x;
void decr () {
        if ( x>0 )
        x = x□1;
}
```

System
x : Int
decr()

- **Axiomatic**

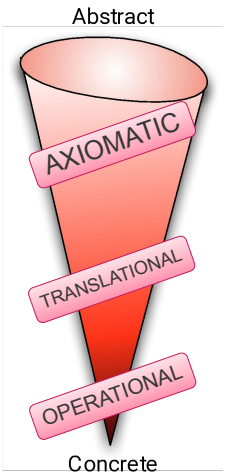
```
context System::decr() post :
    self .x =    if ( self .x@pre>0 )
                      then self.x@pre - 1
                      else self.x@pre
                 endif
```

- **Denotational/translational**



- **Operational**

```
operation decr () is do
    if x>0 then x = x - 1 end
```



Maintenant qu'on a bien compris ce qu'est la sémantique, la question suivante c'est : Comment la formalise-t-on ?
Autrement dit : comment donne-t-on un sens rigoureux à un langage ?

Il existe pour cela trois grandes approches, issues des fondements des langages de programmation :

Première approche : Axiomatic semantics (clic n°1)

Ici, on décrit des propriétés logiques qu'un programme doit satisfaire.

Par exemple :

> Si on exécute ce programme avec une variable x valant 3, alors à la fin, x vaut 7.

C'est l'approche des logiques de Hoare, très utilisée dans la vérification formelle. Elle ne décrit pas comment le programme s'exécute, mais ce qu'on attend comme relation entre états.

Deuxième approche : Denotational / Translational semantics (clic n°2)

Celle-ci repose sur le principe de traduction.

L'idée :

> Pour donner un sens à un programme dans un langage source, on le traduit dans un autre langage dont la sémantique est déjà bien définie.

C’est typiquement ce qu’on fait avec des compilateurs.

Troisième approche : Operational semantics (clic n°3)

Ici, on décrit comment le programme s’exécute étape par étape.

Par exemple :

> Une instruction `x := x + 1` signifie qu’on va lire `x`, ajouter 1, et réécrire dans `x`.

C’est l’approche la plus intuitive, souvent utilisée dans :

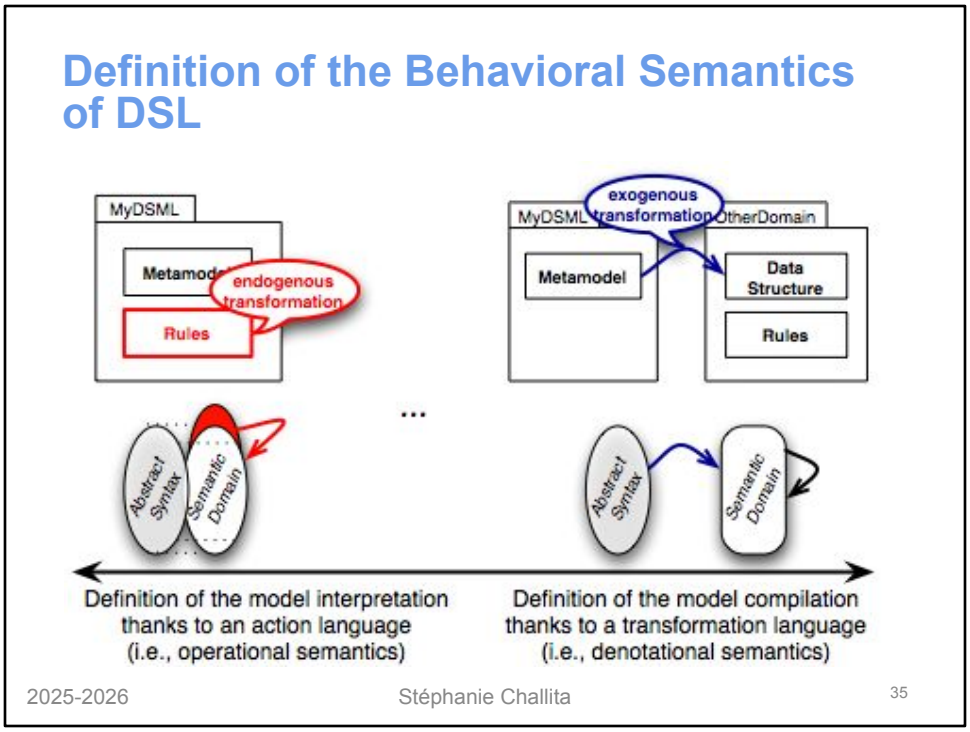
- les interpréteurs,
- les machines virtuelles,
- les langages exécutés directement à partir du modèle.

À retenir :

Ces trois approches sont complémentaires.
Elles peuvent être utilisées ensemble dans les DSLs :

- pour vérifier des propriétés,
- pour traduire vers une plateforme cible,
- pour exécuter ou simuler.

La clé, c’est de choisir celle qui correspond à vos objectifs : preuve, transformation, ou exécution.



Maintenant qu'on a vu les différentes approches pour définir la sémantique d'un langage, regardons comment on les applique concrètement à un DSL.

À gauche, on a un DSL. Ici, représenté par son métamodèle.
Et ce DSL porte ses propres règles : elles sont intégrées dans le langage lui-même.

C'est ce qu'on appelle une transformation endogène.
Autrement dit :

> Le comportement du langage est défini dans le même formalisme que celui qu'il manipule.

Typiquement : on ajoute une couche d'interprétation opérationnelle.
Par exemple, dans un DSL d'automate, une règle "exécute la transition si le signal est reçu".

À droite, on est dans une approche différente.
Le comportement n'est pas défini dans le DSL lui-même, mais via une transformation vers un autre domaine.

C'est une transformation exogène.
Autrement dit :

> On traduit les modèles de notre DSL vers une structure cible, dans un autre langage ou un autre formalisme.

C'est l'approche dénotationnelle : on compile vers un langage cible qui a déjà une sémantique bien établie.

Ces deux approches sont complémentaires :

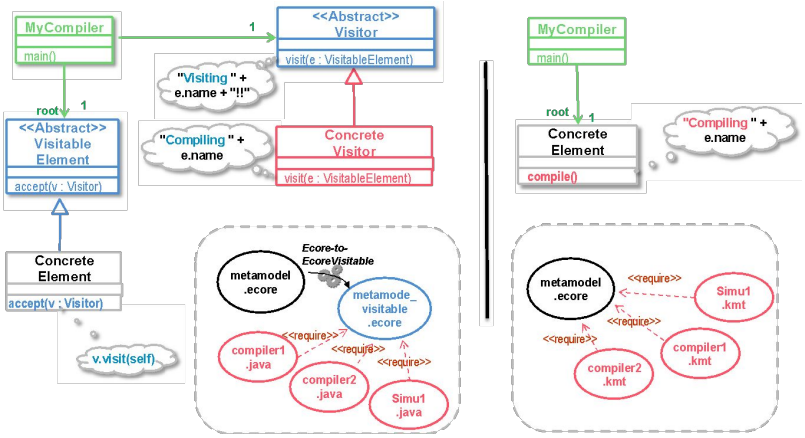
- Opérationnelle : on interprète directement le modèle,
- Dénotationnelle : on le transforme dans un formalisme cible.

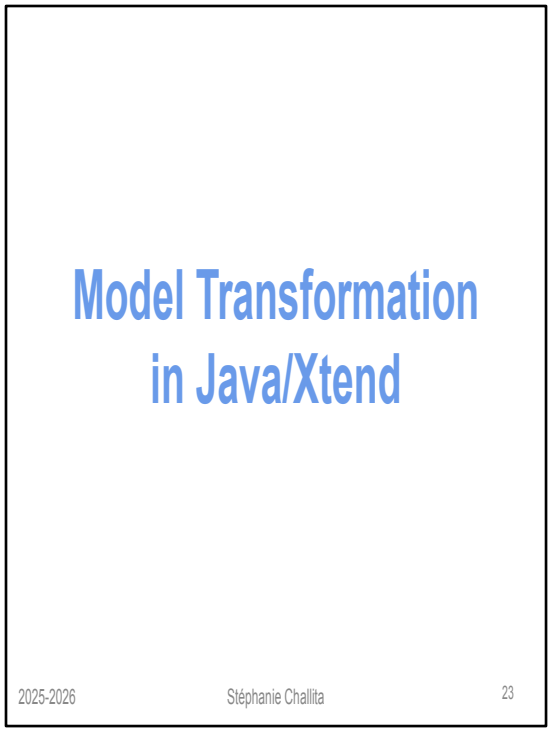
Le choix dépend de vos objectifs :

- Prototypage rapide, simulation ? → opérationnelle.
- Intégration dans une chaîne d'outillage, génération de code ? → dénotationnelle.

Implement your own compiler / interpreter

- Visitor-based?
  - Interpreter/visitor patterns, static introduction (aka. open class)





Maintenant que nous avons exploré les différents types de transformations de modèles, ainsi que leurs objectifs et classifications, passons à une partie plus concrète : la mise en œuvre des transformations de modèles dans un langage de programmation.

Cette section porte sur : Model Transformation in Java/Xtend

Nous allons voir comment réaliser des transformations de modèles de manière pragmatique, en s'appuyant sur la plateforme EMF et les langages comme Xtend ou Java.

L'idée est d'illustrer comment, une fois nos métamodèles et nos modèles définis, on peut écrire du code qui lit, transforme, et produit d'autres modèles ou du code – exactement comme dans les approches M2M ou M2T que nous avons vues.

Cette section met également en lumière le rôle de Xtend, un langage concis et puissant, conçu spécifiquement pour faciliter la manipulation de modèles, avec un bon support EMF et une syntaxe plus expressive que Java.

Effective Model Management

- How to load/serialize a model?
- How to visit, analyze and transform models?
- You can do it in Java (EMF API)

- We arbitrarily choose
  - interesting « features »
  - Integration within Eclipse ecosystem (incl. Xtext) and facilities to manage models
  - An example of a sophisticated language

Dans cette partie, on rentre dans le concret de la gestion de modèles :

Comment charger un modèle ?
Comment le sérialiser, le parcourir, l'analyser ou le transformer ?

Ce sont des opérations fondamentales dès lors qu'on manipule des artefacts modélisés.
Et la bonne nouvelle, c'est que tout cela est possible avec l'API EMF, et donc en Java, sans outil exotique.

Mais plutôt que de passer en revue toutes les fonctions possibles, on va se concentrer sur un sous-ensemble pertinent, c'est-à-dire des fonctionnalités concrètes et directement utiles dans un processus de transformation ou d'édition.

On fait aussi un choix délibéré ici :
On reste dans l'écosystème Eclipse, qui fournit tout un environnement cohérent, bien intégré, notamment avec Xtext.
Et on choisit d'illustrer cela avec Xtend, un langage qui permet de travailler avec EMF de manière plus expressive que Java, tout en restant interopérable avec lui.

L'idée : voir une mise en œuvre réelle, efficace et intégrée de la gestion de modèles.

Before going into details of Xtend…

- Recap of the scenarios
  - Text-to-Model
  - Model(s)-to-Model transformation
    - Interpreter/compiler, refactoring…
  - Metamodels as a « bridge » between techologies
  - Model-to-Text
    - Generators
- The solution of some of the « scenarios »
  - Just to give an overview of Xtend capabilities
  - To give a more practical/concrete view of some of the previous scenarios

Avant de plonger dans le code et les détails d'Xtend, prenons un instant pour revenir sur les grands scénarios que l'on a évoqués jusqu'ici.

Text-to-Model — c'est ce qu'on fait avec Xtext, ou avec tout langage textuel : on part d'un fichier source, on génère un modèle conforme à un métamodèle.

Model-to-Model transformation — un des cas classiques du MDE : on transforme un modèle source en un modèle cible, que ce soit dans le même métamodèle ou dans un autre.

Interpreter, compiler, refactoring… — ici, il s'agit d'exécuter ou d'analyser un modèle : on peut interpréter une machine à états, faire de la vérification, appliquer des refactorings...

Les métamodèles servent ici de pont entre les technologies : ce sont eux qui structurent les données, que vous soyez dans EMF, Xtext, Xtend, ou ailleurs.

Model-to-Text — c'est ce qu'on fait avec des générateurs : on prend un modèle et on génère du code, de la documentation, des scripts de configuration…

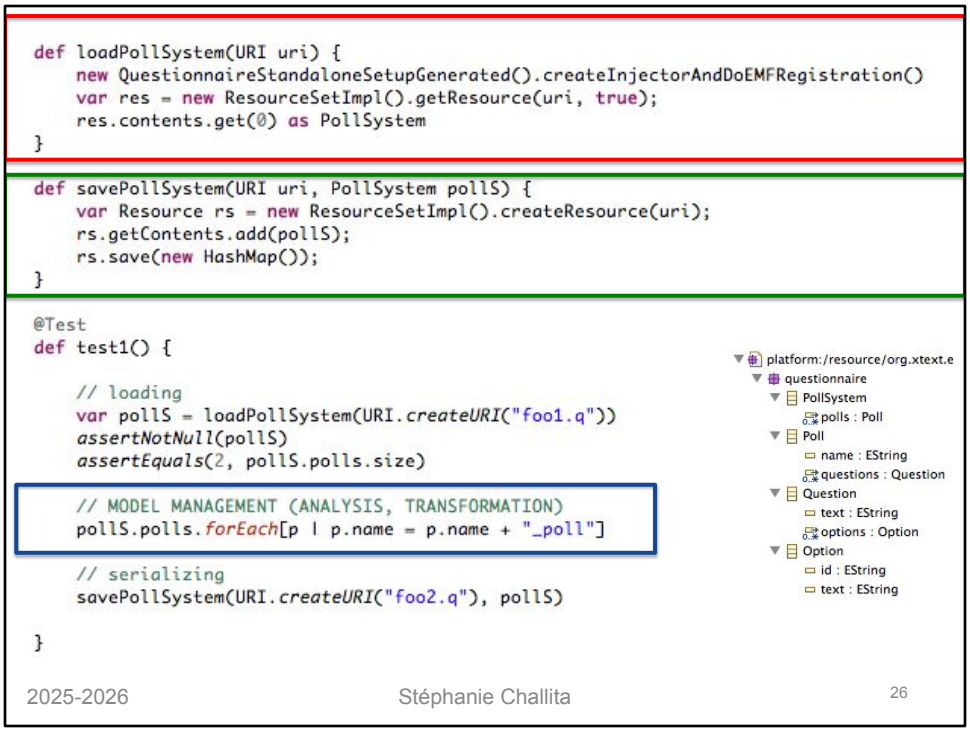
Ce qu'il faut comprendre maintenant, c'est que Xtend peut répondre à plusieurs de ces scénarios.

On va donc voir :

- quelques usages illustratifs,
- qui montrent les capacités de Xtend dans des situations concrètes.

Pas besoin de tout formaliser dès maintenant :

les détails syntaxiques et techniques.



Ce que vous voyez ici, c'est un exemple simple de gestion de modèle avec Xtend. On manipule un fichier `.q` basé sur notre langage PollSystem défini avec Xtext. À gauche, le code qui effectue cette transformation. À droite, la structure du modèle dans Eclipse.

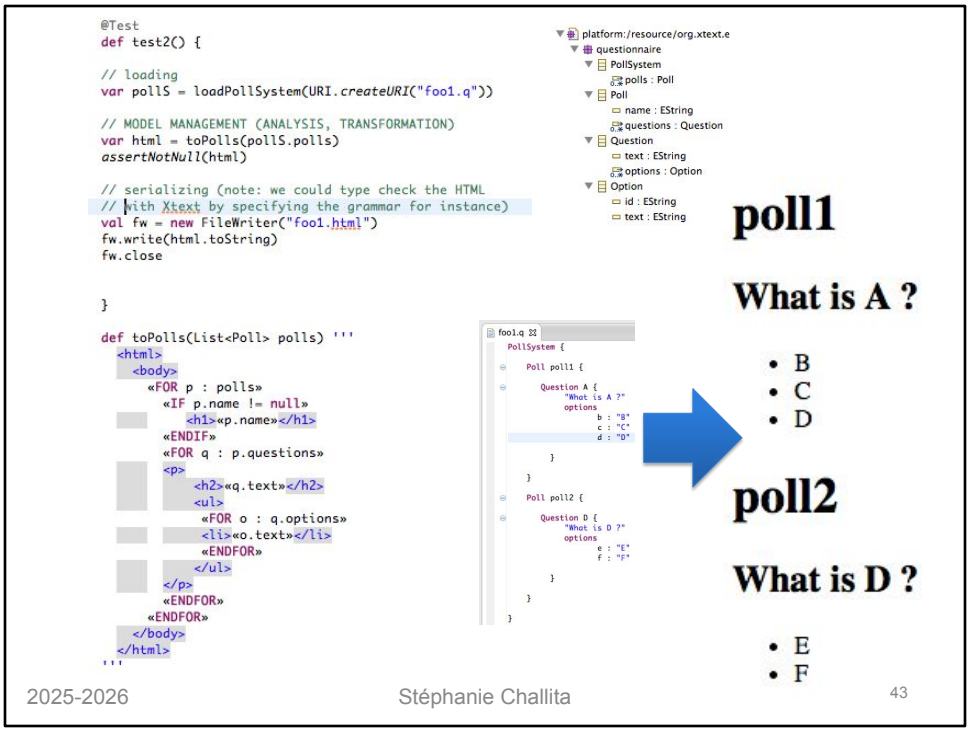
Rectangle rouge (fonction de chargement)
Commençons par la fonction de chargement :
On lit un fichier `.q` grâce à un `ResourceSet`, on en extrait l'objet racine — ici, un `PollSystem`. C'est l'opération de Text-to-Model, réalisée automatiquement via Xtext.

Rectangle vert (fonction de sauvegarde)
Ensuite, la fonction de sauvegarde :
On prend le `PollSystem`, on le réinjecte dans un `ResourceSet`, et on le sauvegarde. Cela correspond à l'opération inverse, Model-to-Text.

Rectangle bleu (test + transformation)
Ici, on voit un test unitaire classique, avec :

- Chargement du modèle
- Transformation légère du modèle : on ajoute `"_poll"` à tous les noms de sondages — c'est une transformation endogène horizontale
- Puis on resérialise le résultat dans un nouveau fichier `.q`.

C'est un exemple très concret de cycle complet de model management avec des outils comme Xtend :
chargement, modification, sauvegarde — tout cela avec un code concis et expressif.



Cette slide montre une autre forme de transformation, cette fois vers du HTML.
On part d'un fichier `.q` (à droite) qui représente un `PollSystem`, et on génère dynamiquement une page HTML lisible dans un navigateur.

Chargement & transformation (bloc en haut à gauche)
Comme tout à l'heure, on charge le modèle avec `loadPollSystem`.
Ensuite, on appelle `toPolls`, une fonction qui prend la liste de sondages (`List<Poll>`) et génère une chaîne HTML.
C'est ça, la transformation modèle-vers-texte.

La fonction `toPolls` (en bas à gauche)
C'est là que la magie opère.
Xtend permet d'écrire des templates très lisibles, où on imbrique directement du HTML avec des instructions Xtend :

- `FOR`, `IF`, `END`, etc.
- Chaque sondage devient une section HTML avec ses questions et options.
- C'est exactement ce qu'on appelle du templating.

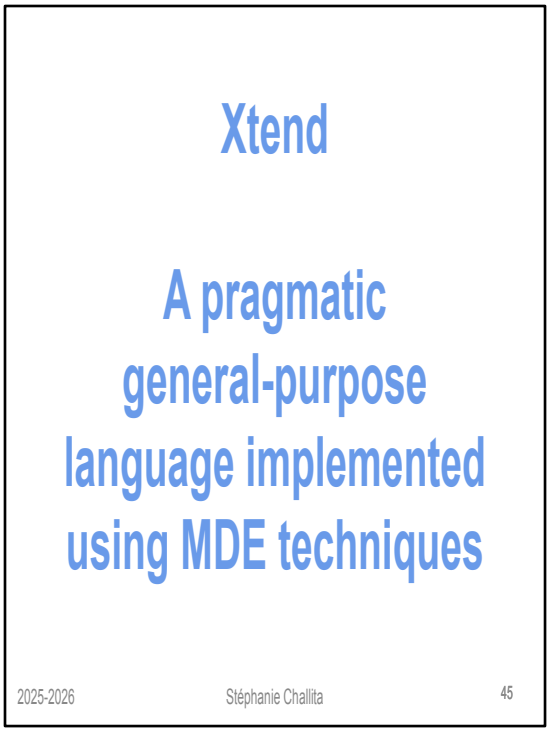
Note : il serait possible d'aller plus loin en typiant le HTML avec Xtext si on avait une grammaire HTML pour vérification.

Résultat (à droite)
Le HTML généré correspond directement à notre modèle `.q`.
Chaque `Poll` devient un `h1`, chaque `Question` un `h2`, et chaque `Option` une entrée de liste.

C’est simple, élégant, maintenable — et génératif.

À retenir
Ce genre de transformation est utile pour créer des vues lisibles sur les modèles :

- documentation automatique
- formulaires interactifs
- UI HTML pour des DSL textuelles.



Introduction à Xtend

On passe maintenant à une nouvelle section consacrée à Xtend, un langage que vous avez déjà entraperçu dans les slides précédentes à travers des exemples de transformations et de génération de code.

Xtend, c'est quoi ?

Xtend est un langage généraliste qui a été conçu pour être simple à écrire, agréable à lire, et surtout : profondément intégré à l'écosystème Java et EMF.

Il reprend les principes des langages modernes (inférence de types, lambdas, expressions concises), mais il est surtout un produit direct des techniques de MDE :

- Il est défini via une grammaire Xtext,
- Il repose sur un métamodèle EMF,
- Il utilise des transformations modèles pour générer du code Java.

Pourquoi on l'étudie ici ? Parce qu'Xtend est un cas d'étude parfait :

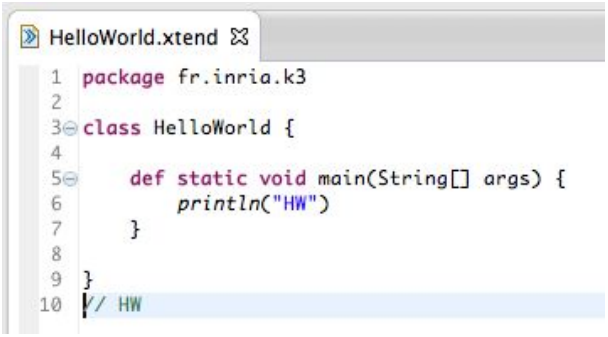
- Il illustre ce qu'on peut construire avec Xtext + EMF,
- Il montre comment DSL et langage généraliste peuvent se rencontrer,
- Et c'est un excellent langage pour manipuler des modèles, que ce soit pour des interprètes, des compilateurs, ou des générateurs.

Bref :

Xtend est à la fois un langage de programmation et un démonstrateur des technologies que vous venez d'apprendre.

On va donc s'y intéresser de plus près.

Hello World



Avant d'entrer dans les détails techniques, on commence comme toujours… par un Hello World.

Ce que vous voyez ici, c'est Eclipse avec un fichier `.xtend` ouvert.

Il s'agit d'un fichier Xtend classique, contenant une classe `HelloWorld` dans le package `fr.inria.k3`.

À l'intérieur, une méthode `main`, exactement comme en Java, mais écrite dans la syntaxe Xtend.

Quelques remarques :

- Le mot-clé `def` est utilisé à la place de `public static`, pour déclarer une méthode.
- L'appel à `println("HW")` est très naturel, sans besoin de préciser `System.out`, comme en Java.
- Xtend s'intègre directement dans Eclipse, avec auto-complétion, validation, navigation, etc.

Ce petit exemple montre déjà deux choses importantes :

1. Xtend ressemble à Java, mais avec une syntaxe plus légère.
2. Xtend est un langage compilé : ce fichier `.xtend` sera compilé automatiquement en un `.java` équivalent.

Ce Hello World va nous servir de point de départ pour explorer les fonctionnalités puissantes de Xtend :

- transformations de modèles,

- génération de code,
- et des mécanismes avancés comme les expressions de template ou les annotations actives.

Semi-colon is optional (within class, methods, etc.)

```
1  package fr.inria.k3.fields
2
3  class MyFielder {
4
5    int count = 1
6    static boolean debug = false
7    var name = 'Foo'           // type String is inferred
8    val UNIVERSAL_ANSWER = 42 // final field with inferred type int
9    // ...
10   public int count2 = 2 ;
11
12 }
13
```


On va faire un tour d'horizon de la syntaxe de xtext.
Tout d'abord, vous verrez qu'il y a pas mal de sucre dans la syntaxe the xtext, a commencé par le fait que les points-virgules sont optionnels.

Package Declaration

HelloWorld.xtend

```
1  package fr.inria.k3
2
3  class HelloWorld {
4
5      def static void main(String[] args) {
6          println("HW")
7      }
8
9  }
10 // HW
```

Semi-colon ‘;’ is optional

‘^’ for avoiding keyword conflicts

HelloWorld.xtend | HelloWorld.java | PackageExploder.xtend

```
1  package fr.inria.k3.^def
2
3  class PackageExploder {
4
5  }
```


Comme pour java, xtent est organisé en package, définit tout en haut d’un fichier.
Il existe un moyen d’échappé les mot-clés si on veut les utiliser, ici comme pour “def”

Methods

```
1  package fr.inria.k3.methods
2
3  class FooMethod {
4
5      def foo1() {
6          "A"
7      }
8
9      def foo2() {
10         6 + 3
11     }
12
13     def private foo3() {
14         6 + 3
15     }
16
17     def public foo4() {
18         foo3() * 8
19     }
20 }
21
22 class FooMethodUses {
23
24     def fooUse() {
25         new FooMethod().foo1
26         new FooMethod().foo3()
27
28
29
30         new FooMethod().f
31
32
33     }
34
35 }
```

foo1 : String – FooMethod.foo1()
foo2 : int – FooMethod.foo2()
foo4 : int – FooMethod.foo4()

By default: visibility conditions set to public

HelloWorld.xtend

```
1  package fr.inria.k3
2
3  class HelloWorld {
4
5      def static void main(String[] args) {
6          println("HW")
7      }
8
9  }
10 // HW
```


On commence par une structure fondamentale : la déclaration de méthodes.
Xtend réutilise la grammaire de Java mais la simplifie considérablement. On introduit une méthode avec le mot-clé `def`, suivi :

- du nom de la méthode,
- de la liste des paramètres typés entre parenthèses,
- éventuellement du type de retour
- et d'un corps entre accolades.

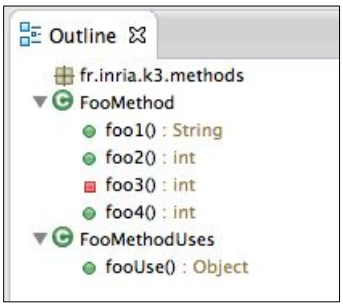
Le mot-clé `return` est souvent inutile : Xtend retourne automatiquement la dernière expression.
Les méthodes peuvent être déclarées dans des classes Xtend, ou directement dans des fichiers utilitaires.

Methods

```
1  package fr.inria.k3.methods
2
3  class FooMethod {
4
5      def foo1() {
6          "A"
7      }
8
9      def foo2() {
10         6 + 3
11     }
12
13     def private foo3() {
14         6 + 3
15     }
16
17     def public foo4() {
18         foo3() * 8
19     }
20 }
21
22 class FooMethodUses {
23
24     def fooUse() {
25         new FooMethod().foo1
26         new FooMethod().foo3()
27
28
29
30         new FooMethod().f
31
32
33     }
34
35 }
```

foo1 : String – FooMethod.foo1()
foo2 : int – FooMethod.foo2()
foo4 : int – FooMethod.foo4()

Type inference (return type)

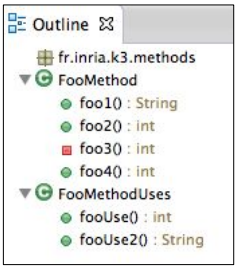


Le typage est obligatoire sur les paramètres, mais facultatif sur le retour si inféré.

Method Calling

You can omit parentheses

```
33    def fooUse2() {
34        3.toString
35        "4".length
36        new FooMethod().foo1
37        new FooMethod().foo1()
38        new FooMethod().foo1 + 3.toString
39
40    }
```

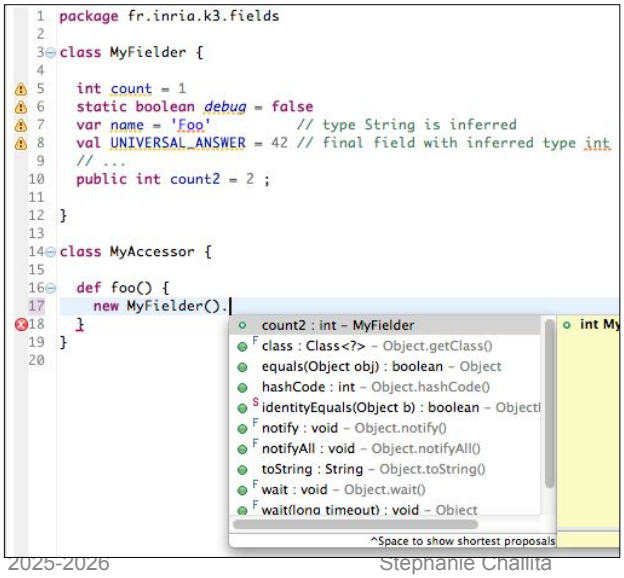


En Xtend, appeler une méthode fonctionne exactement comme en Java : on écrit simplement `objet.methode(paramètres)`.

La différence, c'est que Xtend ajoute :

- des raccourcis syntaxiques,
- des expressions plus concises,
- et une prise en charge élégante des fonctions anonymes et lambda.

Fields



By default: visibility conditions set to private

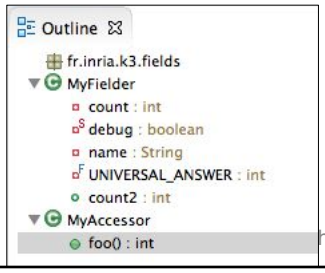
Xtend permet de déclarer des champs d'instance de manière très simple, tout en étant interopérable avec Java.
Le point important, c'est que Xtend génère automatiquement le code Java associé :

- les champs sont privés,
- les getters et setters sont générés selon les conventions JavaBeans,
- vous pouvez y accéder directement via `objet.champ` : c'est traduit correctement en Java.

Xtend infère le type si on l'omet, tant que l'initialisation est présente.

Fields

```
1  package fr.inria.k3.fields
2
3  class MyFielder {
4
5    int count = 1
6    static boolean debug = false
7    var name = 'Foo'            // type String is inferred
8    val UNIVERSAL_ANSWER = 42 // final field with inferred type int
9    // ...
10   public int count2 = 2 ;
11
12 }
13
```



primitive types of Java (int, boolean, etc) with autoboxing

var: type inference

val: constant, « final » in Java

On utilise :

- `val` pour un champ immuable — on ne pourra pas le modifier après initialisation.
- `var` pour un champ modifiable.

Static Methods (::)

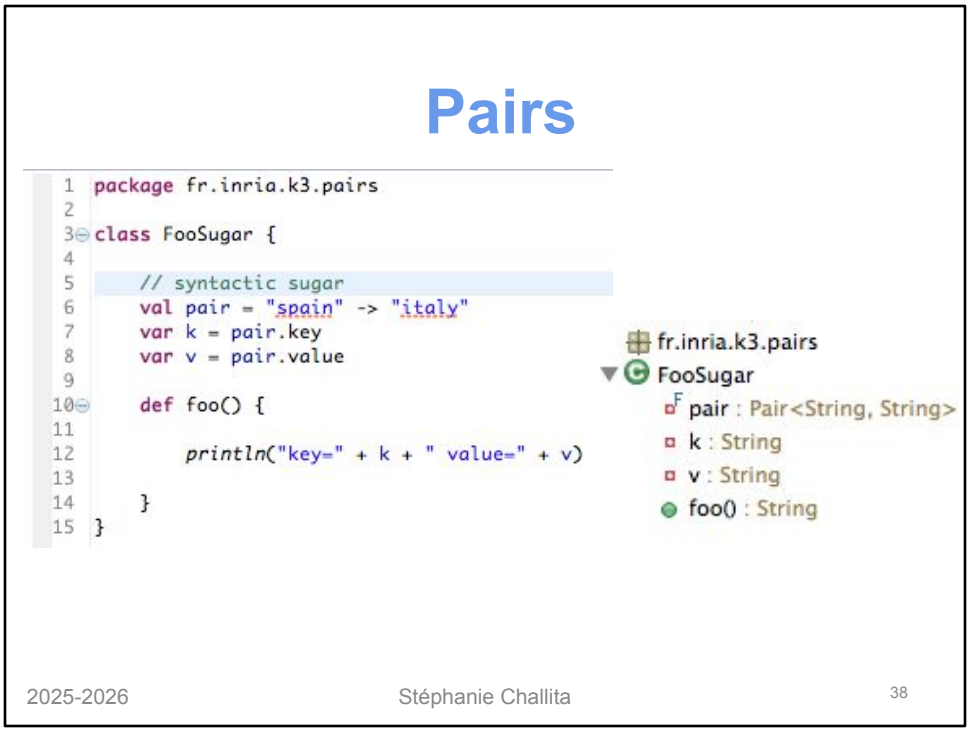
```xtend
package fr.inria.k3.stat

import java.util.Collections

class FooStati {


    static var colors = newArrayList(46, 76, 89, 53)


    def static void main(String... args) {
        println("B " + colors)
        Collections::sort(colors)
        println("A " + colors)

        colors.add(45)
        println("A " + colors)

    }
}
```

```
B [46, 76, 89, 53]
A [46, 53, 76, 89]
A [46, 53, 76, 89, 45]
```


Xtend introduit une syntaxe très pratique pour travailler avec les méthodes statiques, grâce à l'opérateur `::`.
Avec `NomDeClasse::nomDeMéthode`, on peut :

- appeler directement une méthode statique,
- ou référencer cette méthode comme une fonction, par exemple pour la passer à une lambda.

Ici, `Collections::sort(colors)` est une référence de méthode utilisée comme callback dans une boucle.
Cette syntaxe permet une meilleure intégration avec les API fonctionnelles de Java, tout en rendant les appels plus concis et lisibles.



Xtend propose un opérateur très simple pour créer des paires clé-valeur, c'est l'opérateur `->`.

Par exemple :

```
val pair = "spain" -> “italy”
```

Ici, on construit un objet de type Pair<String, String>.

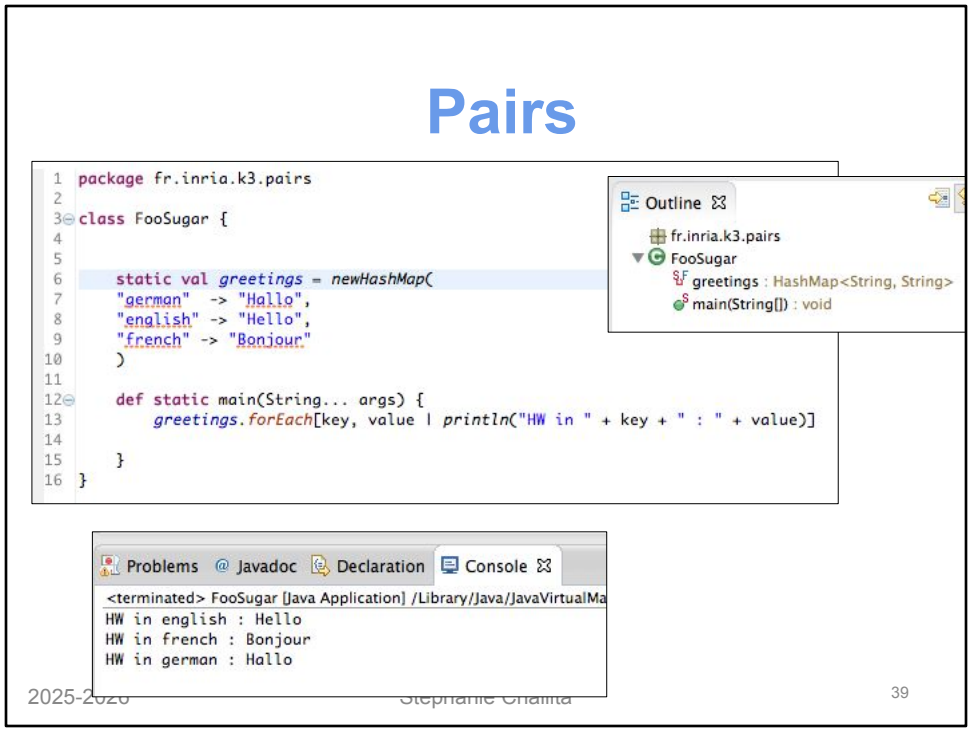
Ces paires sont particulièrement utiles :

- pour construire des structures associatives,
- pour mapper des correspondances entre modèles,
- ou comme valeurs de retour intermédiaires dans des transformations.

Une fois la paire créée, on peut accéder à ses éléments avec .key et .value, comme dans :

```
println(“key=” + k + “ value=” + v)
```

Ce mécanisme est très utilisé avec `map`, `groupBy`, ou `associate`, et permet de structurer proprement des données simples, sans devoir créer une classe dédiée à chaque fois.



Ici, on voit un exemple complet et concret d'utilisation des `Pair` avec `forEach`.

On déclare une variable statique appelée `greetings`, qui contient une liste de paires langue → salutation.

On utilise ensuite `.forEach` pour parcourir cette liste, et on affiche chaque salutation avec sa langue associée.

Ce qu'on voit ici :

- L'opérateur `->` permet de construire les paires très facilement.
- `.forEach` accepte une lambda implicite, ici nommée `g`, qui représente chaque `Pair<String, String>`.
- On peut accéder aux champs `.key` et `.value` pour manipuler le contenu.

Ce genre de structure est très pratique en Xtend pour stocker des correspondances temporaires, configurer des générateurs de code, ou structurer de petits DSLs.

Immutable data structure

```
1  package fr.inria.k3.stat
2
3  import java.util.Collections
4
5  class FooStati {
6
7
8      static var colors = #[46, 76, 89, 53] // newArrayList(46, 76, 89, 53)
9
10
11     def static void main(String... args) {
12         println("B " + colors)
13         Collections::sort(colors)
14         println("A " + colors)
15
16         colors.add(45)
17         println("A " + colors)
18
19     }
20 }
```

```
<terminated> FooStati [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_13.jdk/Contents/
B [46, 76, 89, 53]
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.Collections$UnmodifiableList$1.set(Collections.java:1244)
        at java.util.Collections.sort(Collections.java:159)
        at fr.inria.k3.stat.FooStati.main(FooStati.java:15)
```


Xtend encourage l'utilisation de structures immuables, par défaut.
Dans cet exemple, on définit une variable statique `colors`, une liste de nombres entiers, avec la syntaxe `#[]`.

Cette syntaxe crée une liste immuable :
on ne pourra pas la modifier après sa création.

Dans `main`, on affiche la liste avec `println(colors)`,
puis on appelle `colors.sort` — mais attention :
Cela retourne une nouvelle liste triée,
Sans modifier la liste originale.

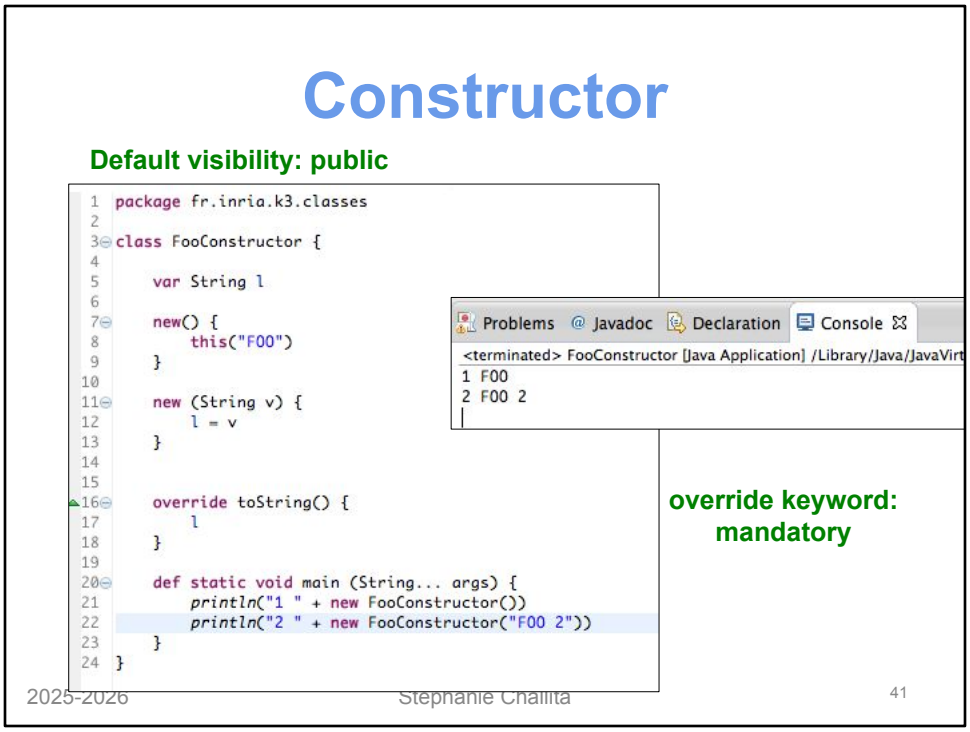
On peut le constater car le deuxième `println(colors)` affiche toujours la version non triée.

Si on essaie d'appeler `colors.add(100)`, on obtient une erreur de compilation :

> Cannot invoke add on List<Integer> because it is read-only.

Cette approche permet de garantir :

- que les données ne changent pas de manière inattendue,
- que le code reste simple et prévisible,
- et que les structures sont plus sûres (notamment en multithreading).



Xtend reprend la structure de Java, mais avec une syntaxe beaucoup plus simple. Pour définir un constructeur, on utilise le mot-clé `new`, suivi des paramètres et du bloc d'initialisation.

Dans l'exemple :

- La classe `FooConstructor` a deux constructeurs: sans paramètre et avec 1 parametre
- Le constructeur affecte les valeurs reçues aux champs à l'aide de `this.nomChamp`.

Points clés à retenir :

- Visibilité par défaut : publique. On ne met donc pas `public` devant le constructeur ou la classe.
- Le mot-clé `override` est obligatoire lorsqu'on redéfinit une méthode héritée comme `toString`.

Lorsqu'on appelle `println(foo)`, Xtend utilise automatiquement `toString()`, qu'on a redéfini ici de façon plus expressive.

Cast and Type

```xtend
package fr.inria.k3.types

class FooTypes {

    static val Object obj = "a string"
    // static val String s = obj
    static val String s = obj as String // cast

    def static void main(String... args) {
        println(typeof(String) + "") // String.class
        println("\t" + s + "\n")

    }
}
```


Ici, on illustre deux fonctionnalités typiques autour des types en Xtend : le cast explicite et l'introspection de type.
On part d'un objet `obj` de type `Object`, initialisé avec une chaîne.
Ensuite, on effectue un cast explicite avec `as String`.
C'est l'équivalent de `(String)obj` en Java, mais avec une syntaxe plus lisible.
Dans `main`, on utilise `typeof(String)` pour obtenir une représentation du type `String`.
Cela permet d'interroger ou de générer dynamiquement du code en fonction de types.
Xtend reste fortement typé, mais autorise ce type de manipulation lorsqu'on a besoin de flexibilité, notamment pour :

- interagir avec du code Java externe,
- manipuler des modèles hétérogènes (ex : EMF),
- ou écrire des générateurs plus dynamiques.

Extension Methods…

« … allow to add new methods to existing types without modifying them. »

```
def removeVowels (String s){
  s.replaceAll("[aeiouAEIOU]", "")
}
```

We can call this method either like in Java:

```
removeVowels("Hello")
```

or as an extension method of String:

```
"Hello".removeVowels
```

The first parameter of a method can either be passed in after opening the parentheses or before the method call

Xtend propose une fonctionnalité très puissante et pratique : les extension methods.
Le principe est simple :
vous définissez une méthode "classique", comme la méthode “removeVowels”.
Mais ensuite, vous pouvez l’appeler comme si elle appartenait à `String` directement
C’est ce qu’on appelle "promouvoir" le premier paramètre en récepteur :
la chaîne `"Hello"` devient l’objet sur lequel on appelle `.removeVowels`.
C’est extrêmement utile pour :

- enrichir les types standards (comme String, List, Map),
- écrire du code fluide, lisible, proche du langage naturel,
- éviter d’avoir des classes utilitaires pleines de méthodes statiques.

Et bien sûr, tout ça est fortement typé, compilé, et compatible avec Java.

Lambda Expression

No need to specify the type for e

```
1.  textField.addActionListener([ e |
2.    textField.text = "Something happened!"
3.  ])
```

You can even omit e

```
1.  textField.addActionListener([
2.    textField.text = "Something happened!"
3.  ])
```


Xtend propose une syntaxe très fluide et concise pour écrire des lambdas — c'est-à-dire des fonctions anonymes.
Dans l'exemple :
La variable `e` représente l'événement reçu, mais on n'a pas besoin de déclarer son type. Xtend l'infère automatiquement à partir du contexte (`ActionListener` → `ActionEvent`).
Encore mieux :
si vous n'utilisez même pas `e` dans le corps, vous pouvez l'omettre complètement :
Cela rend le code beaucoup plus expressif et lisible, notamment pour :

- les callbacks (`onClick`, `onChange`, etc.),
- les opérations sur collections (`map`, `filter`, `forEach`),
- ou tout usage fonctionnel dans les API Java modernes.

En résumé :
Xtend vous permet d'écrire du code fonctionnel propre, concis, et interopérable avec Java — sans sacrifier la clarté.

Lambda Expression

```
1.  Collections.sort(someStrings) [ a, b |
2.    a.length - b.length
3.  ]
```

Java 8:
```
shapes.forEach(s -> { s.setColor(RED); });
```

Xtend:
```
shapes.forEach[color = RED]
```

Java 8:
```
shapes.stream()
      .filter(s -> s.getColor() == BLUE)
      .forEach(s -> { s.setColor(RED); });
```

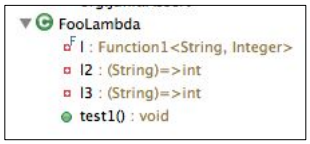
Xtend:
```
shapes.stream
      .filter[color == BLUE]
      .forEach[color = RED]
```


Ici, on voit à quel point Xtend excelle dans l'écriture fonctionnelle, avec une syntaxe plus concise et lisible que Java 8, tout en restant compatible.
Premier exemple : tri personnalisé avec `sort`.
En Java, cela prendrait plusieurs lignes avec un `Comparator`.
En Xtend : La lambda remplace directement l'objet fonctionnel attendu. Ensuite, comparons deux syntaxes `forEach` :
Ensuite, comparons deux syntaxes `forEach` :
En Java, on passe une lambda complète avec `s -> ...`.
En Xtend, c'est : Le paramètre `s` est implicite, et la syntaxe est ultra-lisible.
Même chose pour une chaîne d'opérations fonctionnelles :
Ici, chaque opération est expressive, sans répétition, sans verbosité.
En résumé :

- Xtend rend la programmation fonctionnelle fluide,
- Et permet d'exprimer des transformations complexes avec très peu de code.

Lambda Expression

```
class FooLambda {

    val l = [String s | s.length]
    var (String)=>int l2 = [it.length] // it is a keyword for refering to the first parameter
    var (String)=>int l3 = [length] // we can even ommit it or the first parameter

    @Test
    def test1() {
        assertEquals(l.apply ("RRRR"), l2.apply("PPPP"))
        assertEquals(l2.apply ("RRRR"), l3.apply("PPPP"))
    }
}
```



Ici, on résume de manière très claire **la souplesse de Xtend pour définir des lambdas**, avec trois styles équivalents.
Ligne 1 : syntaxe Java-like, très explicite. On précise le type et on donne un nom de paramètre :
Ligne 2 : plus idiomatique en Xtend. On utilise `it`, le paramètre implicite
Ligne 3 : encore plus concis ! On **omet complètement le paramètre**, quand c'est possible :
Dans tous les cas, ces trois lambdas sont **de type `Function1<String, Integer>`**, comme on le voit dans la vue de droite.
Le test suivant vérifie que les trois notations sont **équivalentes en comportement** :
À retenir :

- Xtend **autorise plusieurs niveaux de concision**
- On peut écrire des lambdas lisibles, explicites, ou extrêmement compactes
- Très utile pour la **programmation fonctionnelle** dans des contextes Java-like

Templates

```
1  package fr.inria.k3.templates
2
3  import org.junit.Test
4  import static org.junit.Assert.*
5
6  class FooTempl {
7
8
9      def someHTML(String content) '''<html><body>«content»</body></html>'''
10
11
12     @Test
13     def test1() {
14         assertEquals("<html><body>HW</body></html>", someHTML('HW').toString)
15     }
16
17 }
```

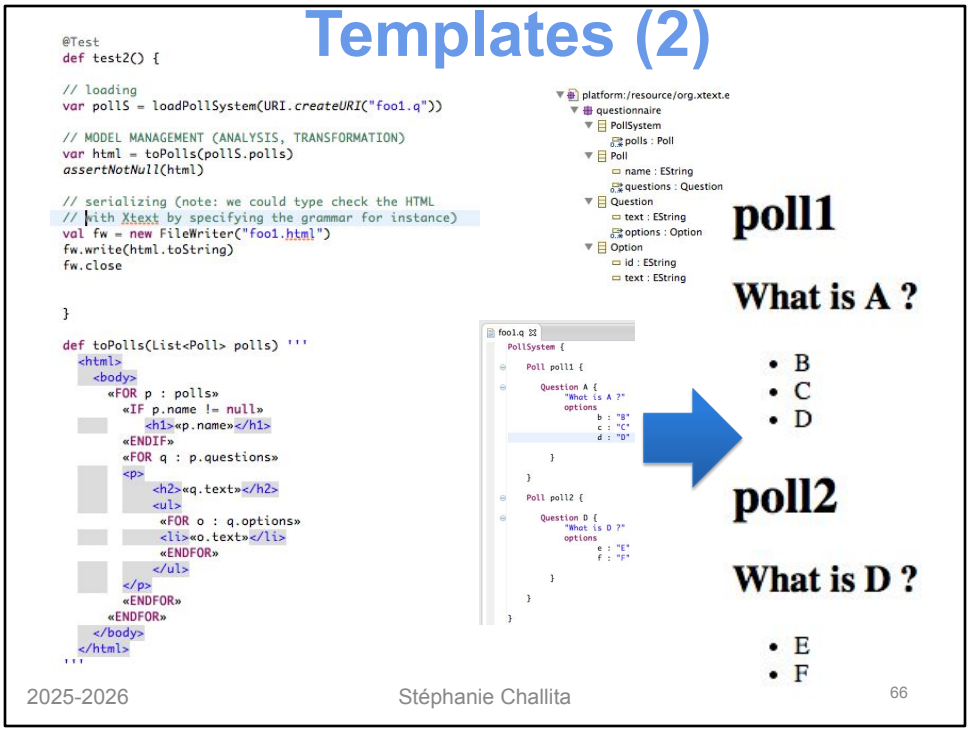

Xtend propose un mécanisme très puissant et intuitif : les template expressions
Dans cet exemple :

On utilise des triple quotes ''' pour délimiter une chaîne multilignes (comme en Python).
À l'intérieur, on peut directement insérer des variables ou expressions Xtend via les guillemets doubles "...".

Ici, on insère la variable content dynamiquement dans une structure HTML.

Le test suivant montre comment on peut générer dynamiquement du HTML.
Cela permet de faire du Model-to-Text très simplement, sans outil supplémentaire.
À retenir :

- Le moteur de template de Xtend est intégré au langage
- Il est type-safe : si on insère une variable mal typée, la compilation échoue
- C'est extrêmement pratique pour générer des fichiers (HTML, Java, SQL, etc.)



Maintenant que vous avez vu la syntaxe de base des templates en Xtend, regardons un exemple plus complet et réaliste de Model-to-Text transformation, appliqué à notre système de sondage.

En haut à gauche, on retrouve notre fonction de test `test2()`.

Elle commence par charger un modèle, comme on l'a déjà vu avec `loadPollSystem`.

Ensuite, on transforme ce modèle en HTML avec la fonction `toPolls`, et on vérifie que le résultat n'est pas nul.

Puis, ce HTML est sérialisé dans un fichier avec un `FileWriter`.

Notez le commentaire important : ici, on écrit une chaîne de caractères HTML à la main, mais on pourrait aller plus loin.

Par exemple, on pourrait définir une grammaire HTML dans Xtext, ce qui nous permettrait de vérifier la validité de notre HTML à la compilation, comme on le ferait avec un modèle.

Dans la fonction `toPolls`, plus bas, vous voyez un exemple typique de template Xtend :

- On utilise des balises HTML standards,
- On intègre des blocs dynamiques grâce aux instructions `FOR`, `IF`, `END FOR`, etc.
- Et on injecte les données du modèle (le nom du poll, les questions, les options) directement dans le HTML.

Sur la droite, vous voyez :

- Le fichier `.q` d'entrée (le modèle),
- Et à droite, le résultat HTML rendu, avec une flèche qui illustre bien la transformation.

Ce slide résume très bien comment, en partant d'un modèle métier (ici, un système de sondage), on peut générer automatiquement un artefact exécutable ou visualisable, ici du HTML pour une interface web.
C'est un exemple concret de modèle opérationnalisé, avec un langage DSL spécifique et un générateur simple, maintenable, et entièrement traçable.

Templates (3)

- You already experiment with web templating engines (JSP, Scala templates in Play!, Symfony templates, etc.)

```
<h1>Exemple de page JSP</h1>
<%-- Impression de variables --%>
<p>Au moment de l'exécution de ce script, nous sommes le <%= date %>.</p>
<p>Cette page a été affichée <%= nombreVisites %> fois !</p>
</body>
</html>
```

- Alternatives exist in the modeling world
  - Multiple predefined and customizables generators

- Xtend: seamless integration into a general purpose language

Ce slide vient relier le monde des DSLs et de la génération de code à des choses que vous avez probablement déjà expérimentées dans d'autres contextes.
Vous avez peut-être déjà utilisé des moteurs de templates pour le Web : comme JSP, les templates Scala dans Play!, ou encore Twig avec Symfony.
Le principe est toujours le même : on écrit du HTML dans lequel on injecte dynamiquement des données issues du code, souvent à partir d'un modèle ou d'un contrôleur.
Eh bien dans le monde de la modélisation et des DSLs, on a des approches similaires.
Il existe plusieurs outils de génération de code qui s'appuient sur des templates :

- Certains sont préconfigurés, mais on peut généralement les personnaliser.
- L'un des plus connus dans l'écosystème Eclipse, c'est Acceleo (vous voyez son logo ici), qui permet de faire de la génération de code conforme au standard OMG MOFM2T.

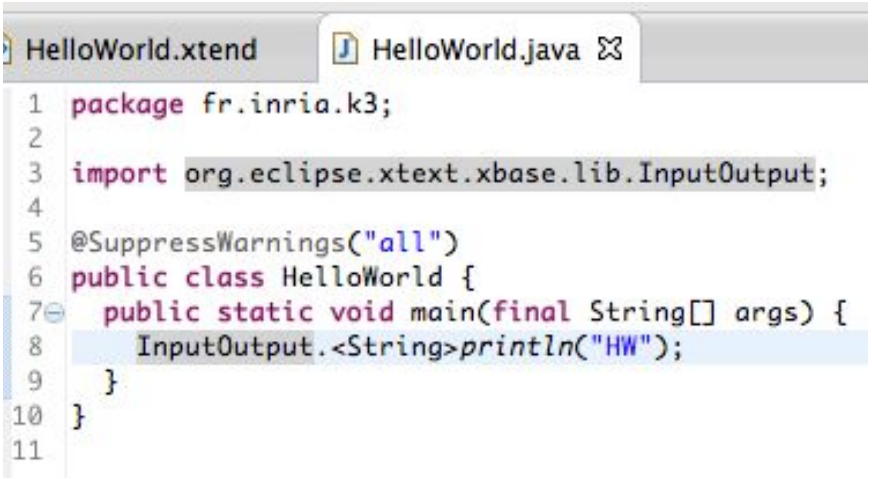
Et enfin, comme vous l'avez déjà vu, Xtend offre une alternative très élégante et moderne :

- Vous pouvez écrire des templates directement dans le langage généraliste,
- Avec une intégration transparente à la logique métier,
- Et sans rupture entre le code, le modèle, et les artefacts générés.

En résumé, que vous veniez du monde du Web ou de l'ingénierie dirigée par les modèles, les templates sont une pièce centrale de la génération automatique, et vous

avez des outils puissants pour les manipuler.

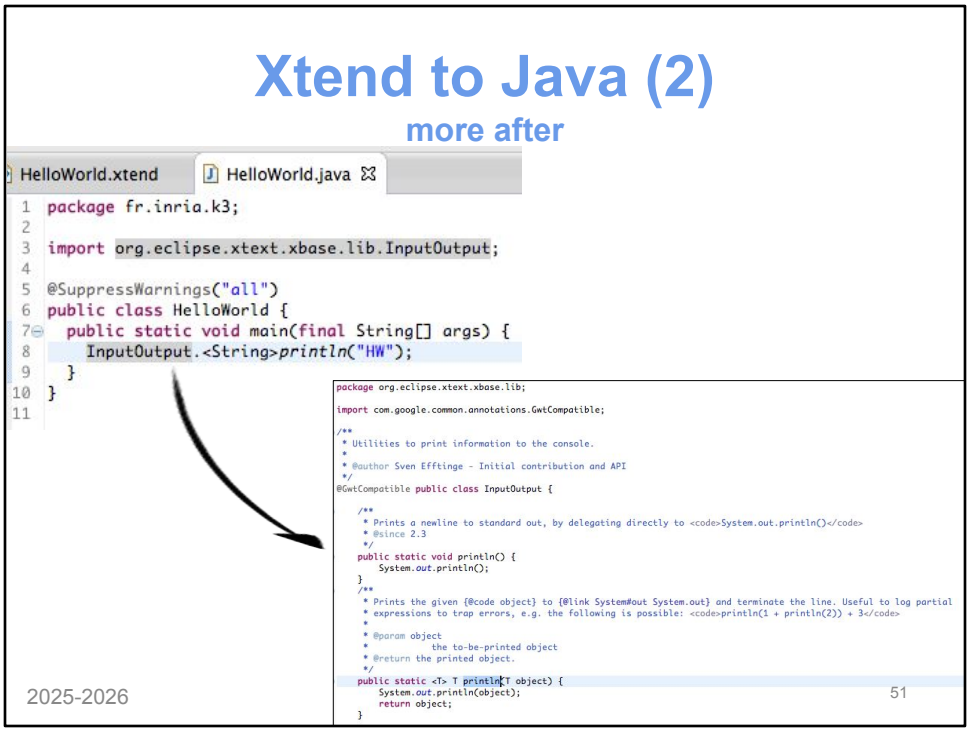
Xtend to Java



Ce slide illustre un point clé de la philosophie Xtend : Xtend n'est pas un langage indépendant, c'est un langage qui compile vers Java.
À gauche, vous voyez un fichier `HelloWorld.xtend`. C'est le code tel que vous, développeur, l'écrivez.
C'est plus concis, plus lisible, plus moderne que du Java classique.
Et à droite, vous voyez ce que le compilateur Xtend génère : un fichier `.java` entièrement standard.
Ici par exemple, l'appel à `println("HW")` est transformé en un appel explicite via `InputOutput.println(...)`, avec les bons types Java.
Ce qu'il faut bien comprendre ici, c'est que :

- Vous bénéficiez de toute l'outillage Java existant : compilation, débogage, exécution, intégration à Maven/Gradle, etc.
- Mais avec une syntaxe plus expressive.
- Et surtout, cela s'intègre très bien avec les outils de modélisation, comme vous l'avez vu avec Xtext.

Autrement dit : Xtend vous donne la puissance de Java, avec la légèreté d'un langage moderne.



Logiquement, on voit ici ce que fait réellement Xtend sous le capot.
On part de notre code très simple en Xtend — une instruction `println("HW")` dans une classe HelloWorld.
Et ce que produit Xtend à l'exécution, c'est du Java parfaitement valide, mais en s'appuyant sur une bibliothèque utilitaire :
`org.eclipse.xtext.xbase.lib.InputOutput`.
Ici, `InputOutput.println(...)` est une méthode statique générique.
Elle redirige simplement vers `System.out.println(...)`, tout en retournant l'objet passé en paramètre.
Cela permet des appels chaînés, ou des utilisations dans des expressions plus complexes.
Ici, on illustre bien un point important :

Xtend repose sur une librairie d'exécution Java bien définie.

Et surtout :

- Le code est interopérable avec le Java standard,
- Le comportement est déterministe et maîtrisé,
- Et la lisibilité reste maximale côté Xtend.

On commence donc à comprendre le rôle de Xtend comme langage génératif, capable de produire du Java robuste, avec une syntaxe bien plus agréable à écrire.