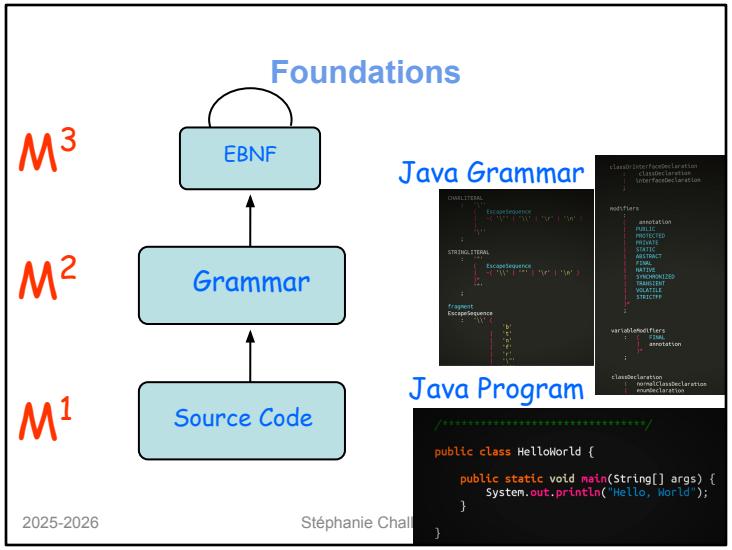
Pour cela, nous allons utiliser Xtext, un framework puissant pour créer des DSLs textuelles.

- Xtext repose sur EMF, mais propose une interface plus conviviale pour définir des grammaires textuelles
- En quelques minutes, on peut :
  - Définir une grammaire
  - Générer un éditeur syntaxique
  - Bénéficier d’un environnement complet (autocomplétion, validation, sérialisation…)

C’est donc un outil central dans notre approche.



Ce schéma montre trois niveaux d'abstraction, qui reprennent l'architecture MDA (M0 → M3),
mais appliqués au domaine des langages de programmation textuels.
En bas, au niveau M1, on a le code source
C'est l'exemple concret — un fichier `.java`, `.sql`, `.c`, ou tout autre langage.
Ce code est ce que l'utilisateur écrit réellement.
Au-dessus, niveau M2, on a la grammaire du langage
C'est le métamodèle syntaxique :
il décrit les règles de structure du langage: les mots-clés, la hiérarchie, les blocs, les expressions, etc.
Par exemple : ce qu'est une classe, une méthode, un identifiant…
Tout en haut, au niveau M3, on trouve EBNF
L'EBNF (Extended Backus–Naur Form) est un métalangage utilisé pour écrire… des grammaires.
C'est une grammaire de grammaires :
on s'en sert pour formaliser ce qu'est une règle syntaxique, une alternance, une répétition, etc.
On voit alors que:

- Le code source est une instance de la grammaire
- Et la grammaire elle-même est une instance d'EBNF

Cela reflète exactement ce qu'on a vu dans la modélisation avec MOF/Ecore :

Le modèle est instance du métamodèle,

le métamodèle est instance d'un métamétamodèle.

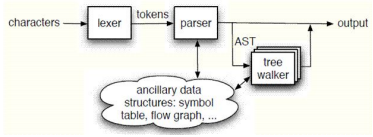
Fondamentalement, pour les langages textuels:

- On ne fait pas juste écrire des fichiers `.txt` : on les structure, on les analyse, et on les décrit formellement
- Et cela repose sur une hiérarchie de définitions, tout comme en modélisation

Cette analogie va nous être utile pour comprendre la suite : comment définir un langage textuel, comment le parser, et comment le transformer.

Compilation Process

- Source code
  - Concrete syntax used for specifying a program
  - Conformant to a grammar
- Lexical analysis
  - Converting a sequence of characters into a sequence of **tokens**
- Parsing (Syntactical analysis)
  - Abstract Syntax Tree (AST)



Avant de plonger dans l'outil Xtext, il est essentiel de rappeler comment fonctionne le processus classique de compilation d'un langage textuel.
Cette slide résume les étapes principales, que l'on retrouve dans tous les compilateurs, et que Xtext automatise en grande partie.
1. Source code
Tout commence par le code source écrit par l'utilisateur.
Ce code respecte une syntaxe concrète, avec une forme spécifique (mots-clés, parenthèses, ponctuation...)
et est censé être conforme à une grammaire définie au préalable.
Il s'agit donc d'une succession de characters
2. Lexical analysis (analyse lexicale)
Première étape du traitement :
on prend une chaîne de caractères, et on la transforme en une séquence de tokens.
Un token, c'est une unité lexicale comme :

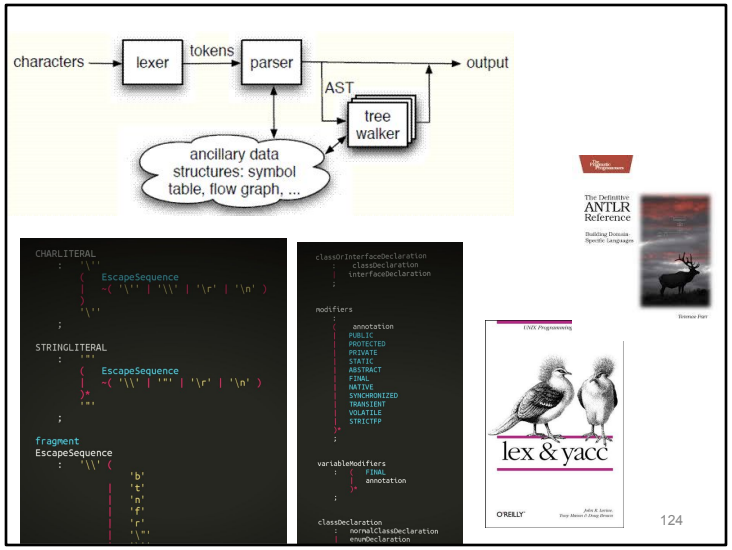
- un mot-clé (`if`, `function`, `class`)
- un identifiant (`myVar`, `x`)
- un nombre (`42`)

- ou un symbole (=, {, ;)

Cette étape est souvent appelée "tokenization", et c'est le **Lexer** qui prend en charge cela
Elle ne comprend aucune logique de structure, juste une classification des symboles.
3. Parsing (syntactical analysis)
Ensuite vient l'analyse syntaxique, effectuée par le **Parser**
À partir des tokens, le compilateur vérifie que la structure respecte la grammaire du langage,
et construit un Abstract Syntax Tree (AST).
L'AST est une représentation hiérarchique du programme, plus proche de la structure logique que de la syntaxe d'origine.
C'est sur cet arbre que reposent ensuite :

- les analyses sémantiques,
- les transformations,
- ou la génération de code.

Que les **Tree Walkers** font.
Ce processus de passage du texte vers l'AST, est exactement ce que Xtext prend en charge automatiquement quand vous lui fournissez une grammaire.
Et l'un des grands avantages de Xtext, c'est qu'il génère tout cela pour vous,
du scanner jusqu'à l'éditeur syntaxique Eclipse, en passant par le parseur et l'AST.



En bas à gauche : un extrait de grammaire ANTLR
On voit ici la définition de littéraux de caractères et de chaînes de caractères (`CHARLITERAL`, `STRINGLITERAL`).
Ces règles utilisent une syntaxe EBNF (ou proche) pour spécifier précisément ce qu'un token doit reconnaître :

- Des guillemets
- Des séquences d'échappement (`\n`, `\t`, etc.)

C'est ce qu'on appelle la grammaire lexicale.
En bas à droite : une grammaire syntaxique
Toujours dans ANTLR, cette fois on définit :

- une classe ou une interface
- la liste des modificateurs Java possibles

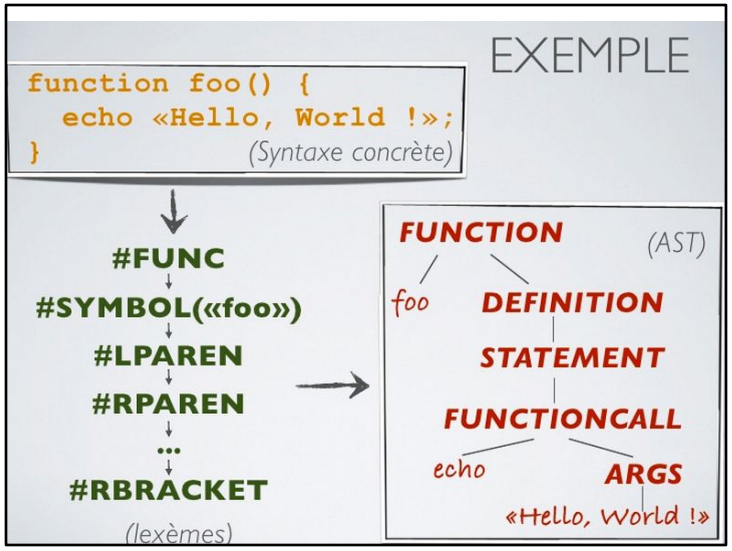
On est ici à un niveau plus haut : la structure des phrases du langage,
par exemple "public class MyClass { … }".
C'est la grammaire contextuelle, qui produit l'AST.
À droite : deux ouvrages de référence

- Lex & Yacc, pionniers des analyseurs syntaxiques sur Unix
- ANTLR, un outil plus moderne, orienté objet, très utilisé pour créer des DSLs

Ce sont les ancêtres directs de Xtext, qui s'inspire des mêmes principes, mais avec un intégration Eclipse complète,
et une chaîne de traitement entièrement générée (parsing, validation, édition…).
Pour résumé :

- On passe du texte à une structure exploitable
- Cette structure est décrite formellement par des grammaires
- Et des outils comme ANTLR ou Lex/Yacc ont longtemps été la norme

Xtext se situe dans cette lignée, mais en y ajoutant toute une infrastructure moderne de développement de langages.



Prenons maintenant un exemple simple et complet, pour visualiser les différentes étapes du traitement syntaxique.
En haut, on a la syntaxe concrète :
C'est une petite fonction, qu'on écrirait dans un langage de type PHP ou similaire.
C'est ce que l'utilisateur tape dans un fichier source.
C'est cette forme qu'on appelle la syntaxe concrète : ce qu'on voit, ce qu'on écrit.
Étape 1 : l'analyse lexicale – les lexèmes
La chaîne de caractères passe d'abord par le lexer, qui la découpe en lexèmes (ou tokens) :

- `#FUNC` pour le mot-clé `function`
- `#SYMBOL(«foo»)` pour le nom de la fonction
- `#LPAREN`, `#RPAREN`, `#LBRACKET`, etc. pour les symboles de ponctuation
- Des tokens pour `echo`, la chaîne `"Hello, World !"`, etc.

Chaque mot ou symbole est reconnu, typé, et identifié.
Étape 2 : l'analyse syntaxique – l'AST
Ensuite, ces tokens sont envoyés au parser,
qui reconstruit la structure logique du programme, sous forme d'un Abstract Syntax Tree (AST).
On voit ici que :

- L’élément racine est un FUNCTION
- Il contient un identifiant (`foo`) et un bloc de définition
- À l’intérieur, on trouve une instruction (`STATEMENT`)
- Qui est un appel de fonction (`FUNCTIONCALL`)
- Avec un identifiant (`echo`) et une liste d’arguments (`ARGS`), ici `"Hello, World !"`

Pourquoi c’est important
Cette structure n’est pas visible par l’utilisateur,
mais c’est elle qui est manipulée par les outils de compilation, de transformation, ou d’interprétation.
Et c’est exactement ce que va générer Xtext à partir d’une grammaire.
Donc ici, on voit bien :

- Concret → Lexèmes → Arbre syntaxique
- Et chaque étape est systématique, mécanisée, et déductible depuis une grammaire bien définie

Xtext va nous permettre de définir cette grammaire,
et de générer automatiquement le lexer, le parser et la structure de l’AST.

```scala
class StringInterp {
  val int = 42
  val dbl = Math.PI
  val str = "My hovercraft is full of eels"

  println(s"String: $str Double: $dbl Int: $int Int Expr: ${int * 1.0}")
}
```

Scala AST (example)

```
Block(
  List(
    ClassDef(Modifiers(), TypeName("StringInterp"), List(), Template(
      List(Ident(TypeName("AnyRef"))), noSelfType, List(DefDef(Modifiers(), termNames.CONSTRUCTOR,
        List(),
        List(List()),
        TypeTree(), Block(List(Apply(Select(Super(This(typeNames.EMPTY), typeNames.EMPTY),
        termNames.CONSTRUCTOR), List())), Literal(Constant(()))), ValDef(Modifiers(), TermName("int"),
        TypeTree(), Literal(Constant(42))), ValDef(Modifiers(), TermName("dbl"), TypeTree(),
        Literal(Constant(3.141592653589793))), ValDef(Modifiers(), TermName("str"), TypeTree(),
        Literal(Constant("My hovercraft is full of eels"))), Apply(Select(Ident(scala.Predef),
        TermName("println")), List(Apply(Select(Apply(Select(Ident(scala.StringContext), TermName("apply")),
        List(Literal(Constant("String: ")), Literal(Constant(" Double: ")), Literal(Constant(" Int: ")),
          Literal(Constant(" Int Expr: "))), Literal(Constant(""))), TermName("s")),
        List(Select(This(TypeName("StringInterp")), TermName("str")), Select(This(TypeName("StringInterp")),
          TermName("dbl")), Select(This(TypeName("StringInterp")), TermName("int")),
          Apply(Select(Select(This(TypeName("StringInterp")), TermName("int")), TermName("$times")),
            List(Literal(Constant(1.0)))))))))
    ))), Literal(Constant(())))
```

Ici, on change un peu d'échelle.
Ce qu'on voit, c'est un exemple réel d'AST produit par le compilateur Scala pour une classe très simple.
En haut : le code source Scala
La classe `StringInterp` contient :

- trois valeurs (`int`, `dbl`, `str`) avec des affectations simples,
- une instruction `println` qui utilise l'interpolation de chaînes (c'est-à-dire du code inséré directement dans une chaîne de caractères avec `$`).

Pour un humain, ce code est simple à lire, compréhensible, presque trivial.
En bas : l'AST correspondant
Ce que produit le compilateur, par contre, est beaucoup plus riche :

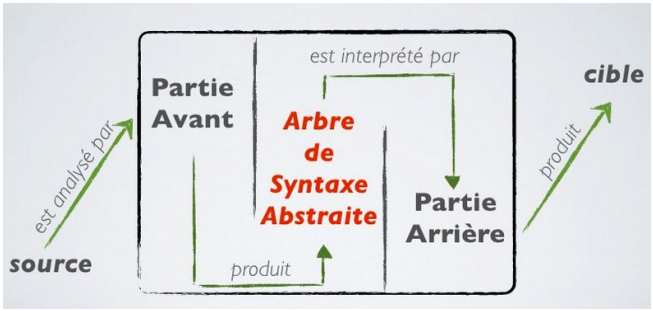
- On retrouve bien la structure de la classe (`ClassDef`, `Template`, `ValDef`, etc.)
- Chaque constante, appel, opérateur ou chaîne est décomposé en éléments distincts
- L'interpolation de la chaîne de caractères est traduite en appel de méthode complexe (avec des `Apply`, `Select`, `Literal`, etc.)
- Les noms de variables deviennent des objets (`TermName`, `Ident`)
- Même la constante `42` est traduite en `Literal(Constant(42))`

Deux enseignements ici :

1. Un AST, ce n'est pas fait pour les humains
   Il est souvent verbeux, imbriqué, et représente toute l'information sémantique et syntaxique
   que le compilateur a besoin pour transformer, analyser, ou générer du code.
2. On ne construit pas ça à la main !
   C'est justement là que des outils comme ANTLR, Xtext, ou les APIs de compilation comme celles de Scala
   ou Java interviennent.

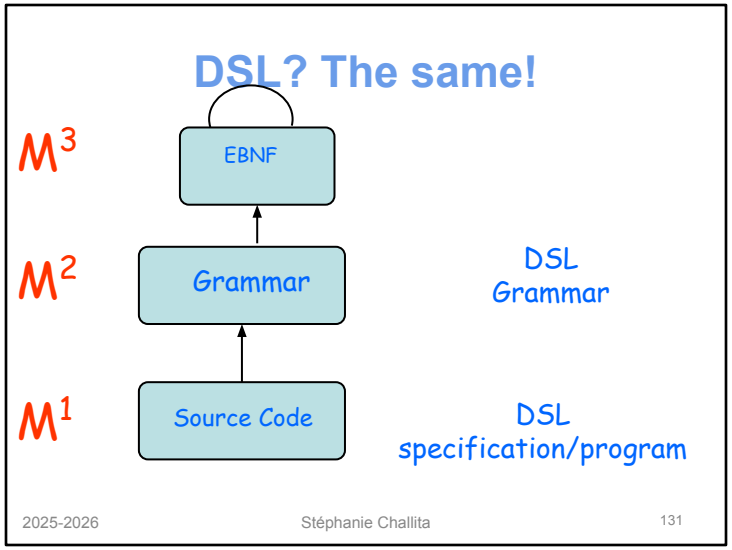
L'intérêt de Xtext, c'est qu'il permet de décrire des langages textuels à un haut niveau,
et que tout ça — parser, AST, éditeur, validations, etc. — est généré automatiquement,
sans qu'on ait besoin de plonger à ce niveau de détail (sauf en cas d'extension avancée).
On va donc pouvoir passer à la suite :
comment décrire notre propre langage avec Xtext, et à quoi ressemble la syntaxe d'une grammaire Xtext.

Compilation (en français)



On retrouve les mêmes étapes que précédemment, présentées ici en français :

- *Grammaire → Source → Analyse lexicale → Analyse syntaxique → AST*
- *Cette base est la même pour les langages de programmation classiques que pour les DSLs.*



Ce schéma reprend exactement la structure à trois niveaux qu'on a vue avec les compilateurs classiques :

- En bas : le code source
- Au milieu : la grammaire
- En haut : la méta-grammaire, ici sous la forme d'EBNF

Mais cette fois, on est dans le contexte d'un DSL, et plus précisément d'un DSL défini avec Xtext.

À droite : des précisions sur les concepts

- Le code source, c'est ici une spécification ou un programme écrit dans notre DSL.
  C'est ce que l'utilisateur final écrira.
- La grammaire, c'est le DSL Grammar que nous allons définir avec Xtext.
  Elle décrit la syntaxe de notre langage, ce qu'on peut écrire, dans quel ordre, avec quelle structure.
- L'EBNF reste la base : c'est le formalisme qui structure la grammaire Xtext elle-même.

Ce que ça signifie:

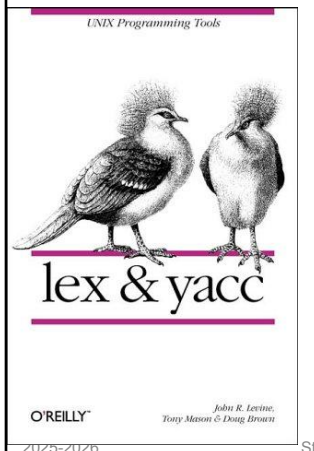
Créer un DSL, c'est écrire un langage. Et écrire un langage, c'est toujours décrire un modèle.

Avec Xtext :

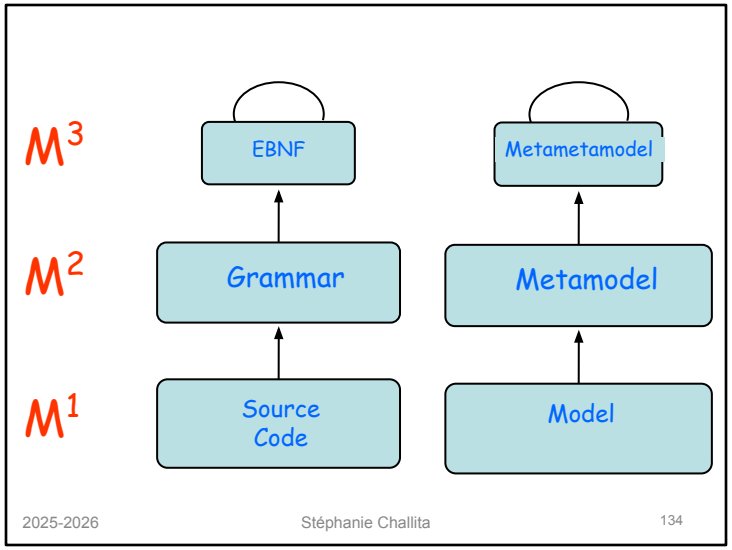
- On définit une grammaire (niveau M2)
- Qui permet de créer des programmes (niveau M1)

- Et cette grammaire repose sur une syntaxe bien formalisée (niveau M3)

Autrement dit : Même principes. Même pipeline. Même hiérarchie.


UNIX Programming Tools
lex & yacc
O'REILLY
John R. Levine,
Tony Mason & Doug Brown


The Pragmatic Programmers
The Definitive
ANTLR
Reference
Building Domain-
Specific Languages
Terence Parr



Ici, on connecte deux univers que l'on traite souvent séparément :

- D'un côté : le monde des langages textuels, avec EBNF, les grammaires, et le code source
- De l'autre : le monde de la modélisation, avec les métamodèles, les modèles, et les méta-niveaux

À gauche : la hiérarchie syntaxique, comme on l'a déjà vue

À droite : la hiérarchie modélisation (MDA)

- M3 – Métamétamodèle : en EMF, c'est Ecore ou MOF.
  Il permet de décrire des métamodèles.
- M2 – Métamodèle : un modèle conforme à Ecore, par exemple un modèle qui décrit la structure d'un DSL.
- M1 – Modèle : une instance du métamodèle — un fichier `.xmi`, `.model`, ou un programme dans le langage qu'on a décrit.

Ce qu'il faut comprendre:

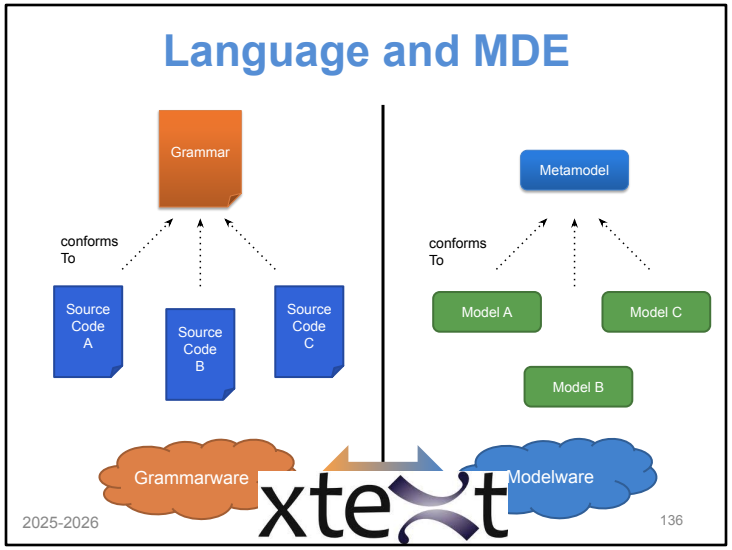
- Une grammaire (M2 syntaxique) est un métamodèle (M2 modélisation).
- Un programme est un modèle.
- Et EBNF joue le rôle de métamétamodèle syntaxique — exactement comme Ecore dans EMF.

Autrement dit :

La construction d'un langage est, elle aussi, un processus de modélisation.

C’est cette correspondance qui fait la force de Xtext :

- On écrit une grammaire textuelle,
- Xtext en déduit un métamodèle EMF,
- Et les fichiers écrits dans ce langage deviennent des modèles EMF manipulables.



Ici, on a deux grandes approches de la conception de langages et d'outils :
celle par grammaires et celle par modèles.
Côté gauche : Grammarware

- On part d'une grammaire, généralement écrite en EBNF ou dans un DSL comme Xtext, ANTLR, etc.
- À partir de cette grammaire, on peut produire du code source conforme à cette syntaxe.
- Chaque fichier source (ici A, B, C) "conforme à" la grammaire.
- Le traitement repose sur des outils comme les compilateurs, parseurs, lexers, analyseurs statiques, etc.

Cette approche vient du monde de la compilation traditionnelle.
Côté droit : Modelware

- On part cette fois d'un métamodèle, défini avec Ecore (EMF) ou tout autre cadre MDA.
- À partir de ce métamodèle, on peut construire des modèles : ici, les modèles A, B, C.
- Chaque modèle "conforme à" son métamodèle.
- Le traitement se fait via des outils de modélisation, validation, transformation de modèles (M2M, M2T...).

C'est l'approche plus moderne, centrée sur la modélisation explicite.
Et Xtext alors ?
Xtext se trouve au croisement exact de ces deux mondes :

- On définit une grammaire (approche grammarware),
- Qui génère automatiquement un métamodèle EMF (approche modelware),
- Et les fichiers source deviennent des modèles EMF qu'on peut manipuler avec tout l'écosystème model-driven.

Pourquoi c'est important ?

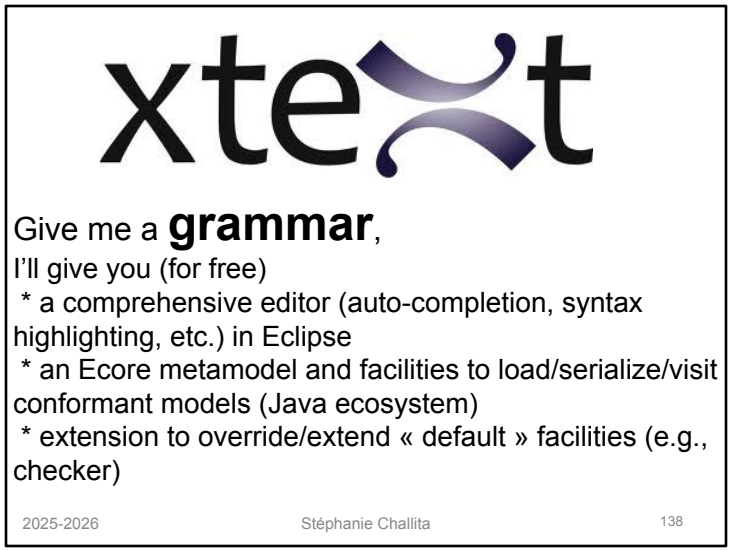
Parce qu'on a souvent tendance à opposer ces deux approches — alors que dans Xtext, elles sont intégrées.

"Un langage textuel devient un modèle, et un modèle devient exécutable."

Transition

C'est cette dualité qui fait de Xtext un outil aussi puissant pour les DSLs.

On va maintenant plonger dans les outils générés automatiquement à partir d'une simple grammaire : éditeur, validation, linking, et bien plus.



Voici Xtext, et voici sa promesse :

"Give me a grammar, I'll give you the rest."

C'est ça, Xtext. Vous lui fournissez simplement une grammaire déclarative,
et il vous génère automatiquement tout un environnement complet autour de votre langage.

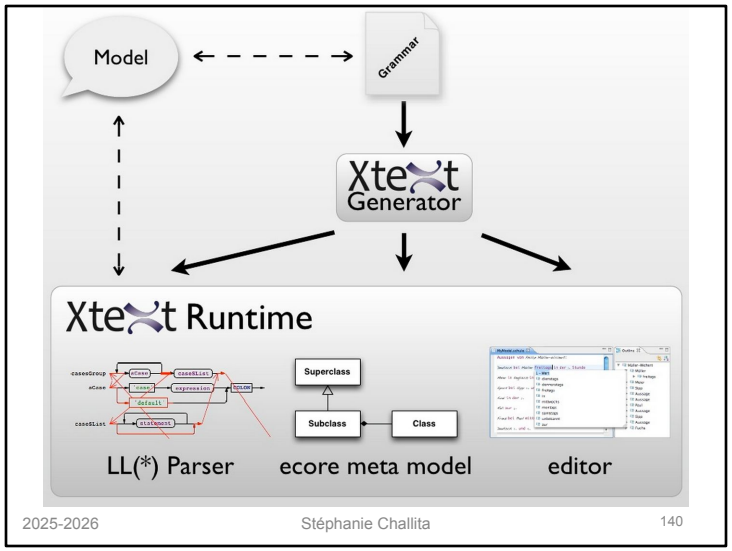
Qu'est-ce qu'il vous donne ?

1. Un éditeur complet dans Eclipse
   - Auto-complétion, coloration syntaxique, vérifications, navigation, folding…
   - Ce n'est pas juste un fichier texte : c'est un vrai IDE généré pour votre langage.
2. Un métamodèle Ecore
   - La grammaire est traduite en un métamodèle EMF.
   - Vous pouvez alors manipuler les fichiers de votre DSL comme des modèles EMF, les charger, les valider, les transformer, les sérialiser.
3. Des points d'extension
   - Vous pouvez personnaliser le comportement du langage : ajout de règles de vérification, transformation personnalisée, génération de code, etc.
   - Tout est extensible via des hooks Java ou Xtend, selon vos besoins.

Pourquoi c’est révolutionnaire ?
Parce qu’avant Xtext, construire un langage, c’était :

- Écrire une grammaire
- Écrire un parseur à la main
- Coder un éditeur, un validateur, une transformation
- Maintenir tout ça en parallèle

Avec Xtext, vous décrivez votre langage une seule fois, et le reste est généré pour vous —
tout en restant customisable et intégré à l’écosystème Eclipse + Java.
On va maintenant voir le cycle de génération et d’utilisation d’un langage avec Xtext. et ce qu’elle produit exactement



1. Tout commence par une grammaire (en haut à droite)
   - On écrit une grammaire Xtext, qui décrit la syntaxe concrète et abstraite de notre langage.
   - Cette grammaire est l'élément d'entrée principal du framework.
2. Cette grammaire est passée au Xtext Generator
   - C'est l'outil qui génère tout le reste automatiquement.
   - Une seule grammaire sert de base à :
     - un parseur,
     - un métamodèle Ecore,
     - un environnement d'édition complet.
3. Le résultat est intégré dans le Xtext Runtime

On distingue ici trois grandes briques générées :

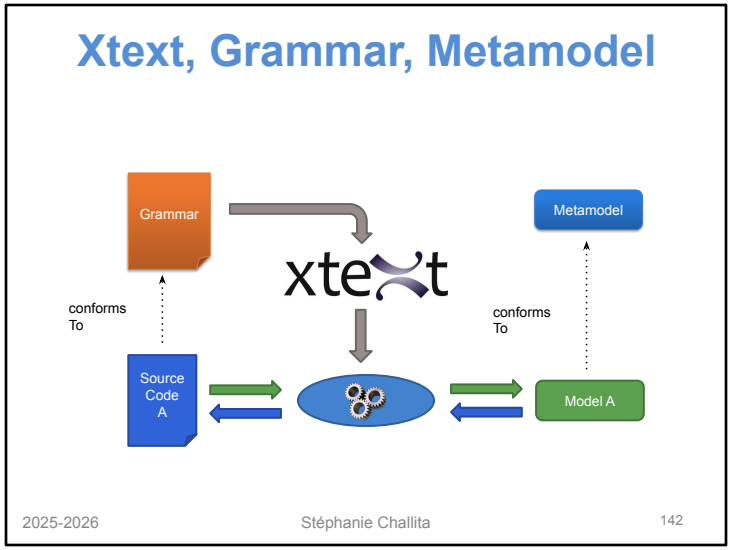
- LL(*) Parser
  Un parseur performant, basé sur ANTLR, généré automatiquement à partir de la grammaire.
  Il convertit un fichier texte en un modèle EMF.
- Ecore Metamodel
  La grammaire est traduite en modèle Ecore, qui devient le métamodèle pour les modèles que l'utilisateur

- écrit dans ce langage.
  Les instances sont des objets EMF classiques, manipulables dans tout l'écosystème Eclipse.
- Editor
  Un éditeur Eclipse complet est généré :
  - auto-complétion,
  - coloration syntaxique,
  - navigation dans le code,
  - validation personnalisée, etc.

4. Résultat final : un modèle conforme au métamodèle
   - Lorsqu'un utilisateur écrit du code dans votre langage,
     ce n'est pas juste une chaîne de caractères : c'est un modèle EMF conforme au métamodèle généré.
   - Il peut être analysé, transformé, ou exécuté avec d'autres outils MDE.

Ce qu'il faut retenir

À partir d'une seule grammaire,
Xtext vous donne un parseur, un métamodèle EMF, et un environnement d'édition complet.
Et ce, de manière cohérente, extensible et industrialisable.



Ici, on synthétise très clairement le double rôle de Xtext :
il fait le lien entre la grammaire d'un langage et le métamodèle EMF, tout en assurant la cohérence entre les deux mondes: celui du texte source et celui des modèles.
À gauche : le monde du texte

- En bas à gauche, on a un fichier source écrit dans le langage que l'on a défini.
- Ce fichier est conforme à la grammaire (en orange) que l'on a spécifiée avec Xtext.
- La grammaire définit à la fois la syntaxe concrète et la structure abstraite du langage.

À droite : le monde du modèle

- En analysant le texte, Xtext génère automatiquement un modèle EMF (ici "Model A").
- Ce modèle est une instance du métamodèle généré par Xtext à partir de la grammaire.
- Ce métamodèle est un Ecore, exactement comme on le ferait à la main avec EMF.

Au centre : le moteur Xtext

- La flèche descendante montre que Xtext transforme la grammaire en deux choses :
  - Un parseur (qui comprend le texte),
  - Un métamodèle EMF (qui structure les modèles en mémoire).
- Le bloc central avec les engrenages symbolise le runtime Xtext :

- il assure la conversion entre fichier source et modèle EMF, et inversement.

Les doubles flèches bleues et vertes

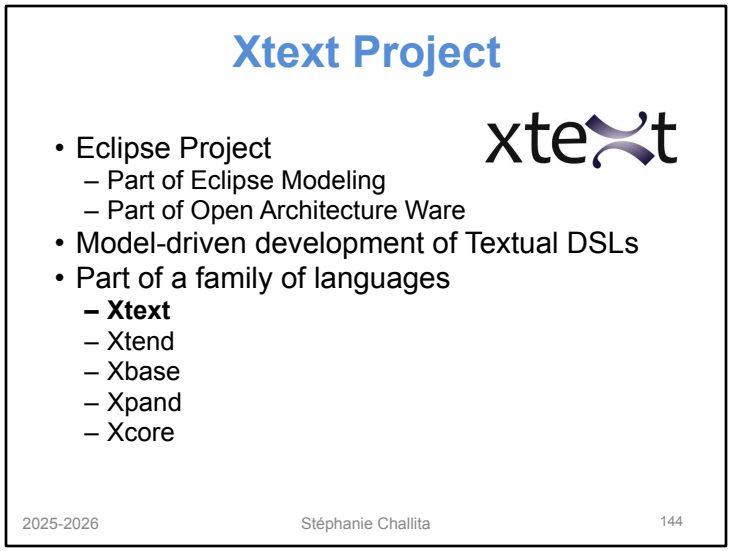
- Vert : on part du texte, et on génère un modèle.
- Bleu: à partir du modèle, on peut aussi régénérer le texte (round-trip possible).

Ce qu'il faut retenir

Xtext relie deux mondes :

- À gauche : le texte, syntaxe, grammaire.
- À droite : les modèles EMF, outils MDE, transformations, validations.

Et tout cela à partir d'un seul fichier de grammaire Xtext.



Pour bien comprendre Xtext, il faut aussi le situer dans son contexte technique et communautaire.

- Xtext est un projet Eclipse
  - Il fait partie intégrante de l'écosystème Eclipse Modeling.
  - Il est open source, soutenu par la fondation Eclipse, avec une forte communauté derrière.
- Héritage de Open Architecture Ware (oAW)
  - Xtext est issu d'un projet plus large, Open Architecture Ware, qui avait pour but de créer une infrastructure pour le développement MDE.
  - L'idée était d'industrialiser le développement de langages spécifiques — avec de meilleurs outils que ceux proposés traditionnellement par les compilateurs classiques.
- Xtext est modèle-centré
  - À la différence des outils purement syntaxiques, Xtext est pensé pour la modélisation.
  - Le texte est vu comme une représentation du modèle, et non l'inverse.
  - Cela permet une intégration fluide avec EMF et tous les outils MDE d'Eclipse.
- Une famille de langages

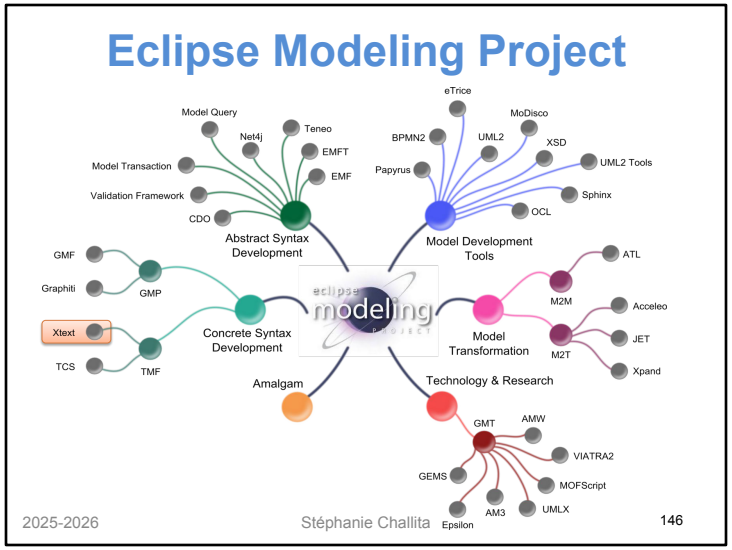
Xtext s'inscrit dans un ensemble d'outils conçus pour le développement orienté langage :

- Xtext : pour définir la grammaire, parser le texte et générer un modèle EMF.

- Xtend : langage de transformation très lisible, qui compile vers du Java.
- Xbase : une base commune de grammaire et d'expressions (ex. : pour ajouter des expressions dans vos DSLs).
- Xpand : un langage de template pour la génération de code.
- Xcore : une façon textuelle de décrire des métamodèles Ecore, avec une syntaxe proche de Java.

Ce qu'il faut retenir

Xtext n'est pas juste un parseur ou un éditeur :
C'est un outil modèle-dirigé, pensé pour s'inscrire dans un écosystème complet,
et intégré à Eclipse pour vous aider à construire rapidement des langages puissants et industriels.



En prenant un peu de recul, on voit que Eclipse Modeling regroupe plusieurs projets, chacun spécialisé sur un aspect du développement dirigé par les modèles.
On peut organiser ces projets autour de 5 grands axes :
1. Abstract Syntax Development (vert foncé)
C'est ici qu'on retrouve les fondations comme :

- EMF et Ecore, pour définir les métamodèles,
- CDO, Teneo, Net4j : pour la persistance, la distribution, l'interopérabilité,
- Validation Framework, Model Transaction : pour la robustesse des modèles.

2. Concrete Syntax Development (vert clair)
Il s'agit de la couche textuelle ou graphique pour manipuler des modèles.

- Xtext (en orange ici) permet de définir des syntaxes textuelles avec un parseur, un éditeur, et un lien vers EMF.
- TCS, TMF, GMF, Graphiti : permettent de définir des syntaxes concrètes graphiques.

3. Model Development Tools (bleu)
Des outils pour modéliser graphiquement selon des standards :

- UML2 Tools, Papyrus, eTrice, BPMN2, Sphinx.

- Souvent utilisés dans l’industrie pour des projets logiciels, cyber-physiques ou métiers.

4. Model Transformation (rose)

Quand on veut transformer un modèle en un autre modèle ou en code, on fait appel à ces outils :

- ATL, Acceleo, JET (génération),
- M2M et M2T : transformation modèle à modèle ou modèle à texte,
- Xpand : génération template puissante.

5. Technology & Research (rouge)

Des projets plus exploratoires ou académiques :

- AM3, Epsilon, GEMS, GMT, VIATRA, etc.
- Ces outils permettent de faire de la recherche sur la composition, la transformation multi-niveaux, l’analyse de modèles complexes, etc.

Xtext n’est qu’un projet parmi beaucoup d’autres dans un écosystème riche et intégré.

Eclipse Modeling offre toute la chaîne outillée pour concevoir, manipuler, transformer et exploiter des modèles.

The Grammar Language of Xtext

- Corner-stone of Xtext
- A… DSL to define textual languages
  - Describe the concrete syntax
  - Specify the mapping between concrete syntax and domain model
- From the grammar, it is generated:
  - The domain model
  - The parser
  - The tooling

Ce slide présente l'idée centrale derrière Xtext :

> Xtext est lui-même un DSL – un langage spécifique au domaine – dédié à la définition de langages textuels.

Concrètement, vous écrivez une grammaire dans un langage Xtext, et cela permet de :

1. Décrire la syntaxe concrète de votre langage (c'est-à-dire ce que les utilisateurs vont écrire).
2. Définir comment cette syntaxe se mappe vers un modèle sous-jacent (c'est-à-dire l'abstraction métier, souvent un modèle EMF).

Et à partir de cette grammaire, Xtext vous génère automatiquement :

- Le modèle métier : un métamodèle EMF, avec des EClasses, EAttributes, etc.
- Le parseur : qui sait transformer un fichier texte en un modèle en mémoire.
- L'outillage Eclipse : éditeur avec auto-complétion, surlignage, navigation, validation…

Cela boucle le cycle qu'on a vu : à partir de la grammaire, on obtient un langage complet — syntaxe + structure + outil.

On pourrait résumer en disant :

> "Vous écrivez une grammaire, Xtext vous donne un langage !"

Main Advantages

- Consistent look and feel
- Textual DSLs are a resource in Eclipse
- Open editors can be extended
- Complete framework to develop DSLs
- Easy to connect to any Java-based language

Xtext n'est pas juste un parseur ou un générateur de code — c'est un écosystème complet pour développer des langages textuels. Voici quelques-uns de ses avantages majeurs :

Consistent look and feel

Xtext s'intègre parfaitement à l'environnement Eclipse. Les langages créés bénéficient immédiatement des conventions d'interface de l'IDE : menus, coloration syntaxique, navigation, etc.

Résultat : vos DSLs ont l'air "professionnels" sans effort supplémentaire.

Textual DSLs are resources in Eclipse

Chaque fichier écrit dans votre DSL est un vrai fichier de ressource Eclipse.

Cela signifie qu'il peut être versionné, ouvert, modifié, indexé, validé… comme n'importe quel autre artefact logiciel.

Open editors can be extended

Les éditeurs générés sont personnalisables. Vous pouvez ajouter :

- des règles de validation métier,
- de l'auto-complétion contextuelle,
- de la génération de code spécifique.

Bref, vous gardez le contrôle.

Complete framework to develop DSLs

Xtext couvre tout le cycle :

- définition syntaxique,
- parsing,
- édition,
- validation,
- génération,
- exécution.

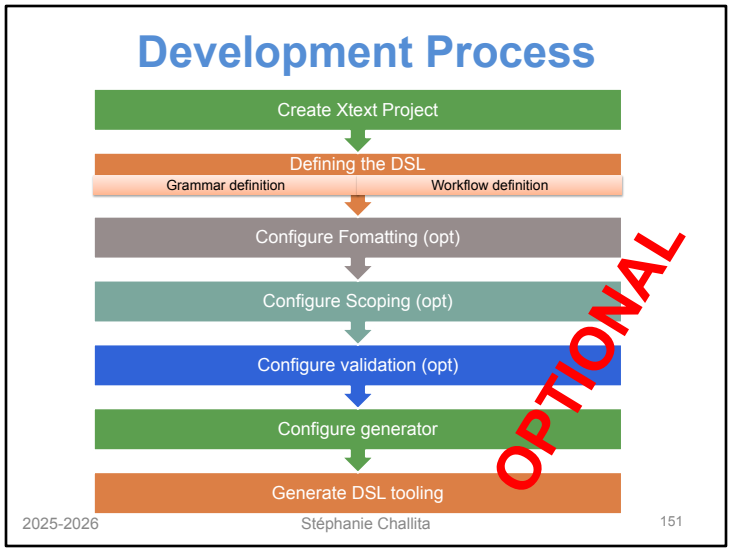
Pas besoin d'assembler des outils manuellement — tout est intégré.

Easy to connect to any Java-based language

Enfin, Xtext étant basé sur EMF et Java, il est facile de le connecter avec des bibliothèques ou frameworks Java existants :

- pour l'interprétation,
- pour la transformation,
- ou même pour l'intégration dans un outil plus large.

En résumé : Xtext vous fait gagner du temps, tout en vous laissant la flexibilité nécessaire pour construire des outils puissants et robustes.



Maintenant que l'on connaît les avantages d'Xtext, voyons comment on crée concrètement un langage avec lui. Ce schéma décrit les étapes typiques d'un projet Xtext, de la création initiale jusqu'à la génération complète des outils.

- On commence avec la création d'un projet Xtext.

C'est une opération guidée dans Eclipse, qui pose les fondations : structure du projet, dépendances, fichiers de configuration.

- Ensuite, on entre dans la définition du langage — c'est le cœur de notre travail.

Et là, on a deux branches principales qui peuvent être développées en parallèle :

À gauche : la définition grammaticale

C'est ici qu'on écrit la grammaire Xtext :

- elle décrit la syntaxe concrète du langage
- elle spécifie le lien avec le métamodèle Ecore

À partir de cette grammaire, Xtext génère l'AST, les classes Java, le modèle, etc.

À droite : la définition du workflow

Xtext utilise un workflow de génération — une sorte de script qui orchestre toutes les étapes automatiques de compilation et génération.

On y configure quels générateurs lancer, dans quel ordre, et comment chaîner les étapes de transformation.

Ensuite, on personnalise les comportements “par défaut” d’Xtext à travers plusieurs extensions optionnelles mais essentielles :

- Le formattage automatique du code (indentation, règles de style, etc.)
- Le scoping, c’est-à-dire la gestion des liens entre éléments du modèle (ex : résolution des identifiants)
- La validation sémantique, avec des règles que vous définissez pour vérifier la cohérence métier

🛠️ Puis, on passe à la configuration du générateur :

Que produit votre langage ? Du code Java, SQL, HTML, JSON ?

Vous définissez ici les règles de transformation vers votre cible.

Enfin, on génère l’environnement complet :

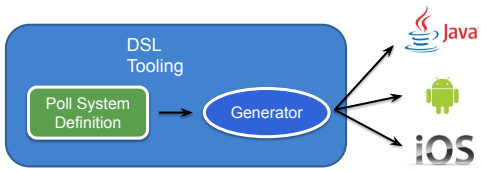
éditeur, parseur, infrastructure de tests, services Eclipse, tout est prêt pour utiliser et faire évoluer votre langage dans de vraies conditions.

Ce processus est modulaire, automatisé, et extensible :

Xtext vous donne un DSL… pour créer des DSLs.

Motivating Scenario

- Poll System application
  - Define a Poll with the corresponding questions
  - Each question has a text and a set of options
  - Each option has a text
- Generate the application in different platforms



Avant de se lancer dans la syntaxe d'une grammaire Xtext, posons une question simple :

Pourquoi aurait-on besoin d'un DSL ?

Cette slide propose un scénario simple mais très représentatif.

Le cas d'usage : une application de sondage (Poll System)

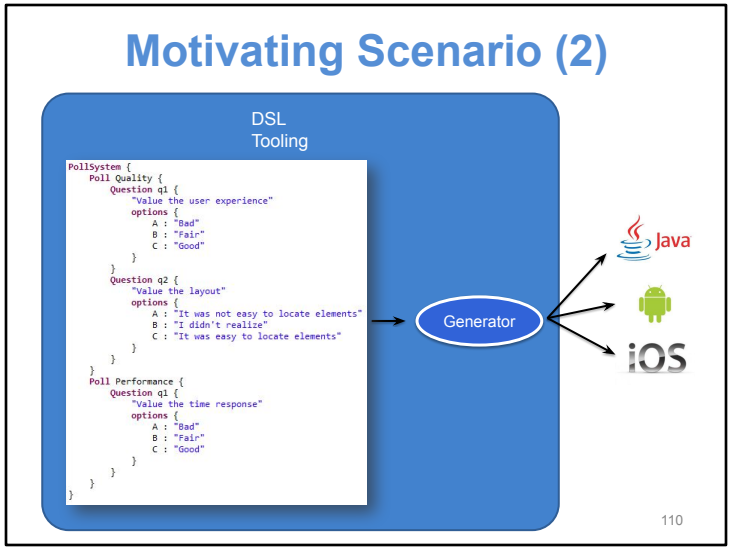
Imaginons qu'on veuille développer une application qui permet de :

- définir des **sondages**,
- chaque sondage contient une **liste de questions**,
- chaque question a un **libellé** (un texte),
- et une liste **d'options de réponse**, elles aussi décrites par un texte.

C'est une structure simple, bien définie, mais **répétitive** et **métier-spécifique**.

Et surtout, ce genre de contenu pourrait être :

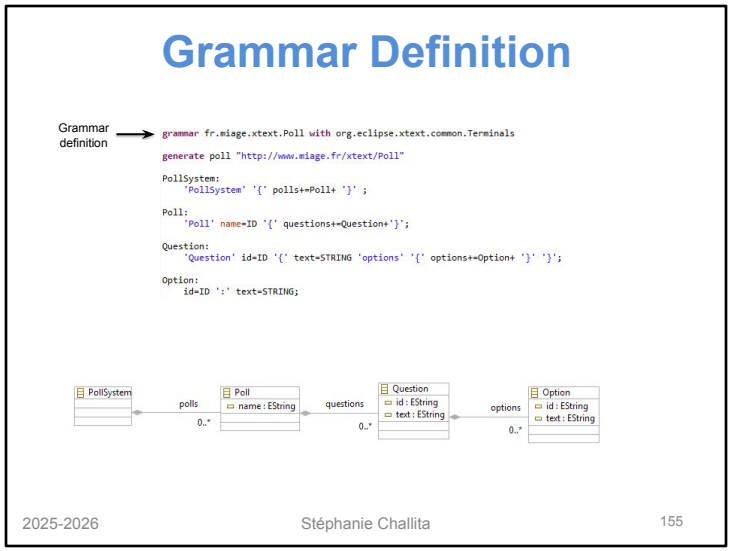
- édité par des non-développeurs,
- décliné vers **plusieurs plateformes** : une app web, un back-end, une version mobile…



L'idée

Plutôt que de coder chaque sondage à la main dans chaque langage cible,
pourquoi ne pas créer un **DSL dédié**, simple et lisible,
dans lequel on pourrait écrire quelque chose comme ca
Et ensuite, utiliser Xtext pour :

- générer du code Java, TypeScript ou HTML à partir de cette spécification,
- produire automatiquement une app, une API ou une interface graphique.



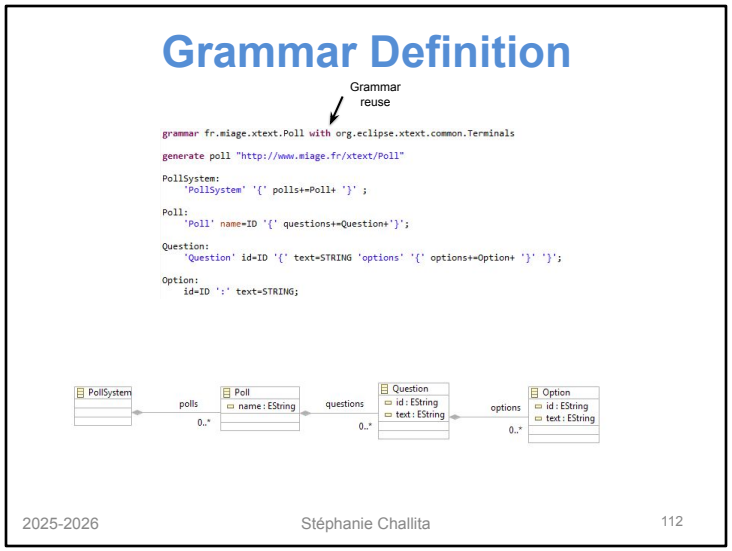
*C'est ici que l'on entre dans le cœur technique : la **définition de la grammaire** Xtext. Chaque slide détaille un aspect syntaxique :*

- **114** — *Point d'entrée logique :* `Grammar definition` *dans Xtext*
- **115** — *Xtext permet la **réutilisation de grammaires** existantes pour accélérer le développement (e.g., expressions)*
- **116** — *Un **métamodèle dérivé** (Ecore) est généré automatiquement à partir de la grammaire*
- **117** — *Les* `parser rules` *définissent les structures reconnues par le langage*
- **118** — *Les* `keywords` *sont des mots réservés dans votre langage : ils doivent être définis clairement pour éviter les conflits de parsing*
- **119** — *Les types d'affectation :*
  - `=` : affectation simple
  - `+=` : affectation multiple (liste)
  - `?=` : booléen (présence ou non d'un mot-clé)

Slide 1 – Grammar definition

Commençons par la base : dans Xtext, **tout langage commence par une grammaire**.
C'est ce fichier `.xtext` que vous allez écrire pour décrire la **structure de votre DSL**.

Cette grammaire a une syntaxe déclarative, lisible, qui ressemble beaucoup à de l'EBNF, et elle permet de **définir à la fois la syntaxe concrète et abstraite** du langage.



Une grammaire Xtext peut réutiliser des éléments d'autres grammaires.
Par exemple, si votre langage a besoin d'expressions arithmétiques, plutôt que de les redéfinir,
vous pouvez importer une grammaire commune comme `org.eclipse.xtext.common.Terminals`.
Cela permet d'accélérer le développement et de standardiser le comportement des parties génériques du langage.

Grammar Definition

Derived metamodel →

```
grammar fr.miage.xtext.Poll with org.eclipse.xtext.common.Terminals

generate poll "http://www.miage.fr/xtext/Poll"

PollSystem:
    'PollSystem' '{' polls+=Poll+ '}' ;

Poll:
    'Poll' name=ID '{' questions+=Question+'}';

Question:
    'Question' id=ID '{' text=STRING 'options' '{' options+=Option+ '}' '}';

Option:
    id=ID ':' text=STRING;
```

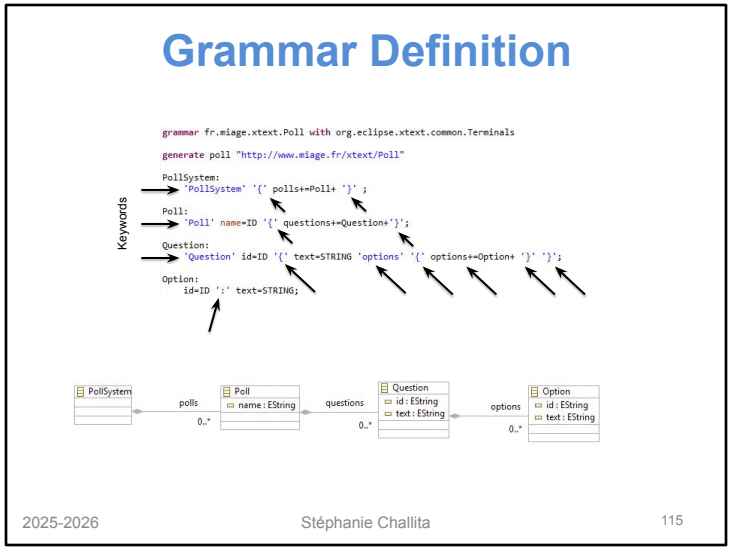
Chaque grammaire Xtext induit automatiquement un métamodèle EMF.
Cela veut dire que pour chaque règle que vous écrivez, Xtext génère des EClasses, EAttributes, EReferences qui forment la structure du modèle abstrait.
On peut donc dire que la grammaire Xtext génère le métamodèle — ce qui fait le lien direct entre la syntaxe textuelle et l'univers EMF.

Grammar Definition

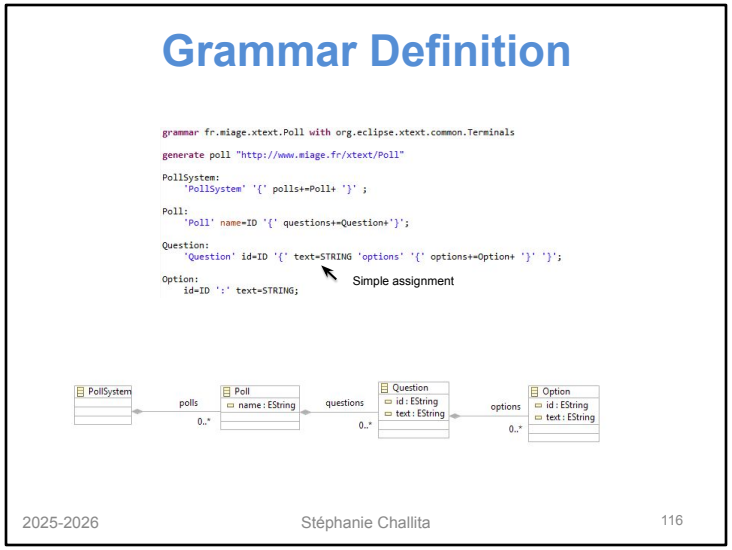
```
grammar fr.miage.xtext.Poll with org.eclipse.xtext.common.Terminals

generate poll "http://www.miage.fr/xtext/Poll"

PollSystem:
    'PollSystem' '{' polls+=Poll+ '}' ;

Poll:
    'Poll' name=ID '{' questions+=Question+'}';

Question:
    'Question' id=ID '{' text=STRING 'options' '{' options+=Option+ '}' '}';

Option:
    id=ID ':' text=STRING;
```

Parser Rules

Les parser rules sont les blocs principaux d'une grammaire.
Elles décrivent la structure des phrases de votre langage.
Chaque règle commence par un nom (qui devient une EClass) et se poursuit par une séquence d'éléments.
Ces règles définissent comment reconnaître les constructions du langage à partir du texte.



Dans Xtext, les mots-clés sont des chaînes de caractères entourées de guillemets (`"poll"`, `"question"`…).
Ils permettent de délimiter la syntaxe concrète du langage.
Ils sont fixes, reconnaissables par le parseur, et apparaissent aussi tels quels dans l'éditeur généré, avec coloration syntaxique.



Un simple assignment permet d'attribuer une valeur à une propriété dans le modèle.
Par exemple :

Question: 'question' text=STRING;

Ici, `text=STRING` signifie : "la propriété `text` du modèle reçoit la chaîne trouvée ici".
C'est la forme la plus courante d'affectation dans une grammaire Xtext.

Grammar Definition

```
grammar fr.miage.xtext.Poll with org.eclipse.xtext.common.Terminals

generate poll "http://www.miage.fr/xtext/Poll"

PollSystem:
    'PollSystem' '{' polls+=Poll+ '}' ;

Poll:
    'Poll' name=ID '{' questions+=Question+'}';

Question:
    'Question' id=ID '{' text=STRING 'options' '{' options+=Option+ '}' '}';

Option:
    id=ID ':' text=STRING;
```

Multivalue assignment

Quand on veut capturer plusieurs éléments (une liste), on utilise une assignation multivaluée :

Poll: 'poll' name=ID '{' questions+=Question* '}';

Le += indique que la propriété questions est une liste d'objets Question.

Ce genre de construction permet de représenter des arborescences (ex : un sondage avec plusieurs questions).

Grammar Definition

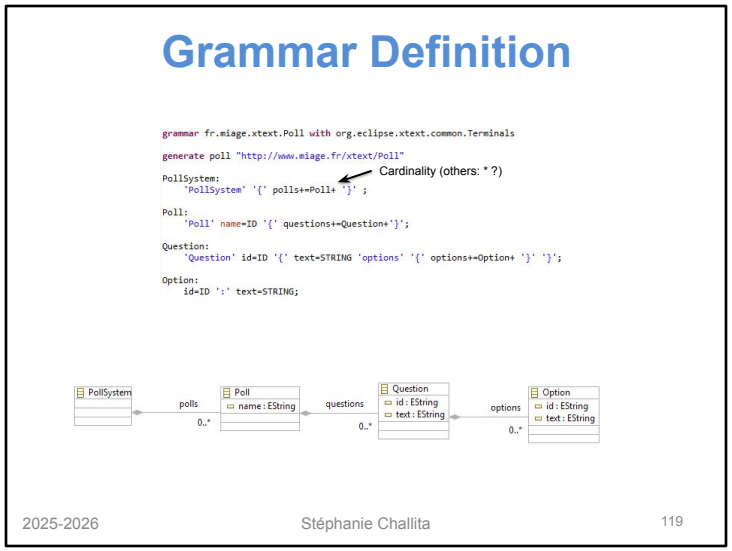
```
grammar fr.miage.xtext.Poll with org.eclipse.xtext.common.Terminals

generate poll "http://www.miage.fr/xtext/Poll"

PollSystem:
    'PollSystem' '{' polls+=Poll+ '}' ;

Poll:
    'Poll' name=ID '{' questions+=Question+'}';

Question:
    'Question' id=ID '{' text=STRING 'options' '{' options+=Option+ '}' '}';

Option:
    id=ID ':' text=STRING;
```

?= Boolean assignment

Le dernier assignment, non utilisé dans cette grammaire, et le `?=`, qui est utilisé pour les attributs booléens.
Il permet d'indiquer la présence ou l'absence d'un mot-clé, sans valeur associée :

```
Question: 'question' optional?='optional'? text=STRING;
```

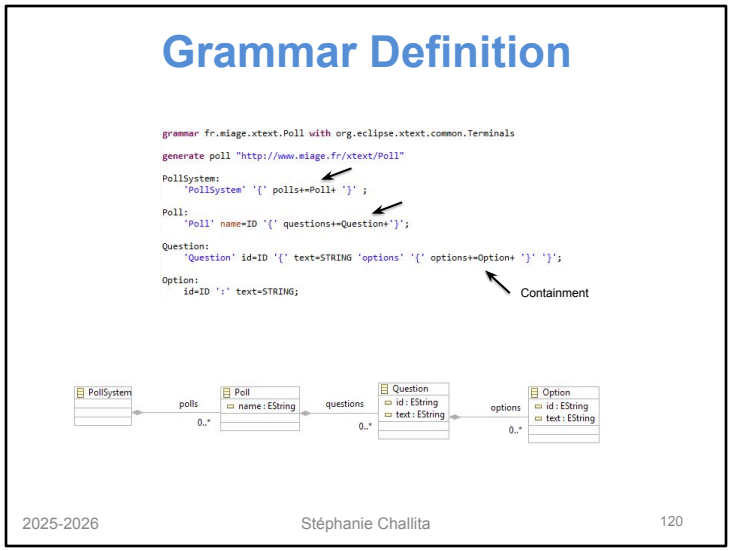
Si `"optional"` est présent dans le texte, la propriété booléenne sera `true`, sinon `false`.
C'est très pratique pour des modificateurs ou flags.



Enfin, Xtext permet de gérer la cardinalité des éléments, de façon similaire à UML :

- ? → zéro ou un (optionnel)
- * → zéro ou plusieurs (liste)
- + → un ou plusieurs (liste obligatoire)

Elles fonctionnent comme en EBNF, mais elles influencent aussi le type généré dans le métamodèle (attribut simple ou liste).



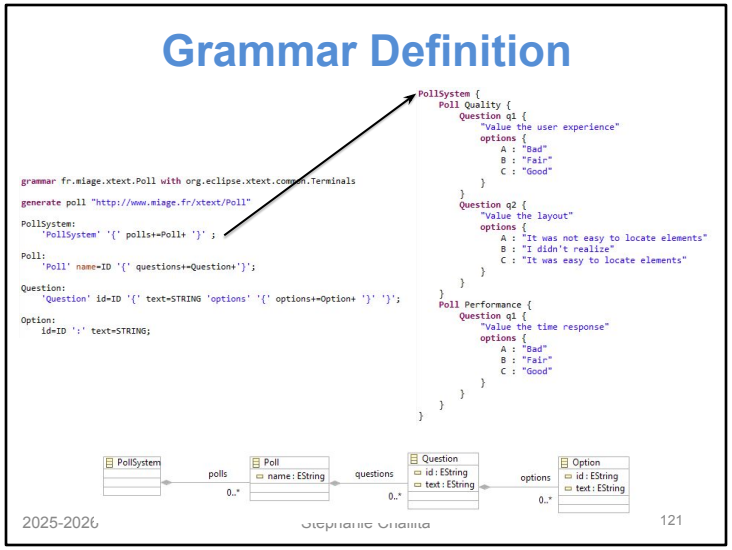
Enfin, il est essentiel de comprendre la notion de containment :

quand vous utilisez `+=`, vous déclarez une relation de composition entre les éléments.

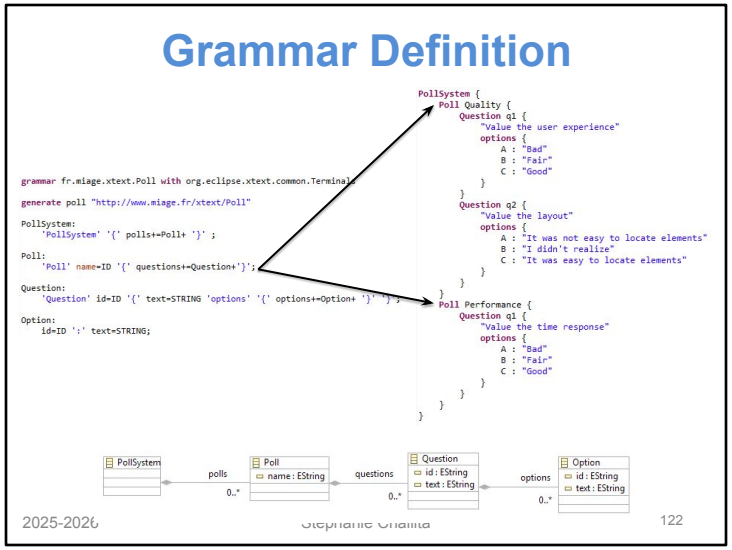
Cela signifie que l'objet parent possède les objets enfants, comme un `Poll` possède ses `Question`.

Par défaut, les affectations en `+=` créent des contenus EMF imbriqués (containment = true).

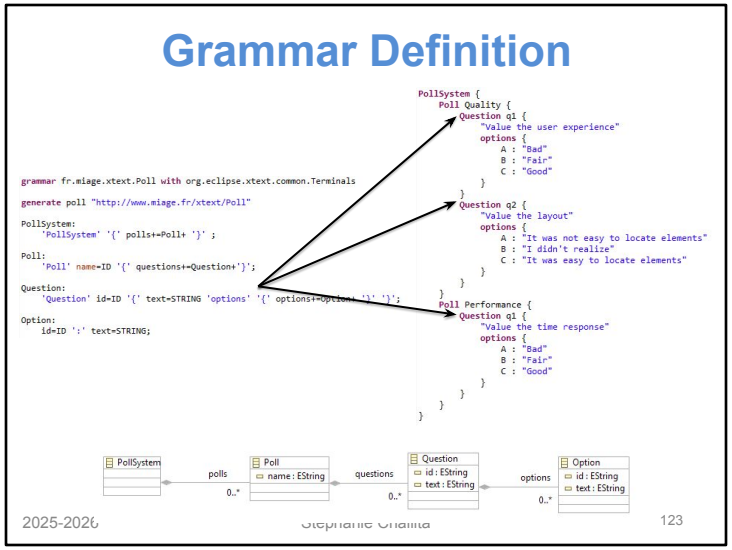
Si on souhaite faire une simple référence (non containment), il faudra l'indiquer explicitement.



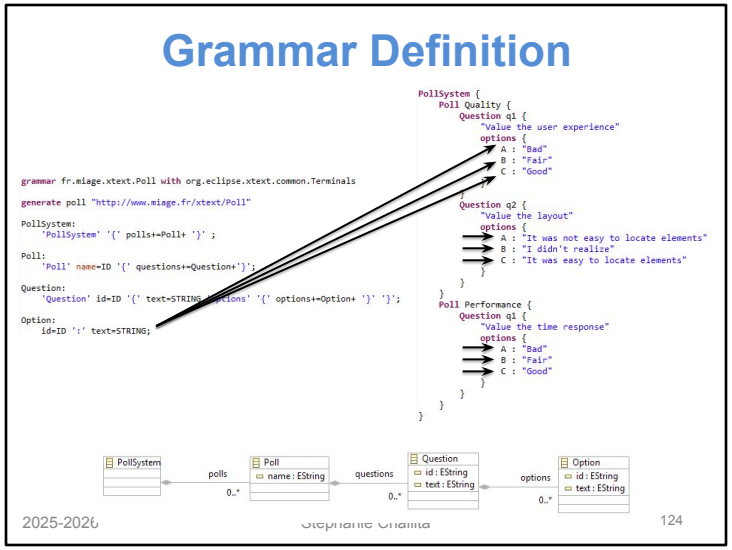
Ici, on peut voir la correspondance entre la grammaire et un exemple de programme de cette grammaire
Ici, on veut tout en haut à droite, la définition du PollSystem, c'est la racine du programme.
Le PollSystem est ensuite un enchaînement de Poll



Puis, on voit deux définitions de Poll, avec leur nom respectifs, Quality et Performance.
Les Poll sont ensuite un enchaînement de Question



On a alors la correspondance ici pour les questions, il y a en 3. De même, les questions ont des Ids, ici q1, q2 et q1.
On peut le text pour chacune d'entre elle et enfin les options.



Finalement, on a ici toutes les options rattachées à leur question respective. Ces options ont un id et un text de type string.
Ce DSL est simple, mais ultra efficace pour définir des questions, et en plus, accessible à des non-développeurs car il a une syntaxe spécifique