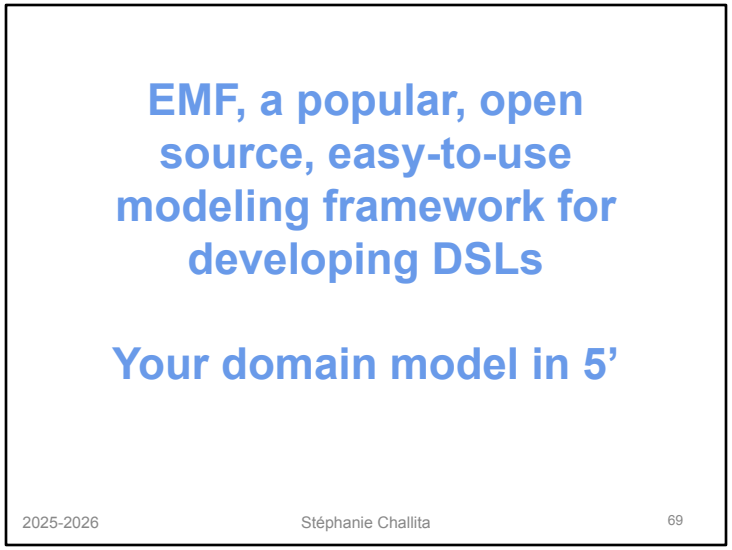


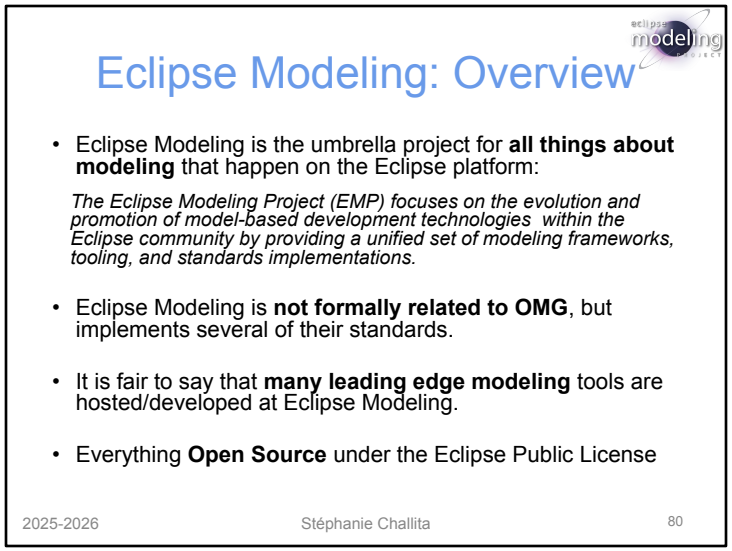
On termine ce panorama des outils avec l’un des plus utilisés et les plus influents dans le monde de la MDE : EMF, pour Eclipse Modeling Framework. EMF, c’est un framework open source, gratuit, robuste, et surtout extrêmement bien intégré à Eclipse.
Il permet de définir un modèle de domaine (métamodèle), de façon déclarative — souvent sous forme de diagrammes ou de classes Java annotées — et de générer automatiquement :

- Du code Java
- Des éditeurs de modèles
- Des API d’accès, de persistance et de notification

Pourquoi EMF est populaire ?

- Il est rapide à prendre en main — en cinq minutes, on peut générer un modèle exploitable
- Il est standard : utilisé dans Xtext, Papyrus, Capella, etc.
- Il sert de colonne vertébrale pour les DSLs outillées, en mode MDE

Donc dans ce cours, on s’appuiera sur EMF pour définir votre premier métamodèle, base de votre propre DSL. Le but : en quelques minutes, avoir votre modèle de domaine opérationnel, prêt à être enrichi avec de la syntaxe, de la sémantique et des outils.



Quand on parle de modélisation dans l'écosystème Eclipse, on fait en réalité référence à un ensemble de projets réunis sous une même bannière : Eclipse Modeling.
C'est ce qu'on appelle un umbrella project — un projet "parapluie" — qui regroupe tous les outils, frameworks, et standards liés à la modélisation au sein d'Eclipse.
L'objectif de l'Eclipse Modeling Project (EMP) est double :

- Favoriser le développement basé sur les modèles
- Offrir une plateforme cohérente pour construire, éditer, transformer, simuler, et déployer des modèles

Parmi les projets hébergés, on retrouve :

- EMF (Eclipse Modeling Framework)
- Xtext pour les DSL textuels
- Sirius pour les éditeurs graphiques
- Papyrus, Acceleo, QVTo, et bien d'autres…

Il est important de noter que même si Eclipse Modeling n'est pas directement lié à l'OMG (l'organisme de standardisation derrière UML, MOF, etc.), il implémente plusieurs de leurs standards.
En pratique, c'est la référence industrielle open source pour tout ce qui touche à la MDE.

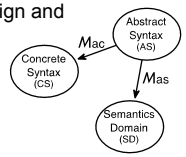
Et surtout : tout est open source, sous la Eclipse Public License — ce qui en fait un écosystème accessible, extensible et durable, idéal pour la recherche, l’industrie, et l’enseignement.

Eclipse Modeling: Overview

The answer to "What is Eclipse Modeling?" depends on who you ask!

A set of Eclipse projects dedicated to…

- **… Modeling**: modeling tools
  - Model Development Tools (UML2, OCL, SysML, MARTE, BPMN2, etc.)
- **… Metamodeling**: workbench for language design and implementation
  - Abstract Syntax Development (EMF)
  - Concrete Syntax Development (GMP, TMF)
  - Model Transformation (M2M, M2T)



- See http://www.eclipse.org/modeling

Cette slide part d'un constat simple mais fondamental : la réponse à la question "Qu'est-ce qu'Eclipse Modeling ?" dépend… à qui on pose la question.
Pour certains, c'est avant tout un ensemble d'outils de modélisation :

- Des outils pour créer des modèles standards :
  - UML2, OCL, SysML, MARTE, BPMN2, etc.
- Ce sont les Model Development Tools : pratiques pour les ingénieurs systèmes, analystes, architectes.

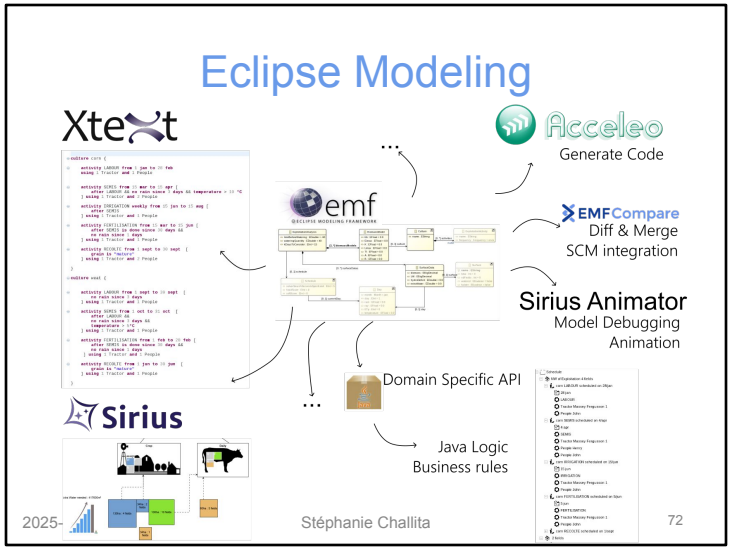
Pour d'autres, Eclipse Modeling, c'est un atelier de conception de langages — un langage workbench :

- Pour faire du métamodélage :
  - EMF pour la syntaxe abstraite
- Pour créer des syntaxes concrètes :
  - GMP (graphique), TMF/Xtext (textuelle)
- Et pour définir des transformations de modèles :
  - M2M (Model-to-Model), M2T (Model-to-Text)

Donc selon votre point de vue, Eclipse Modeling est soit :

- Un ensemble d'outils métier pour manipuler des modèles standards
- Soit une boîte à outils complète pour créer vos propres langages et transformations

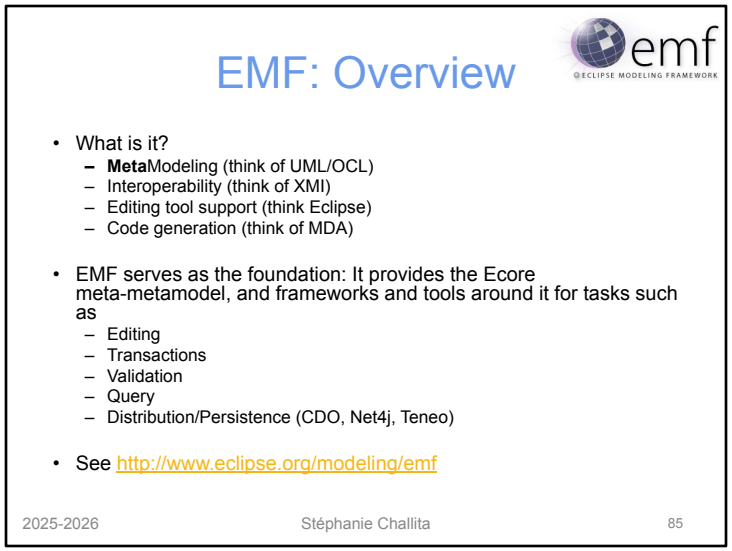
Et si vous voulez plonger dans cet univers, le site eclipse.org/modeling regroupe tous les sous-projets et ressources associées.



Selon votre perspective, Eclipse Modeling peut désigner plusieurs choses :

- Des outils pour **le développement de modèles** : UML2, OCL, SysML, etc.
- Des workbenches pour le **design de langages** : EMF pour la syntaxe abstraite, GMF/Xtext pour la syntaxe concrète.
- Des outils de **transformation de modèles** : M2M (Model-to-Model) et M2T (Model-to-Text).

C’est donc un écosystème très riche et modulaire.



Cette slide résume les fondations et fonctionnalités principales de EMF, l'Eclipse Modeling Framework — probablement l'outil le plus central de tout l'écosystème Eclipse Modeling.

Qu'est-ce que EMF ?

- C'est d'abord un framework de métamodélisation — on peut le voir comme une alternative pratique à UML/OCL pour définir des modèles métiers.
- Il favorise l'interopérabilité, notamment grâce à XMI, un format d'échange standardisé des modèles.
- Il fournit des outils d'édition automatiques, directement intégrés à Eclipse.
- Et surtout, il permet de générer du code Java à partir de votre métamodèle — dans l'esprit de l'approche MDA (Model-Driven Architecture).

Techniquement, EMF repose sur un méta-métamodèle appelé Ecore, qui permet de décrire la structure d'un modèle de manière formelle, comme une sorte de "UML simplifié et exécutable".

Autour d'Ecore, EMF propose toute une suite d'outils et de services pour accompagner le cycle de vie des modèles :

- Editing : génération automatique d'éditeurs de modèles
- Transactions : gestion fine des modifications, avec annulation

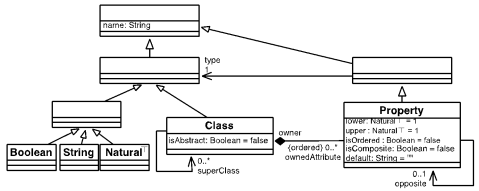
- Validation : règles de cohérence (statiques ou dynamiques)
- Query : interrogation de modèles avec EMF Query
- Persistence/Distribution :
    - CDO (1) pour le versionnement et la collaboration
    - Net4j et Teneo pour la persistance (BDD, Hibernate…)

En résumé, EMF est le socle technique sur lequel reposent de très nombreux outils MDE : Xtext, Sirius, Papyrus…
Si vous travaillez avec des modèles dans Eclipse, vous travaillez presque forcément avec EMF, même sans le savoir.
Plus d'infos ici : eclipse.org/modeling/emf

(1): **CDO = Connected Data Objects**
CDO est un **framework de persistance et de collaboration** pour les modèles EMF. Il permet de **stocker, partager et versionner** des modèles dans un environnement distribué.
En gros, c'est "git" pour EMF.

OMG (Essential) MOF

- Provides language constructs for specifying a DSL metamodel
  - mainly based on Object-Oriented constructs: *package*, *classes*, *properties* (*attribute* and *reference*), and (multiple) *inheritance*.
  - specificities: composition, opposite…
- Defined as a model, called *meta*metamodel:



Ici, on entre dans le cœur d'EMF : la manière dont on spécifie un métamodèle, c'est-à-dire la structure d'un langage de modélisation.
EMF fournit un ensemble de constructs orientés objet pour décrire ce métamodèle. On y retrouve des éléments très familiers pour les développeurs Java ou UML :
Les éléments principaux :

- Package : un conteneur logique (comme un namespace)
- Classes : les entités principales de votre modèle
- Propriétés :
  - Attributs (pour les valeurs simples : `String`, `int`, etc.)
  - Références (pour relier des objets entre eux)

On peut aussi gérer :

- L'héritage multiple
- La composition (propriétés "contenantes")
- Les références opposées (navigabilité bidirectionnelle)

Tous ces éléments sont eux-mêmes définis dans un modèle, qu'on appelle un métamétamodèle — en l'occurrence, Ecore dans EMF.

Cela signifie que le langage qui sert à définir des langages (comme EMF) est lui-même un modèle structuré.
En résumé :

- Avec EMF, on peut décrire un langage métier sous forme de classes et relations
- Ce métamodèle est interprété et exploité automatiquement : pour générer du code, construire des éditeurs, stocker des modèles, etc.

Ecore: a metamodel for metamodels

- Ecore is an implementation proposed by EMF, and aligned to EMOF
- Provides a language to build languages
- A metamodel is a model; and its metamodel is Ecore.
  - So a metamodel is an Ecore model!
- Ecore has concepts like:
  - Class – inheritance, have properties
  - Property – name, multiplicity, type
- Essentially this is a simplified version of class modeling in UML

Ici, on revient plus en détail sur Ecore, qui est la pierre angulaire du métamodélage dans EMF.

Ecore est une implémentation de EMOF (Essential MOF), proposée par EMF :

- EMOF, c'est un standard de l'OMG (Object Management Group)
- Et Ecore en est une version simplifiée, allégée et pratique, très utilisée dans l'écosystème Eclipse

Ecore est un langage pour construire des langages :

- C'est ce qu'on appelle un métamétamodèle
  Il sert à décrire des métamodèles, qui eux-mêmes décrivent des modèles métier

En d'autres termes :

Un métamodèle EMF est un modèle conforme à Ecore
Autrement dit : tout métamodèle est un modèle Ecore

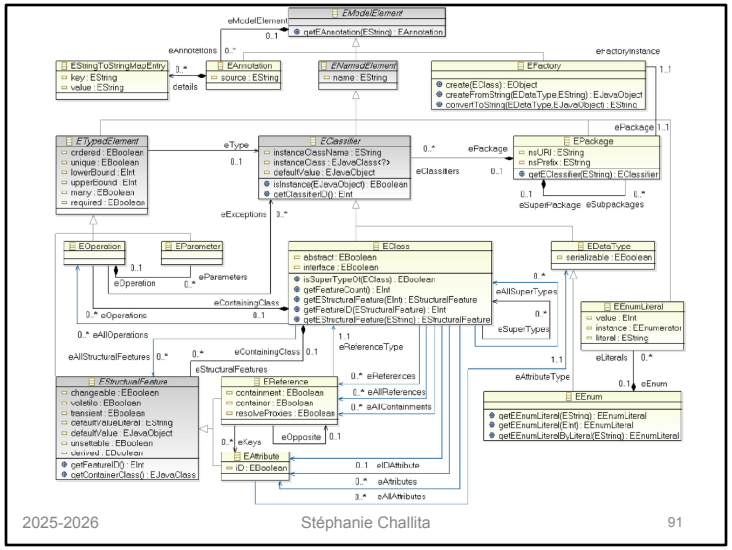
Les concepts fondamentaux d'Ecore sont inspirés de la modélisation objet :

- EClass : représente une classe, avec héritage, attributs, références…
- EProperty : propriété d'une classe, avec un nom, une multiplicité, un type

Et tout cela est suffisant pour modéliser la plupart des concepts métier, tout en étant léger et bien outillé.

En résumé :

- Ecore simplifie UML, tout en restant expressif
- C’est une base unifiée pour construire des métamodèles, des éditeurs, et des DSLs
- C’est le cœur technique d’EMF et de tout l’écosystème Eclipse Modeling



Cette slide montre le métamodèle d'Ecore lui-même — autrement dit, la structure formelle utilisée pour décrire tous les métamodèles dans EMF.
Ce qu'on voit ici, c'est que tout métamodèle EMF repose sur quelques concepts fondamentaux, définis dans un modèle Ecore… qui est lui-même un modèle !
Les éléments principaux :

- EPackage :
  C'est un conteneur logique. Il regroupe les classes de votre langage, un peu comme un package Java ou un namespace.
- EClass :
  C'est le cœur du système — chaque classe représente un concept métier (ex : "Personne", "Voiture", "Événement"). Elle peut avoir des superclasses, des propriétés, et un nom.
- EAttribute :
  Propriété simple d'une classe (ex : `name: String`, `age: int`).
- EReference :
  Lien vers une autre classe (ex : `owner → Person`). Peut être multiple, optionnel, bidirectionnel, etc.
- EDataType :

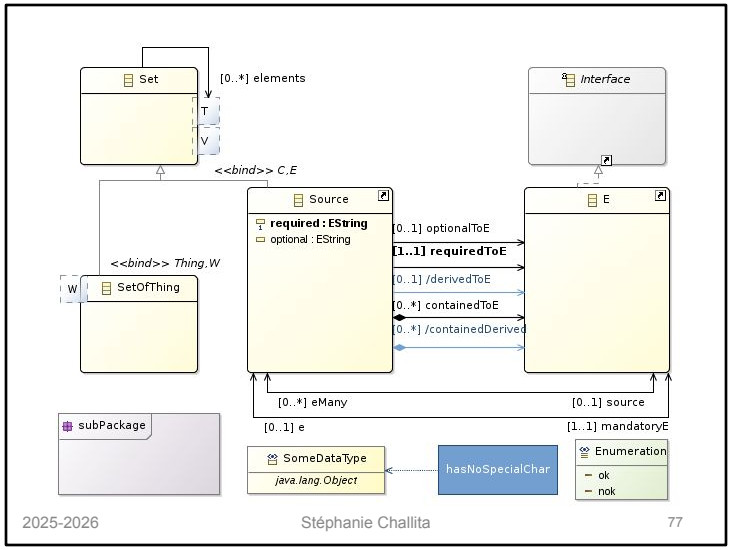
- Représente les types primitifs (`int`, `String`, `Date`…), ou vos types personnalisés.

Ce métamodèle est auto-descriptif : il permet de décrire lui-même des structures de données, et d'être utilisé par les outils EMF pour générer du code, créer des éditeurs, faire de la validation, etc.

C'est ce qu'on appelle une hiérarchie de méta-niveaux :

- M3 : Ecore (le métamétamodèle)
- M2 : votre métamodèle métier
- M1 : votre modèle (vos objets)
- M0 : l'exécution, les instances réelles

En résumé : tout modèle EMF est une instance de ce métamodèle Ecore.
Et c'est grâce à cette structure uniforme qu'on peut outiller facilement les DSLs, générer du code ou synchroniser des vues.



Voici maintenant un exemple concret de métamodèle défini avec Ecore, qui illustre comment on peut structurer un langage spécifique au domaine.
On retrouve ici plusieurs éléments clés du formalisme Ecore :

- Des classes comme `Source`, `E`, `Set`, `SetOfThing`, etc.
- Des attributs (`required : EString`, `optional : EString`)
- Des références entre classes (`requiredToE`, `containedToE`, etc.), avec des multiplicités, des opposées et même des dérivées
- Une énumération (`Enumeration`) avec deux valeurs possibles : `ok`, `nok`
- Un exemple de contrainte OCL-like : `hasNoSpecialChar`, on détaillera OCL plus tard.

Ce modèle montre également des concepts plus avancés :

- L'utilisation de types paramétriques (`<<bind>>`), qui rappelle la généricité
- Des interfaces (comme la classe `Interface`) pour la factorisation ou l'extension de comportements
- La hiérarchie de classes (`SetOfThing` hérite de `Set`)

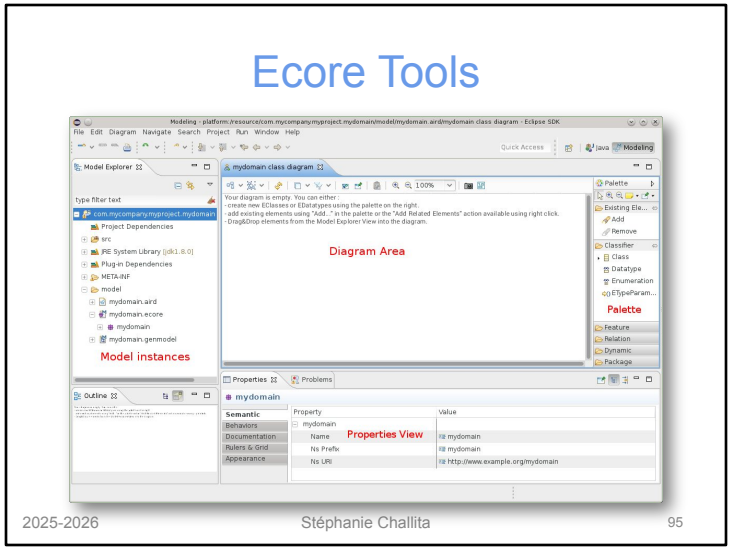
Ce métamodèle peut être utilisé pour :

- Valider des données (ex : vérifier qu'un champ ne contient pas de caractères spéciaux)
- Décrire une transformation (ex : relier `Source` à un ou plusieurs éléments `E`)

- Ou générer du code permettant de manipuler ces structures dans une application

En résumé :

- Ce modèle exploite toute la puissance d'Ecore : héritage, références, contraintes, types composés
- Il peut servir de base à un DSL métier, que l'on pourra ensuite outiller avec EMF, Xtext ou Sirius



Cette slide montre une vue d'ensemble de l'environnement de modélisation dans Eclipse, en perspective "Modeling".
C'est typiquement ce que vous verrez lorsque vous utilisez Sirius ou un éditeur EMF graphique personnalisé.
L'interface est divisée en plusieurs zones clés, chacune jouant un rôle spécifique dans l'expérience de modélisation :
À gauche : Model Instances
C'est ici qu'on retrouve l'arbre du modèle, c'est-à-dire les instances créées à partir de votre métamodèle.
Chaque nœud correspond à un objet (par exemple une entité, une transition, un état…), organisé hiérarchiquement.
C'est souvent là qu'on crée ou supprime des éléments.
Au centre : Diagram Area
Il s'agit de la zone de modélisation graphique.
C'est ici que vous visualisez et manipulez les objets sous forme de diagramme : vous les positionnez, les reliez, les modifiez visuellement.
Cette vue est synchronisée avec le modèle sous-jacent.
À droite : Palette

La palette contient les outils de création : différents types de nœuds ou de relations que vous pouvez insérer dans le diagramme.

Elle est entièrement configurable selon le DSL que vous utilisez.

En bas : Properties View

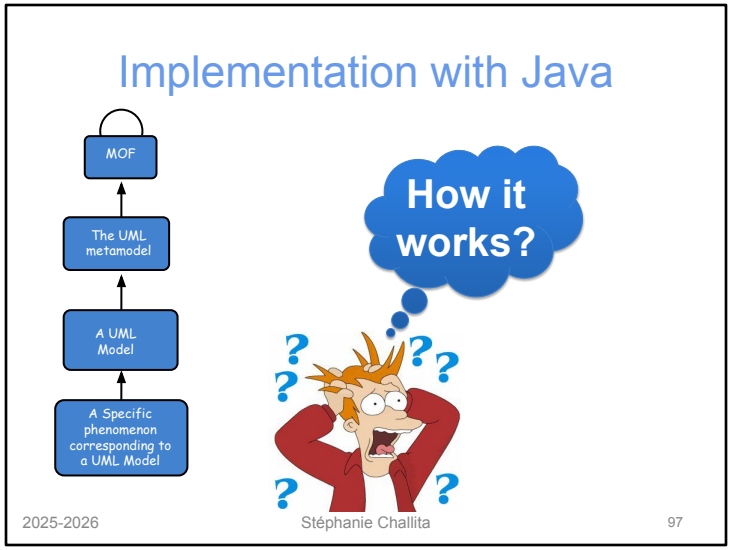
Cette vue affiche et permet de modifier les propriétés de l'élément sélectionné.

C'est l'équivalent d'un formulaire d'édition : vous pouvez y renseigner des noms, des types, des contraintes, etc.

En résumé, cette interface offre une expérience de modélisation interactive et cohérente, qui relie de manière fluide :

- la structure du modèle (à gauche),
- sa représentation graphique (au centre),
- les outils de construction (à droite),
- et les détails des objets (en bas).

C'est ce type de configuration que vous utiliserez tout au long du cours pour manipuler vos propres langages.

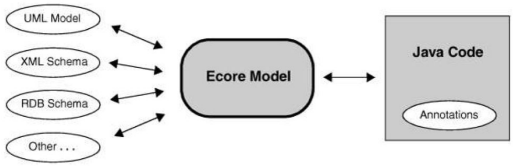


1. (Première boîte, tout en bas) – "A specific phenomenon corresponding to a UML model"
On commence ici, tout en bas, avec quelque chose de très concret :
une situation réelle, un phénomène du monde, un comportement, un objet…
C'est ce qu'on veut modéliser. Par exemple : une application qui gère des utilisateurs et des rôles.
2. (Deuxième boîte) – "A UML model"
Ce phénomène, on le représente par un modèle UML :
un diagramme de classes, une structure, des relations.
On est maintenant dans le monde de la modélisation, mais à un niveau encore spécifique (niveau M1).
3. (Troisième boîte) – "The UML metamodel"
Mais UML lui-même, ce n'est pas de la magie : ce langage a été défini formellement.
Ce que vous voyez ici, c'est le métamodèle UML — un modèle qui décrit ce qu'est une "classe", un "attribut", une "relation".
On est au niveau M2.
4. (Quatrième boîte) – "MOF"
Et enfin, tout en haut de cette hiérarchie, on trouve MOF, le métamétamodèle.
C'est le langage qui permet de définir… des langages.

UML est une instance de MOF, comme Ecore dans EMF.
(Apparition du “How it works???”)
Et à ce stade, on comprend si vous vous sentez un peu comme ça...
Oui, ça peut sembler abstrait, mais cette hiérarchie est ce qui permet à l’ingénierie logicielle moderne d’être modulaire, générative et vérifiable.
Cette chaîne du concret vers l’abstrait — du phénomène réel jusqu’au métalangage — est la base de l’approche model-driven.
Et on va justement apprendre à manipuler chacun de ces niveaux, jusqu’à produire votre propre langage exécutable.

EMF

An Ecore model and its sources
(from *EMF: Eclipse Modeling Framework 2nd*)



Pour expliquer tout ceci, nous allons utiliser le rôle central d'Ecore au niveau M2.
Ecore est un métamétamodèle exécutable, et donc une implémentation concrète du niveau M3.
Il permet de définir vos propres métamodèles (M2), et sert ensuite de point de jonction entre des sources hétérogènes et la génération de code.
On voit ici que le Ecore model peut être obtenu à partir de plusieurs types de sources :

- Un modèle UML existant (niveau M1) peut être converti en Ecore
- Un schéma XML (XSD) ou relationnel (RDB) aussi
- Même des formats spécifiques (fichiers, schémas personnalisés…)

Autrement dit, on peut importer un modèle existant dans EMF, et le transformer en un métamodèle Ecore.
C'est un passage du niveau M1 au niveau M2, sous une forme exploitable par les outils Eclipse.
Et de l'autre côté, à partir de ce modèle Ecore, EMF est capable de générer automatiquement du code Java :
ce qui correspond cette fois à un passage du niveau M2 vers M0, via M1 si on le considère comme un modèle de structure logicielle.
Ce code représente :

- Des classes Java
- Des interfaces

- Des annotations
- Et même un éditeur de modèles

Ce slide montre que le modèle Ecore est un point de convergence entre des formats variés, des niveaux d’abstraction différents, et une chaîne d’outillage cohérente.

C’est à partir de lui qu’on va pouvoir construire, manipuler, outiller et exécuter nos DSLs.

Implementation with Java

- EMF is a software (E)framework
- Model driven…, but implemented using a programming language!
- Reification MDE → Java:
  - Metamodels are represented with EClasses
  - Models are represented with EObjects

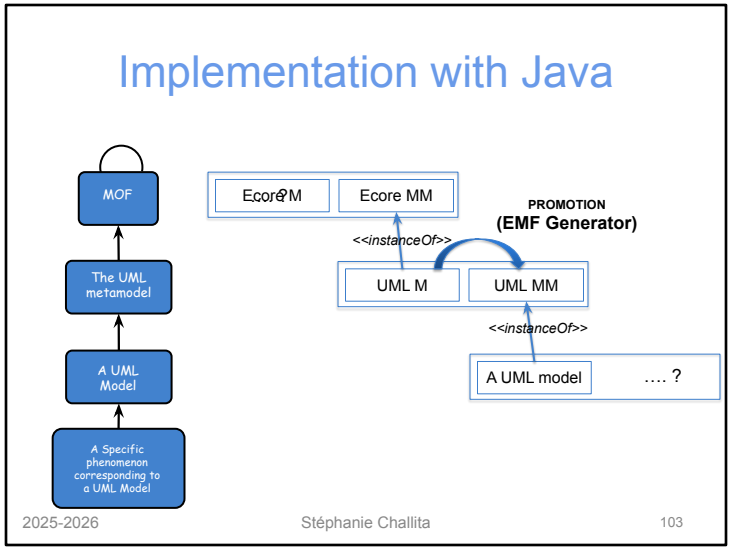
On a vu jusqu'à présent EMF comme un framework de modélisation, un outil conceptuel pour manipuler des métamodèles, des modèles, et des données.
Mais EMF est aussi un framework logiciel concret — une implémentation Java.
Autrement dit : EMF est "model-driven", mais il fonctionne grâce à du code Java.
Il y a donc une correspondance explicite entre les concepts abstraits de la MDE, et leur représentation concrète dans le code.
C'est ce qu'on appelle la reification, ou la concrétisation des concepts MDE dans un langage de programmation.
Voici comment cette correspondance se manifeste :

- Les métamodèles (niveau M2) sont représentés par des objets de type `EClass` dans EMF.
  Chaque `EClass` représente une classe de votre métamodèle — avec ses attributs, références, superclasses, etc.
- Les modèles (niveau M1) sont représentés par des instances de `EObject`.
  Ce sont les objets réels de votre modèle, créés dynamiquement en mémoire.

Ces deux types (`EClass`, `EObject`) sont fournis par EMF. Ce sont les briques de base sur lesquelles repose l'exécution des modèles.
Ce mécanisme permet à EMF de :

- Charger dynamiquement des modèles
- Manipuler leur structure à l’exécution
- Générer automatiquement du code, des vues, des éditeurs…

Même si l’on reste dans une logique "model-driven", tout est finalement implémenté avec des objets Java.
Et c’est précisément cette réification qui permet à EMF de faire le lien entre les abstractions MDE et les outils logiciels concrets.



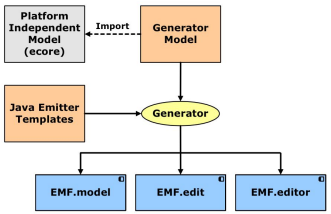
(Début : schéma de gauche uniquement affiché)
Commençons avec ce qu'on connaît : la hiérarchie des niveaux d'abstraction.
Tout en bas, on a une situation concrète — un système, un comportement réel.
On en fait un modèle UML (niveau M1), conforme à un métamodèle UML (niveau M2),
lui-même défini selon MOF, au niveau M3.
C'est la vision théorique, conforme aux standards de l'OMG.
Voyons maintenant comment EMF permet de mettre ça en œuvre avec Java.
(Apparition de "Ecore MM")
Voici Ecore MM : c'est le métamodèle d'EMF.
Il correspond au niveau M3, comme MOF, mais sous forme exécutable.
C'est ce qui permet à EMF de définir et manipuler des métamodèles de manière opérationnelle.
(Apparition de "UML M", avec flèche <<instanceOf>> vers Ecore MM)
Ici, on a un modèle Ecore correspondant à UML, que l'on appelle UML M.
Il est une instance du métamodèle Ecore — autrement dit, un modèle qui utilise les concepts d'Ecore (comme EClass, EAttribute, etc.).
Cette relation "instanceOf" correspond à ce qu'on a déjà vu : c'est la relation M2 → M3.

(Apparition de “UML MM”, flèche de promotion, double cadre autour de UML M et UML MM)
Mais EMF ne s’arrête pas là.
Grâce à son générateur, il peut promouvoir ce modèle Ecore UML
en un métamodèle exploitable, ici appelé UML MM.
On a donc construit un métamodèle UML à partir d’un modèle Ecore,
et ces deux-là forment désormais une paire métamodèle–modèle.
(Apparition de “A UML Model” avec flèche <<instanceOf>> vers UML MM)
On peut alors instancier ce métamodèle UML MM :
on obtient un modèle UML concret, comme n’importe quel diagramme UML que vous pourriez créer avec Papyrus, Modelio, ou dans Eclipse.
(Apparition de “... ?” à droite du UML Model)
Mais là, une question se pose :
Si je peux créer un modèle UML à partir d’un métamodèle UML...
et que ce métamodèle lui-même a été généré à partir d’un modèle Ecore...
alors... est-ce que je viens de créer un nouveau langage UML ?
C’est exactement ça.
À partir d’un modèle, j’ai généré un métamodèle,
et donc un nouveau langage conforme à mon modèle d’UML.
(Apparition de “... ?” à gauche de Ecore MM)
Et si on remonte encore plus haut, on peut se poser la question :
Mais ce métamodèle Ecore lui-même, il vient d’où ?
(Puis apparition finale de “Ecore M”)
La réponse est logique : il vient de Ecore M — c’est le modèle Ecore
qui a permis de définir le métamodèle Ecore MM.
Autrement dit, Ecore s’auto-décrit.
Conclusion

Ce slide montre que toute la hiérarchie des niveaux M0 à M3 peut être reproduite concrètement dans EMF.
Et grâce au générateur EMF, on peut même passer d'un modèle à un métamodèle,
et créer son propre langage à partir d'un modèle.
C'est ça, la puissance de la modélisation exécutée en Java avec EMF.

EMF Toolset

- The EMF Generator do not work on the .ecore
- EMF defines a .genmodel in parallel:
  - We can customize the code generator!
  - The IDE takes care of maintaining the consistency (or not!)



From "Mastering Eclipse Modeling Framework", V. Bacvanski and P. Graff

Maintenant qu'on a vu comment EMF structure les niveaux de modélisation et permet de générer du code à partir de modèles, regardons comment cela se met concrètement en place dans l'outil.
Et c'est là qu'intervient un élément fondamental du toolset EMF : le `.genmodel`.
Première chose à savoir :

Le générateur EMF ne travaille pas directement sur le fichier `.ecore`.

C'est souvent une source de confusion : on pourrait penser que `.ecore` suffit pour générer le code, mais ce n'est pas le cas.
EMF introduit un fichier complémentaire : le `.genmodel`, qui est dérivé du `.ecore`.
Ce fichier contient :

- Des informations de configuration pour le générateur
- Des chemins de package
- Des options spécifiques (générer un éditeur ? Support XML ? etc.)
- Des personnalisations possibles du comportement de génération

Un point important ici :

Le `.genmodel` permet de personnaliser le code généré, sans toucher au modèle conceptuel `.ecore`.

On peut donc adapter l'implémentation Java à ses besoins métiers, sans casser le métamodèle.

Enfin, dernier point à retenir :

L'IDE tente de maintenir la cohérence entre `.ecore` et `.genmodel`,

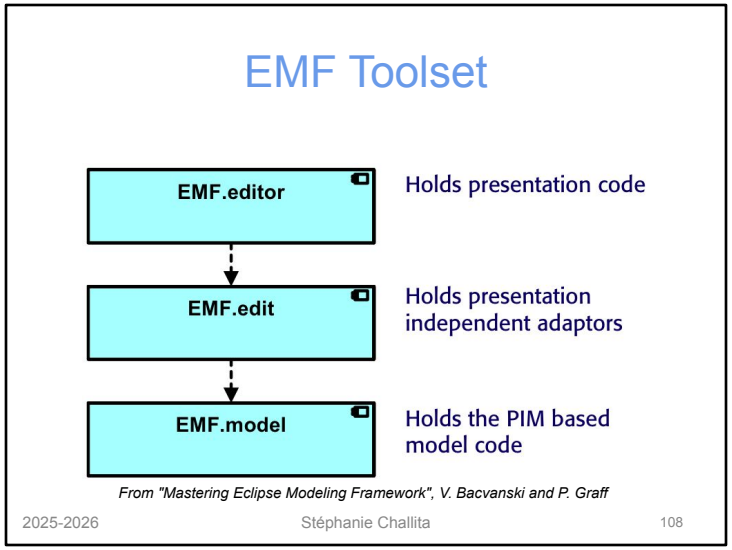
mais ce n'est pas automatique à 100 %.

Il faut parfois re-synchroniser manuellement ou régénérer les artefacts si l'un a été modifié.

Le `.ecore` décrit la structure du langage.

Le `.genmodel` décrit comment on veut générer du code à partir de cette structure.

Et l'ensemble de l'outillage EMF repose sur cette séparation.



Ici, on a l'architecture modulaire générée automatiquement par EMF, avec une séparation claire des responsabilités.
On retrouve ici les trois couches principales qu'on a déjà mentionnées, mais cette fois dans leur relation hiérarchique et logique.
En bas : EMF.model
C'est la couche de base, celle qui contient le code métier du modèle.

- C'est ici qu'on retrouve les `EClass`, `EObject`, les attributs, références, etc.
- On parle parfois de PIM — Platform-Independent Model — car ce code est centré sur le domaine, pas sur l'interface.

Il s'agit vraiment du noyau structurel du langage.
Au milieu : EMF.edit
Cette couche joue le rôle de pont entre le modèle et la présentation.

- Elle contient des adaptateurs (notamment les `ItemProviderAdapter`), qui décrivent comment représenter chaque élément du modèle dans une interface graphique.
- Elle est indépendante de la présentation concrète, ce qui la rend réutilisable dans différents contextes (UI Eclipse, web, etc.).

En haut : EMF.editor

C’est le plugin Eclipse complet qui permet de visualiser, éditer, sauvegarder et manipuler des instances du modèle.

- Il est basé sur les couches inférieures (edit et model)
- Il contient le code de l’éditeur : arborescence, formulaires, menus contextuels…

C’est ce qu’on appelle la présentation concrète (UI).

Lien entre les trois couches

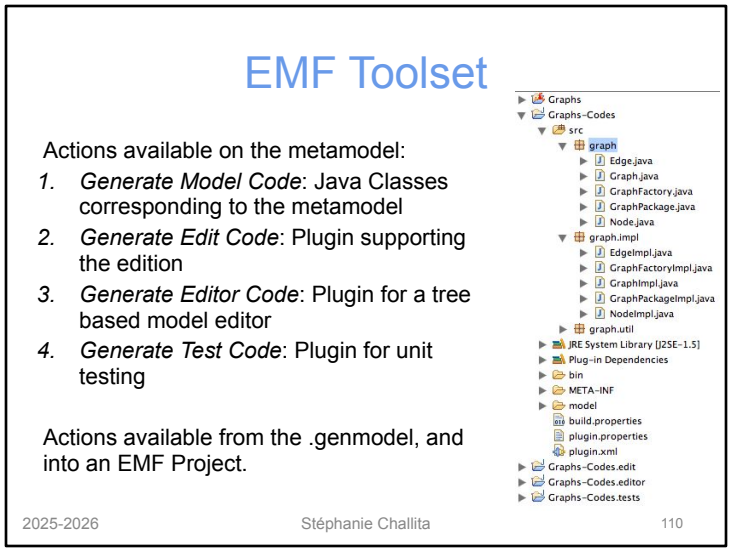
- `EMF.editor` dépend de `EMF.edit`,
- `EMF.edit` dépend de `EMF.model`,

  mais chacune peut être utilisée ou omise en fonction des besoins du projet.

Par exemple :

- Un back-end de transformation a juste besoin du model
- Une interface personnalisée pourrait réutiliser les adapters de edit sans l’éditeur Eclipse

Ce schéma permet de bien comprendre que le générateur EMF ne produit pas un seul bloc de code,
mais une architecture propre et modulaire, qu’on peut exploiter partiellement ou totalement selon les besoins du projet.



Quand on travaille avec EMF, une étape essentielle est la génération de code à partir du métamodèle.
Mais ce n'est pas une opération unique : EMF propose en réalité plusieurs générateurs ciblés, chacun avec un rôle précis.
Tous ces générateurs sont accessibles à partir du `.genmodel`, directement dans l'environnement Eclipse.
Et chacun correspond à une fonctionnalité bien précise du projet EMF.
Voyons-les un à un :

1. Generate Model Code

C'est l'action la plus basique :

Elle génère les classes Java correspondant à vos `EClass`, `EAttribute`, `EReference`, etc.

C'est le noyau du langage, ce qu'on appelle souvent EMF.model.
Ce code contient la structure logique du domaine (le "modèle métier").

2. Generate Edit Code

Cette action génère un plugin d'adaptation, qu'on appelle EMF.edit.
Il contient des ItemProviders et d'autres classes qui servent à connecter le modèle à une interface utilisateur, tout en restant indépendant de la technologie de rendu (SWT, Web, etc.).

3. Generate Editor Code

Cette fois, EMF génère un éditeur complet pour Eclipse, souvent basé sur une arborescence interactive.

C'est le plugin EMF.editor qu'on a vu précédemment.

Il permet à l'utilisateur final de créer, modifier, sauvegarder et charger des modèles instanciés.

1. Generate Test Code

Moins connue, cette option permet de générer automatiquement un projet de tests unitaires.

Il s'appuie sur JUnit, et contient des squelettes de tests pour vérifier le bon fonctionnement du modèle ou des contraintes personnalisées.

Tous ces générateurs sont accessibles depuis le `.genmodel`

Le `.genmodel` est donc le point d'entrée principal pour la génération de code dans EMF.

C'est là que vous choisissez quoi générer, où, et comment,

et que vous pouvez personnaliser ou régénérer en cas d'évolution du modèle.

Conclusion

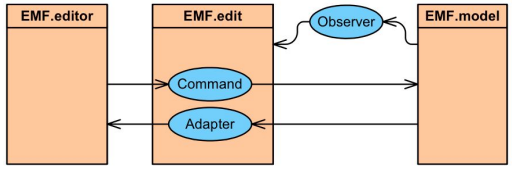
Le `.genmodel`, c'est votre centre de contrôle :

il vous permet de déclencher la génération des différents modules, en fonction de vos besoins (modèle, UI, tests…).

Et tout ça se fait directement depuis Eclipse, souvent en un clic droit → Generate…

EMF: open the box

- The EMF.edit separates the GUI from the business model
- To understand the EMF.edit plug-in, it is essential to understand three basic design patterns
  - Observer pattern
  - Command pattern
  - Adapter pattern



From "Mastering Eclipse Modeling Framework", V. Bacvanski and P. Graff

Jusqu'à maintenant, on a vu EMF comme un outil de génération de code structuré.
Mais que se passe-t-il quand on "ouvre la boîte" ?
C'est-à-dire, quand on veut comprendre comment l'infrastructure EMF fonctionne en interne ?
Et en particulier, comment EMF.edit fait le lien entre l'éditeur graphique et le modèle métier ?
C'est ce que montre cette slide :
EMF.edit joue un rôle d'intermédiaire essentiel, en isolant la présentation (EMF.editor) de la logique métier (EMF.model).
Pour comprendre comment EMF.edit opère, il faut connaître trois design patterns fondamentaux, tous utilisés de manière centrale dans sa conception :

Observer pattern

C'est le pattern qui permet de notifier automatiquement les vues lorsqu'un objet du modèle change.
Quand une donnée dans le `EMF.model` est modifiée,
EMF.edit transmet l'événement à l'éditeur via le pattern observateur.
C'est ce qui permet à l'interface graphique de se mettre à jour automatiquement.

Adapter pattern

Ce pattern est au cœur de EMF.edit.

Les fameux `ItemProvider` sont des adaptateurs :
ils traduisent les objets métier en éléments éditables et présentables dans l'éditeur.
Par exemple, ils définissent les noms à afficher, les icônes, les propriétés modifiables, etc.

Command pattern

EMF.editor n'interagit jamais directement avec le modèle.
Quand on modifie quelque chose (ajout, suppression, renommage…),
EMF.editor émet une commande, qui est prise en charge par EMF.edit.
Cette commande encapsule l'action, la rend undoable/redoneable, et centralise la logique métier.
En résumé :

- `EMF.model` contient les données
- `EMF.editor` contient l'interface
- `EMF.edit` assure la coordination entre les deux

Et cette coordination repose sur trois patterns bien connus :
Observer, Adapter et Command.

Conclusion

Comprendre ces trois patterns est indispensable pour étendre ou personnaliser EMF.edit,
car ils expliquent toute la dynamique interne entre les plugins.

EOperation Implementation

Localization of the methods in the generated code

1. In the subpackage `graph.impl`
2. In the class `GraphImpl`
3. Scattered in the code automatically generated by EMF…

```
/**
 * @generated NOT
 */
public int order () {
 return this.getEdges().size();
}
```

Do not forget to mark (`@generated NOT`) to prevent crushing!

Quand on travaille avec EMF, on utilise le générateur pour produire automatiquement une grande partie du code Java.
Mais parfois, on a besoin d'ajouter des comportements métier spécifiques — des méthodes personnalisées — directement dans ce code généré.
Où les écrire ?
Par convention, EMF place l'implémentation des classes dans un sous-paquet appelé `.impl`.
Par exemple, pour un modèle nommé `Graph`, la classe générée s'appellera `GraphImpl`,
et elle contiendra la logique par défaut.
C'est dans cette classe que vous pouvez insérer votre propre code métier.
Oui, mais attention : le générateur EMF écrase les fichiers lors de la régénération.
Donc, si vous ajoutez une méthode manuellement…
et que vous régénérez sans précaution…
votre méthode disparaît.
Pour éviter ça, EMF propose une convention simple :

Ajoutez `@generated NOT` au-dessus de toute méthode que vous avez écrite vous-même.

Cela signale au générateur EMF de ne pas toucher à cette méthode lors de la prochaine génération.

Ici, on a un exemple simple :
Ici, on ajoute une méthode `order()` pour calculer dynamiquement le degré d'un graphe.
Elle n'était pas prévue dans le `.ecore`, donc elle doit être manuellement codée
et protégée avec `@generated NOT`.
Conclusion
Quand vous travaillez avec du code généré EMF :

- Ajoutez vos méthodes dans les classes `Impl`
- Marquez-les avec `@generated NOT`
- Et vérifiez bien après chaque génération que tout est encore en place

C'est une bonne pratique indispensable pour garder le contrôle sur votre logique métier.