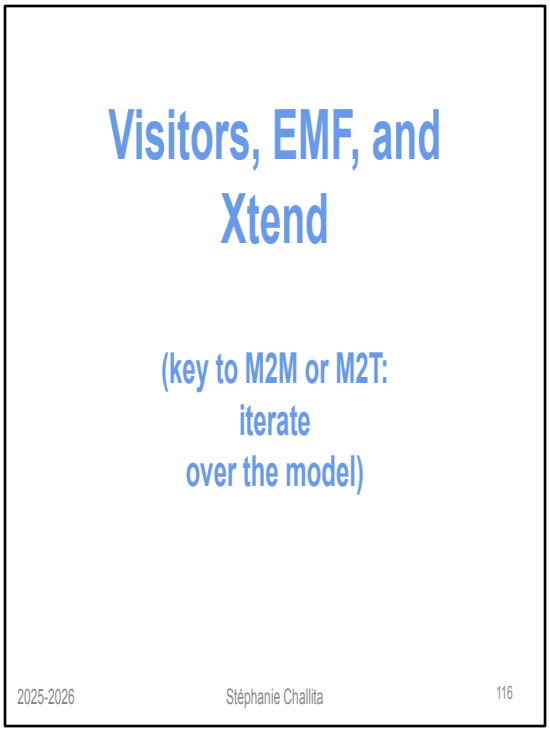


On va démarrer une nouvelle section, où l'on va aborder un concept central dans le cadre des transformations de modèles (M2M) ou de génération de code (M2T) : le parcours de modèles.
Que ce soit pour :

- transformer un modèle en un autre (Model-to-Model),
- ou pour générer du code source (Model-to-Text),

la logique reste la même :

Parcourir le modèle, élément par élément,
et exécuter une action adaptée à chaque type de nœud.

Ce principe est fortement lié au pattern "Visitor",
que l'on retrouve en programmation orientée objet pour séparer les algorithmes de la structure de données.
Dans notre cas :

- Le modèle EMF est une arborescence d'objets
- Chaque objet représente un élément du langage : classe, attribut, opération, etc.
- Et on souhaite appliquer une logique différente selon le type d'élément

Avec Xtend, ce type de parcours est extrêmement expressif :

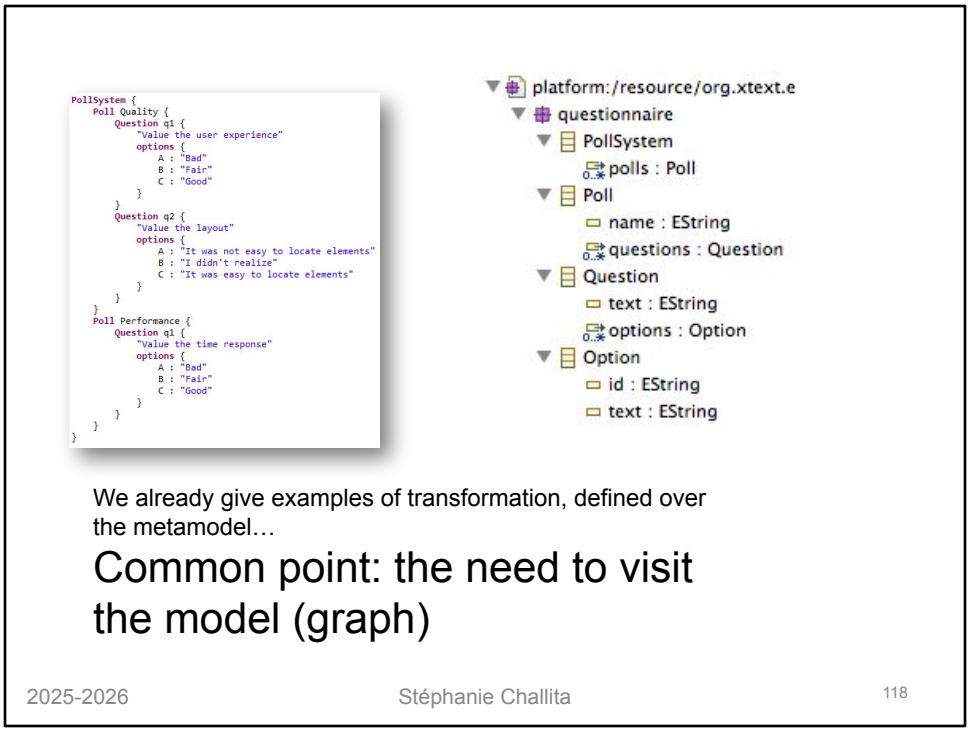
- Grâce aux méthodes dispatch, on peut définir un comportement par type de nœud
- Et ainsi construire des visiteurs propres, typés et lisibles

Conclusion de la transition :

Ce que vous allez voir dans cette section, c’est comment combiner :

- EMF pour la structure du modèle
- Xtend pour le code de transformation
- Et le pattern visitor pour itérer élégamment sur le modèle

C’est la clé pratique pour écrire des transformations puissantes, maintenables et élégantes.



Jusqu'ici, on a déjà vu plusieurs exemples de transformations — que ce soit pour :

- Générer du code HTML à partir d'un modèle de sondage (`PollSystem`),
- Produire une interface avec `@Extract`,
- Ou encore pour transformer dynamiquement la structure d'une classe via une annotation.

Ces transformations ont toutes un point commun fondamental :

Elles nécessitent de parcourir le modèle, souvent sous forme de graphe d'objets EMF.

À gauche, on voit un fichier `.q` – une spécification textuelle de sondage dans notre DSL.

Et à droite, sa représentation sous forme de modèle EMF dans Eclipse, en mode "Modeling Perspective".

Ce que cela montre, c'est le lien direct entre :

- Le texte source du langage
- Et sa structure objet conforme au métamodèle (généré automatiquement par Xtext)

Pour écrire une transformation, on ne travaille pas directement sur le texte — on s'appuie sur ce modèle structuré en mémoire.

Et pour agir efficacement sur ce modèle, il faut visiter ses éléments un par un :

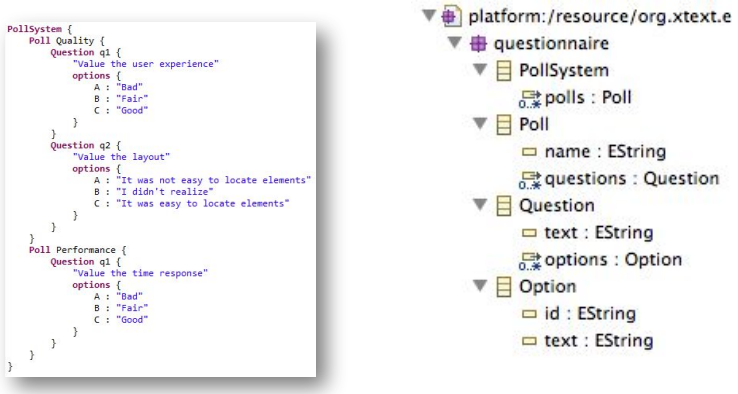
- un `PollSystem` contient des `Polls`,
- chaque `Poll` contient des `Questions`,

- chaque `Question` contient des `Options`…

Ce parcours récursif, hiérarchique, est au cœur de toutes les transformations — qu'elles soient M2M ou M2T.

La suite du cours va donc se concentrer sur :

- Les outils pour visiter un modèle EMF proprement
- Et comment Xtend facilite ce type de traitement avec un code léger, lisible et typé



Visit the model (graph)
Possible solution: a series of casts (lots of if-statements and traversal loops)

On vient de voir que pour transformer un modèle — que ce soit pour générer du texte ou un autre modèle —
il faut parcourir la structure du modèle, c'est-à-dire visiter le graphe d'objets EMF.
Mais… comment fait-on concrètement ?
Une solution naïve, que l'on retrouve souvent au début, c'est d'utiliser :

une série de tests de type (`instanceof`), des casts explicites, et des boucles d'itération manuelles.

En Java par exemple, cela donnerait :

```
if (e instanceof PollSystem) {
  PollSystem ps = (PollSystem) e;
  for (Poll p : ps.getPolls()) {
    // etc.
  }
}
```

Ou pire :

- Des `switch` avec des `EClass.getName()`
- Des structures conditionnelles profondément imbriquées
- Beaucoup de répétition et peu de robustesse

Le problème avec cette approche, c'est qu'elle devient vite :

- Verbeuse : beaucoup de code pour peu de logique réelle
- Fragile : difficile à maintenir si le métamodèle évolue

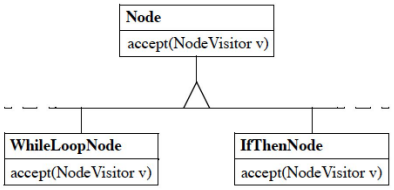
- Peu modulaire : pas de séparation claire entre structure et comportement

Il existe des solutions bien plus élégantes.
Et c’est là que des outils comme Xtend, ou des design patterns comme le visitor, entrent en jeu.
La suite va justement vous montrer comment écrire des visiteurs de modèle efficaces, sans `instanceof`, sans cast explicite, et avec une syntax très fluide et typée.
C’est une bonne pratique essentielle dans toute approche MDE/M2T moderne.

Visitor Pattern

separating an algorithm from an object structure on which it operates

```
public class WhileLoopNode extends Node {
  protected Node condition, body;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitWhileLoop(this);
  }
}
public class IfThenNode extends Node {
  protected Node condition, thenBranch;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitIfThen(this);
  }
}
```




```
public abstract class NodeVisitor {
  /* ... */
  public abstract void visitWhileLoop(WhileLoopNode n);
  public abstract void visitIfThen(IfThenNode n);
}
public class TypeCheckingVisitor extends NodeVisitor {
  /* ... */
  public void visitWhileLoop(WhileLoopNode n) { n.getCondition().accept(this); /* ... */ }
  public void visitIfThen(IfThenNode n) { /* ... */ }
}
```

new operations can be added modularly, without needing to edit any of the Node subclasses: the programmer simply defines a new NodeVisitor subclass containing methods for visiting each class in the Node hierarchy.

Nous arrivons maintenant à une solution classique et puissante pour séparer un algorithme de la structure de données sur laquelle il opère : le *Visitor Pattern*, ou patron de visiteur.

L'idée est simple :

On a une hiérarchie d'objets — ici des nœuds syntaxiques (WhileLoopNode, IfThenNode, etc.) —

et on veut implémenter un traitement sur ces objets sans modifier leurs classes.

Regardons l'exemple :

Chaque nœud hérite de Node, et redéfinit une méthode accept(NodeVisitor v)

Cette méthode délègue le contrôle à un objet visiteur qui sait quoi faire avec chaque type de nœud.

Ensuite, on a une classe abstraite NodeVisitor qui définit une méthode visitX pour chaque type de nœud :

Et une implémentation concrète comme TypeCheckingVisitor peut définir exactement ce qu'elle fait pour chaque nœud.

L'avantage ?

On peut ajouter de nouveaux comportements (visiteurs) sans modifier les classes de nœud :

- Type checking
- Interprétation
- Traduction vers du code

- Génération de documentation
- Etc.

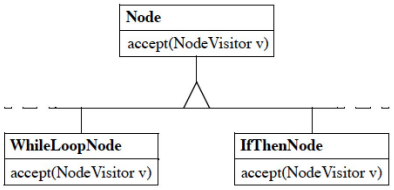
En résumé :

✅ Ajout de comportements = simple, modulaire

C’est une solution robuste dans beaucoup de cas, et elle est au cœur de la plupart des transformations MDE, y compris dans Xtend (et EMF).

Visitor Pattern (problems)

```
public class WhileLoopNode extends Node {
  protected Node condition, body;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitWhileLoop(this);
  }
}
public class IfThenNode extends Node {
  protected Node condition, thenBranch;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitIfThen(this);
  }
}
```




```
public abstract class NodeVisitor {
  /* ... */
  public abstract void visitWhileLoop(WhileLoopNode n);
  public abstract void visitIfThen(IfThenNode n);
}
public class TypeCheckingVisitor extends NodeVisitor {
  /* ... */
  public void visitWhileLoop(WhileLoopNode n) { n.getCondition().accept(this); /* ... */ }
  public void visitIfThen(IfThenNode n) { /* ... */ }
}
```

#1 stylized double-dispatching code is tedious to write and prone to error.

Maintenant que l'on connaît les fondements du *Visitor Pattern*, parlons de ses limitations concrètes, notamment en contexte MDE.

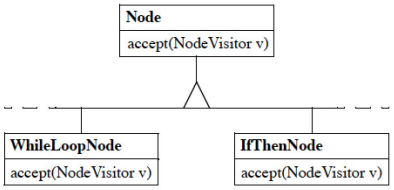
Premier problème :

Le *code de double dispatch* (i.e. ce fameux accept(this)) est fastidieux à écrire et surtout, très sujet à erreurs.

On répète le même patron pour chaque type de nœud, ce qui rend le code verbeux et difficile à maintenir.

Visitor Pattern (problems)

```
public class WhileLoopNode extends Node {
  protected Node condition, body;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitWhileLoop(this);
  }
}
public class IfThenNode extends Node {
  protected Node condition, thenBranch;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitIfThen(this);
  }
}
```




```
public abstract class NodeVisitor {
  /* ... */
  public abstract void visitWhileLoop(WhileLoopNode n);
  public abstract void visitIfThen(IfThenNode n);
}
public class TypeCheckingVisitor extends NodeVisitor {
  /* ... */
  public void visitWhileLoop(WhileLoopNode n) { n.getCondition().accept(this); /* ... */ }
  public void visitIfThen(IfThenNode n) { /* ... */ }
}
```

#2 the need for the Visitor pattern must be anticipated ahead of time, when the Node class is first implemented

Deuxième problème :

Il faut anticiper dès le départ l'utilisation d'un visiteur.

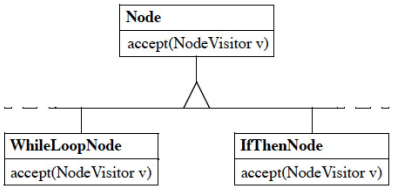
Si on n'a pas prévu de méthode accept dans la classe de base (Node),

on ne peut pas appliquer rétroactivement le pattern sans modifier toute la hiérarchie.

Donc si l'architecture change, ou si un nouveau besoin apparaît, on est bloqué.

Visitor Pattern (problems)

```
public class WhileLoopNode extends Node {
  protected Node condition, body;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitWhileLoop(this);
  }
}
public class IfThenNode extends Node {
  protected Node condition, thenBranch;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitIfThen(this);
  }
}
```

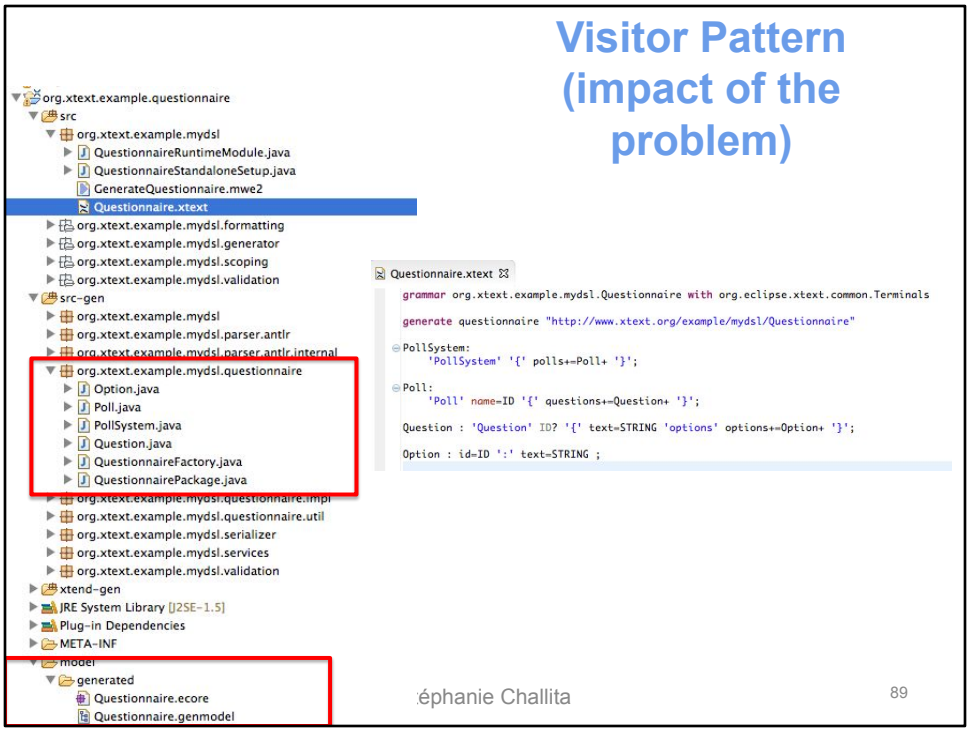



```
public abstract class NodeVisitor {
  /* ... */
  public abstract void visitWhileLoop(WhileLoopNode n);
  public abstract void visitIfThen(IfThenNode n);
}
public class TypeCheckingVisitor extends NodeVisitor {
  /* ... */
  public void visitWhileLoop(WhileLoopNode n) { n.getCondition().accept(this); /* ... */ }
  public void visitIfThen(IfThenNode n) { /* ... */ }
}
```

#3 class hierarchy evolution (e.g., new Node subclass) forces us to rewrite NodeVisitor

Troisième problème :
Chaque fois qu'on ajoute une nouvelle sous-classe dans la hiérarchie —
disons un nouveau type de nœud pour notre langage —
il faut réécrire ou mettre à jour tous les visiteurs existants pour qu'ils sachent quoi faire avec ce nouveau type.

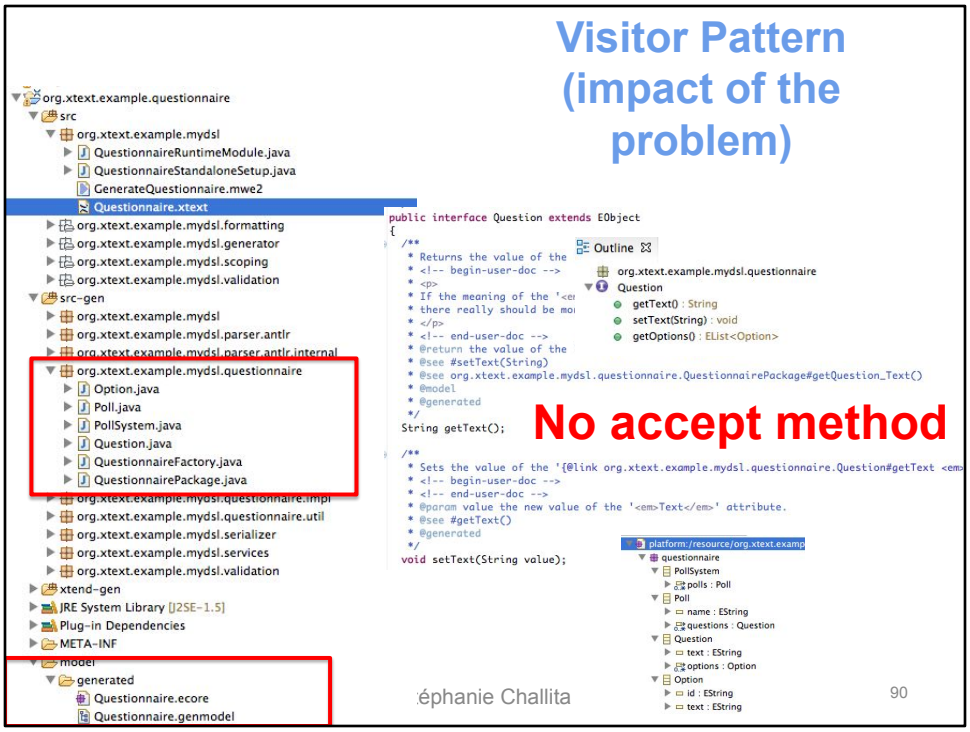
Donc, en résumé : le *Visitor Pattern* est puissant,
mais lourd à maintenir à mesure que la hiérarchie de classes évolue.
C'est pour ça que les outils comme EMF ou les langages comme Xtend proposent des variantes automatisées ou plus légères,
qu'on va justement explorer ensuite.



Nous voyons ici l'impact concret des problèmes liés au Visitor Pattern dans un projet basé sur Xtext.

À gauche, la structure du projet :

- La grammaire Questionnaire.xtext est définie dans le dossier src.
- Et comme vous pouvez le constater, cette définition génère automatiquement tout un ensemble de classes Java sous src-gen : Poll, Option, Question, etc.



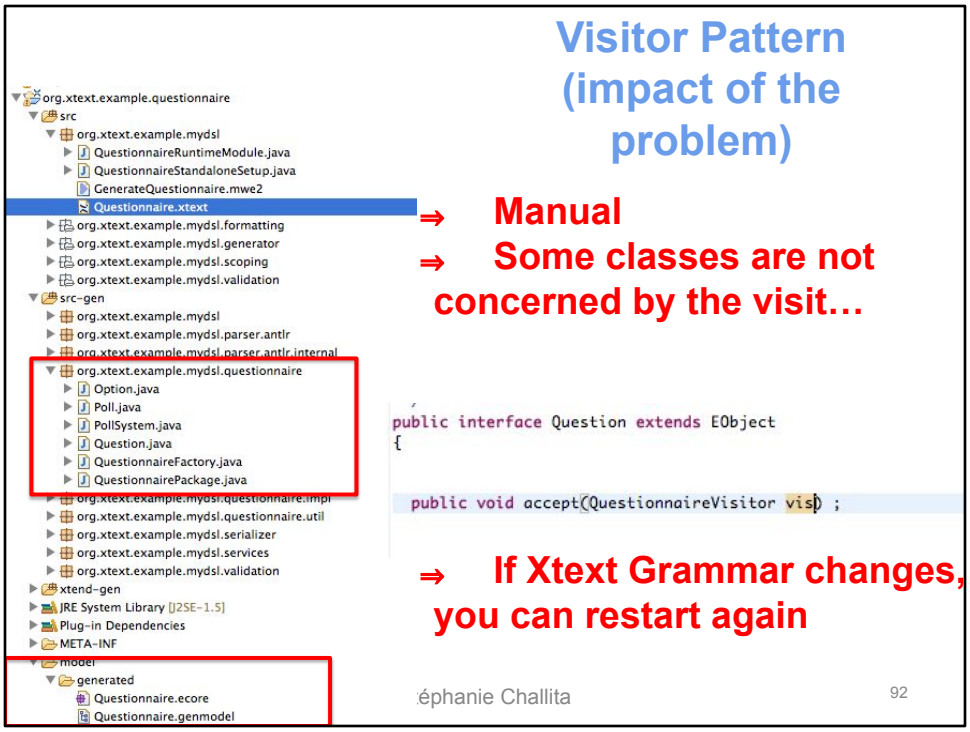
Ce que cela veut dire :
À partir de la grammaire déclarative qu'on voit à droite, Xtext et EMF vont générer la hiérarchie de classes Java correspondantes à notre métamodèle.

Et maintenant, imaginez qu'on veuille appliquer un traitement sur ces objets, par exemple, vérifier tous les IDs, ou générer du code.

CLICK

On a pas de méthode accept qui nous permettrait de visiter tous les noeuds de notre grammaire.

Avec une approche classique, on serait obligé d'écrire des visiteurs manuellement pour chaque type (Poll, Option, Question, etc.).



Résultat : on est fortement couplé à la hiérarchie générée, et toute évolution dans la grammaire (par exemple l'ajout d'un nouveau concept comme Section ou Category) impliquerait de mettre à jour manuellement tous les visiteurs associés.

C'est exactement le type de friction qu'on cherche à éviter… et c'est ce que propose de résoudre Xtend, qu'on va explorer plus en détail juste après.

Visitor Pattern (requirements)

#1 stylized double-dispatching code is tedious to write and prone to error.

Automation

#2 the need for the Visitor pattern must be anticipated ahead of time, when the Node class is first implemented

No accept method
Violation of open/close principle: no way

#3 class hierarchy evolution (e.g., new Node subclass) forces us to (completely) rewrite NodeVisitor

Automation

On fait ici une synthèse des problèmes identifiés avec le Visitor Pattern, et on les reformule en exigences pour une solution plus moderne, plus flexible.

Premier point :

> *"Stylized double-dispatching code is tedious to write and prone to error."*
> Cela fait référence à toute la mécanique répétitive du Visitor Pattern : chaque classe a sa méthode accept, chaque visiteur redéfinit visitX pour chaque classe X…
> C'est lourd, verbeux, et souvent source d'erreurs.
> On veut de l'automatisation.

Deuxième point :

> *"The need for the Visitor pattern must be anticipated ahead of time."*
> Le Visitor Pattern doit être prévu dès le départ. Il faut penser à mettre les méthodes accept() dans toutes les classes, sinon c'est trop tard.
> Mais surtout, cela viole le principe open/closed : impossible d'ajouter un nouveau type de traitement sans modifier les classes existantes.
> Il nous faut une solution non invasive, sans méthode accept().

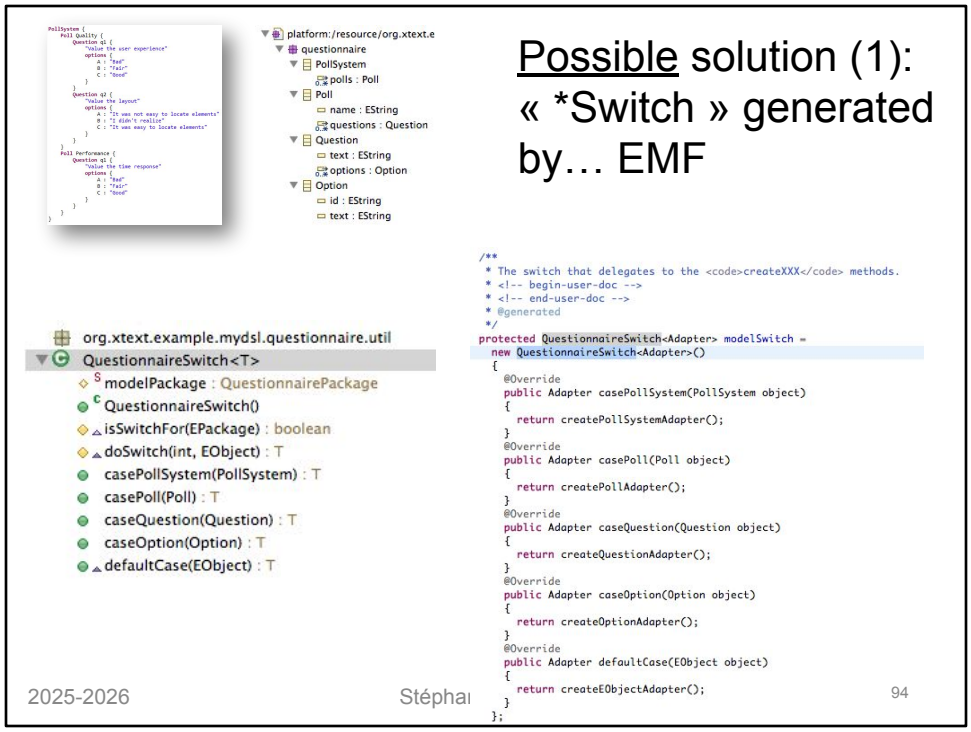
Troisième point :

> *"Class hierarchy evolution forces us to rewrite NodeVisitor."*
> Si on ajoute une nouvelle classe à la hiérarchie (disons une nouvelle sorte de Node), alors tous les visiteurs doivent être mis à jour pour la prendre en charge.

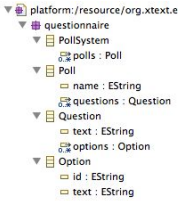
Cela casse les anciens traitements, ou demande des efforts constants pour rester synchronisés.

Là encore, on veut automatiser cette prise en compte des évolutions.

Vous voyez ici émerger les besoins fondamentaux qui ont conduit à proposer des alternatives comme les dispatchers d'Xtend, qu'on va introduire ensuite.



Alors ici, on regarde une première solution possible au problème de la visite du modèle, notamment pour contourner les limites du Visitor Pattern qu'on a évoquées juste avant.
Cette solution est générée automatiquement par EMF : il s'agit de la classe QuestionnaireSwitch<T>, que vous voyez ici à gauche dans le projet.
Cette classe propose une structure de méthodes caseXXX(...), une par type du métamodèle : casePollSystem, casePoll, caseQuestion, etc. Vous voyez un extrait à droite, où ces méthodes redirigent vers des adaptateurs créés.
Ce mécanisme de dispatch explicite permet de parcourir le graphe du modèle et d'appliquer du traitement spécifique à chaque type: sans avoir à modifier les classes du modèle lui-même, ni à introduire de double dispatch manuellement.
C'est donc une solution pratique, surtout quand le modèle est généré par EMF, et elle est plus souple que le Visitor Pattern classique."**



Possible solution (2): Extension Methods of Xtend

```
def foo(PollSystem sys, Context c) {
    // treatment
}
```

```
pollSystem.foo (new Context)
```

Context (classical with the Visitor)

Can be seen as a way
to avoid a (very) long list of
parameters and record
the « state » of the visit

Maintenant, nous allons explorer ici une autre solution, plus idiomatique avec Xtend : les méthodes d'extension.
L'idée, c'est de définir une fonction foo qui prend en paramètre le système à traiter (PollSystem) ainsi qu'un contexte, qui permet de mémoriser l'état de la visite, un peu comme on le ferait avec un accumulateur ou un environnement. Cette méthode peut ensuite être appelée sur n'importe quelle instance du modèle comme s'il s'agissait d'une méthode native de l'objet : pollSystem.foo(new Context).
Ce pattern permet d'éviter deux choses :

1. L'ajout d'une infinité de paramètres à chaque appel de méthode, ce qui rend le code difficile à maintenir.
2. La nécessité d'implémenter un Visitor Pattern complexe, rigide, et coûteux à faire évoluer.

Ainsi, les méthodes d'extension de Xtend offrent une alternative souple et élégante à l'écriture de visiteurs explicites, en s'appuyant sur les capacités de transformation du langage.