Support de la paire IA — JavaFX en pratique : de la dépendance au runtime

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 Support de la paire IA — JavaFX en pratique : de la dépendance au runtime 1
1.1 1. Objet et mode d'emploi : ce que le rendu montre, ce que votre poste montre 2
1.2 2. Récupérer un support épinglé 2
1.3 3. Construire sans écran 7
1.4 4. JPMS : déclarer pour réutiliser 9
1.4.a fxapp1, l'application minimale 12
1.5 5. MVC : fxapp2 sous la loupe 12
1.6 6. Un runtime qui se build : le profil jlink 15
1.7 7. Sur votre poste : ouvrir la fenêtre (bloc non exécuté par le rendu) 17
1.8 8. Ce que ce support apporte à la suite 18

1 Support de la paire IA — JavaFX en pratique : de la dépendance au runtime

Ce support prolonge le cours en appliquant ses outils — Maven, le système de modules JPMS, la gestion de versions par étiquettes — à une véritable application graphique JavaFX. Il a été conçu pour fonder la paire de leçons consacrées à l'assistance automatisée par IA, dont il matérialise les engagements de maintenabilité. Il est construit sur le dépôt public `ebpro/sample-javafx-client`, qui contient deux petites applications (`fxapp1`, `fxapp2`) et un profil de build `jlink`, et dont le README fixe la version des modules JavaFX à **25.0.2** (provenance : README du tag `v1.0.0` ; section 7). Statut à énoncer : JavaFX ne fait pas partie du JDK — ce sont des bibliothèques et des outils distincts, développés par le projet OpenJFX et livrés ici comme dépendances déclarées dans le POM (projet OpenJFX).

1.1 1. Objet et mode d'emploi : ce que le rendu montre, ce que votre poste montre

Une interface graphique paraît difficile à réutiliser dans un rendu automatisé : elle exige un écran. En réalité, presque tout le cycle de vie d'une application JavaFX — récupérer, déclarer ses dépendances, compiler, construire un runtime complet — s'observe sans aucune interface graphique. Le mécanisme en est simple : la compilation, la lecture des sources et le linkage `jlink` opèrent sur du bytecode et sur le graphe de modules ; ils n'initialisent pas le pipeline graphique. L'affichage n'intervient qu'à l'exécution de l'application, lorsque le toolkit se lie à un serveur graphique — étape que le conteneur de rendu ne peut pas franchir, et que ce support ne lui demande pas de franchir. Le tableau ci-dessous est le contrat d'exécution de tout ce qui suit.

Étape	Dans ce support (CI, sans écran)	Sur votre poste
Clonage du dépôt épinglé `v1.0.0`	oui	oui
`./mvnw package` (compilation, jar de module)	oui	oui
Lecture de `module-info.java` et des sources MVC	oui	—
`./mvnw -q -P jlink package` (runtime sur mesure)	oui	oui
Ouverture de la fenêtre (`javafx:run`, `myapp`)	**non**	oui

Rien n'est affirmé ici sans vérification : chaque commande présentée a été vérifiée dans le dépôt cloné, et tout chemin cité ici en est extrait.

1.2 2. Récupérer un support épinglé

Nous clonons la version **étiquetée** `v1.0.0`, et non la branche de développement : un document reproductible épingle aussi ses supports, exactement comme le cours épingle ses outils — c'est la même thèse de maintenabilité appliquée à un dépôt d'exemples. Le dépôt est public ; le clone passe donc en HTTPS, avec une boucle défensive de trois tentatives contre les faux départs réseau.

```
rm -rf /tmp/javafx-tp
mkdir -p /tmp/javafx-tp
cd /tmp/javafx-tp
```

```
for i in 1 2 3; do
  git clone --branch v1.0.0 --depth 1 https://github.com/ebpro/sample-javafx-client.git && break
  echo "tentative $i échouée — nouvelle tentative" && sleep 5 && rm -rf sample-javafx-client
done
ls sample-javafx-client
```

```
Cloning into 'sample-javafx-client'...
remote: Enumerating objects: 50, done.
remote: Counting objects:   2% (1/50)
remote: Counting objects:   4% (2/50)
remote: Counting objects:   6% (3/50)
remote: Counting objects:   8% (4/50)
remote: Counting objects:  10% (5/50)
remote: Counting objects:  12% (6/50)
remote: Counting objects:  14% (7/50)
remote: Counting objects:  16% (8/50)
remote: Counting objects:  18% (9/50)
remote: Counting objects:  20% (10/50)
remote: Counting objects:  22% (11/50)
remote: Counting objects:  24% (12/50)
remote: Counting objects:  26% (13/50)
remote: Counting objects:  28% (14/50)
remote: Counting objects:  30% (15/50)
remote: Counting objects:  32% (16/50)
remote: Counting objects:  34% (17/50)
remote: Counting objects:  36% (18/50)
remote: Counting objects:  38% (19/50)
remote: Counting objects:  40% (20/50)
remote: Counting objects:  42% (21/50)
remote: Counting objects:  44% (22/50)
remote: Counting objects:  46% (23/50)
remote: Counting objects:  48% (24/50)
remote: Counting objects:  50% (25/50)
remote: Counting objects:  52% (26/50)
remote: Counting objects:  54% (27/50)
remote: Counting objects:  56% (28/50)
remote: Counting objects:  58% (29/50)
remote: Counting objects:  60% (30/50)
remote: Counting objects:  62% (31/50)
remote: Counting objects:  64% (32/50)
remote: Counting objects:  66% (33/50)
```

```
remote: Counting objects:  68% (34/50)
remote: Counting objects:  70% (35/50)
remote: Counting objects:  72% (36/50)
remote: Counting objects:  74% (37/50)
remote: Counting objects:  76% (38/50)
remote: Counting objects:  78% (39/50)
remote: Counting objects:  80% (40/50)
remote: Counting objects:  82% (41/50)
remote: Counting objects:  84% (42/50)
remote: Counting objects:  86% (43/50)
remote: Counting objects:  88% (44/50)
remote: Counting objects:  90% (45/50)
remote: Counting objects:  92% (46/50)
remote: Counting objects:  94% (47/50)
remote: Counting objects:  96% (48/50)
remote: Counting objects:  98% (49/50)
remote: Counting objects: 100% (50/50)
remote: Counting objects: 100% (50/50), done.
remote: Compressing objects:   3% (1/31)
remote: Compressing objects:   6% (2/31)
remote: Compressing objects:   9% (3/31)
remote: Compressing objects:  12% (4/31)
remote: Compressing objects:  16% (5/31)
remote: Compressing objects:  19% (6/31)
remote: Compressing objects:  22% (7/31)
remote: Compressing objects:  25% (8/31)
remote: Compressing objects:  29% (9/31)
remote: Compressing objects:  32% (10/31)
remote: Compressing objects:  35% (11/31)
remote: Compressing objects:  38% (12/31)
remote: Compressing objects:  41% (13/31)
remote: Compressing objects:  45% (14/31)
remote: Compressing objects:  48% (15/31)
remote: Compressing objects:  51% (16/31)
remote: Compressing objects:  54% (17/31)
remote: Compressing objects:  58% (18/31)
remote: Compressing objects:  61% (19/31)
remote: Compressing objects:  64% (20/31)
remote: Compressing objects:  67% (21/31)
remote: Compressing objects:  70% (22/31)
remote: Compressing objects:  74% (23/31)
remote: Compressing objects:  77% (24/31)
remote: Compressing objects:  80% (25/31)
remote: Compressing objects:  83% (26/31)
```

```
remote: Compressing objects:  87% (27/31)
remote: Compressing objects:  90% (28/31)
remote: Compressing objects:  93% (29/31)
remote: Compressing objects:  96% (30/31)
remote: Compressing objects: 100% (31/31)
remote: Compressing objects: 100% (31/31), done.
remote: Total 50 (delta 4), reused 46 (delta 4), pack-reused 0 (from 0)
Receiving objects:   2% (1/50)
Receiving objects:   4% (2/50)
Receiving objects:   6% (3/50)
Receiving objects:   8% (4/50)
Receiving objects:  10% (5/50)
Receiving objects:  12% (6/50)
Receiving objects:  14% (7/50)
Receiving objects:  16% (8/50)
Receiving objects:  18% (9/50)
Receiving objects:  20% (10/50)
Receiving objects:  22% (11/50)
Receiving objects:  24% (12/50)
Receiving objects:  26% (13/50)
Receiving objects:  28% (14/50)
Receiving objects:  30% (15/50)
Receiving objects:  32% (16/50)
Receiving objects:  34% (17/50)
Receiving objects:  36% (18/50)
Receiving objects:  38% (19/50)
Receiving objects:  40% (20/50)
Receiving objects:  42% (21/50)
Receiving objects:  44% (22/50)
Receiving objects:  46% (23/50)
Receiving objects:  48% (24/50)
Receiving objects:  50% (25/50)
Receiving objects:  52% (26/50)
Receiving objects:  54% (27/50)
Receiving objects:  56% (28/50)
Receiving objects:  58% (29/50)
Receiving objects:  60% (30/50)
Receiving objects:  62% (31/50)
Receiving objects:  64% (32/50)
Receiving objects:  66% (33/50)
Receiving objects:  68% (34/50)
Receiving objects:  70% (35/50)
Receiving objects:  72% (36/50)
Receiving objects:  74% (37/50)
```

```
Receiving objects:  76% (38/50)
Receiving objects:  78% (39/50)
Receiving objects:  80% (40/50)
Receiving objects:  82% (41/50)
Receiving objects:  84% (42/50)
Receiving objects:  86% (43/50)
Receiving objects:  88% (44/50)
Receiving objects:  90% (45/50)
Receiving objects:  92% (46/50)
Receiving objects:  94% (47/50)
Receiving objects:  96% (48/50)
Receiving objects:  98% (49/50)
Receiving objects: 100% (50/50)
Receiving objects: 100% (50/50), 25.36 KiB | 5.07 MiB/s, done.
Resolving deltas:   0% (0/4)
Resolving deltas:  25% (1/4)
Resolving deltas:  50% (2/4)
Resolving deltas:  75% (3/4)
Resolving deltas: 100% (4/4)
Resolving deltas: 100% (4/4), done.
warning: refs/tags/v1.0.0 6c54224cdd80442a9392f9b38eae4901b21c7e3b is not a
commit!
Note: switching to '4ea660551853bbf03830e6465e007fd48d85f18f'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

LICENCE    _quarto-utils  mvnw.cmd  sonar-project.properties
README.md  mvnw       pom.xml   src
```

La racine contient notamment `pom.xml`, le wrapper `mvnw`, et `src/` : un projet Maven standard, avec un wrapper — vous n'avez donc pas besoin d'installer Maven pour la suite.

> ℹ Définition — Épinglement par étiquette
>
> L'**épinglement** consiste à figer la base de travail d'un support sur une révision identifiée par une **étiquette** (`tag`) plutôt que sur une branche. Une étiquette désigne une révision immobile : elle ne bouge pas quand d'autres commits sont ajoutés, alors qu'une branche avance à chaque commit. Travailler sur l'étiquette `v1.0.0` garantit donc que la lecture, la compilation et le linkage de ce support portent sur le même contenu, aujourd'hui et l'an prochain.
>
> Exemple : la cellule exécutée ci-dessus clone `--branch v1.0.0 --depth 1` ; la sortie `ls` attendue est la racine de cette révision figée (`pom.xml`, `mvnw`, `src/`), non celle de la branche de développement.
>
> Cas limite : si aucune étiquette n'existe, le support retombe sur une branche — et devient mobile : deux rendus à deux dates peuvent construire deux contenus différents sans qu'aucune erreur ne le signale. L'absence d'étiquette est un défaut du support, pas une option du mode d'emploi.
>
> Source : page de manuel `gitglossary(7)`, entrée « tag object » ; la mécanique des étiquettes est enseignée dans la séquence Git du cours (sec-10), non réouverte ici.

1.3 3. Construire sans écran

Passez en revue l'arborescence des sources avant de compiler :

```
cd /tmp/javafx-tp/sample-javafx-client
find src -type f | sort
```

```
src/main/java/fr/univtln/bruno/samples/jfx/fxapp1/App.java
src/main/java/fr/univtln/bruno/samples/jfx/fxapp1/Controller.java
src/main/java/fr/univtln/bruno/samples/jfx/fxapp2/App.java
src/main/java/fr/univtln/bruno/samples/jfx/fxapp2/model/Page.java
src/main/java/fr/univtln/bruno/samples/jfx/fxapp2/model/Person.java
src/main/java/fr/univtln/bruno/samples/jfx/fxapp2/model/PersonRepository.java
src/main/java/fr/univtln/bruno/samples/jfx/fxapp2/view/Controller.java
src/main/java/module-info.java
src/main/resources/fr/univtln/bruno/samples/jfx/fxapp1/fxapp01.fxml
src/main/resources/fr/univtln/bruno/samples/jfx/fxapp1/styles.css
```

```
src/main/resources/fr/univtln/bruno/samples/jfx/fxapp2/fxapp02.fxml
src/main/resources/fr/univtln/bruno/samples/jfx/fxapp2/styles.css
```

Deux applications cohabitent dans un seul module : `fxapp1` (quatre fichiers : une classe `App`, un contrôleur, un FXML, une feuille CSS) et `fxapp2` (sept fichiers : cinq classes Java réparties en paquetages `view` et `model`, plus le FXML et la feuille CSS). Lançons la construction :

```
cd /tmp/javafx-tp/sample-javafx-client
./mvnw -q package
```

```
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
WARNING: A restricted method in java.lang.System has been called
WARNING: java.lang.System::load has been called by
org.fusesource.jansi.internal.JansiLoader in an unnamed module (file:/home/
jovyan/.m2/wrapper/dists/apache-maven-3.9.9/3477a4f1/lib/jansi-2.4.1.jar)
WARNING: Use --enable-native-access=ALL-UNNAMED to avoid a warning for callers
in this module
WARNING: Restricted methods will be blocked in a future release unless native
access is enabled

WARNING: A terminally deprecated method in sun.misc.Unsafe has been called
WARNING: sun.misc.Unsafe::objectFieldOffset has been called by
com.google.common.util.concurrent.AbstractFuture$UnsafeAtomicHelper (file:/
home/jovyan/.m2/wrapper/dists/apache-maven-3.9.9/3477a4f1/lib/guava-33.2.1-
jre.jar)
WARNING: Please consider reporting this to the maintainers of class
com.google.common.util.concurrent.AbstractFuture$UnsafeAtomicHelper
WARNING: sun.misc.Unsafe::objectFieldOffset will be removed in a future release
```

La première exécution télécharge les dépendances : le wrapper Maven lui-même, puis les artefacts JavaFX et les bibliothèques tierces déclarées dans le POM. Cette phase peut durer plusieurs minutes ; les exécutions suivantes sont quasi instantanées grâce au cache local. Deux faits de construction méritent d'être notés avant toute interprétation. D'une part, le POM du tag compile avec la cible `release` 21 : c'est une propriété du dépôt épinglé (POM du tag `v1.0.0`), pas une consigne du cours, et rien ici ne demande à la corriger. D'autre part, le POM redirige le jar d'application vers `target/modules` (`maven-jar-plugin`, élément `outputDirectory`) — le jar est traité comme un module destiné à `jlink`, non comme un exécutable autonome : un jar autonome porte un manifeste qui nomme une classe principale et s'exécute par `java -jar`, tandis qu'ici le jar est une unité de linkage que `jlink` intègre dans une image ; la

raison de cette destination — l'alimentation de `jlink` — est démontrée à la section 6. Inspectons le résultat :

```
cd /tmp/javafx-tp/sample-javafx-client
ls target/
ls target/modules/
```

```
classes  generated-sources  maven-archiver  maven-status  modules
sample-jfx-0.1.0-SNAPSHOT.jar
```

La sortie attendue montre `target/modules` peuplé du jar produit. Notez enfin le décalage d'identification : l'étiquette Git `v1.0.0` épingle le code source, tandis que le POM porte encore sa version de développement `0.1.0-SNAPSHOT` — les deux mécanismes sont complémentaires, pas redondants : l'un fixe une révision du dépôt, l'autre nomme l'artefact que Maven publie.

1.4 4. JPMS : déclarer pour réutiliser

Le fichier qui distingue ce projet d'un simple dépôt de classes est en tête des sources. Lisons-le intégralement :

```
cd /tmp/javafx-tp/sample-javafx-client
cat src/main/java/module-info.java
```

```
/**
 * Module definition for JavaFX applications.
 * This module contains sample JavaFX applications that demonstrate
 * various concepts and patterns in JavaFX development.
 */
module fr.univtln.bruno.samples.jfx {
    // Required JavaFX modules
    requires javafx.controls;    // For JavaFX UI controls
    requires javafx.fxml;        // For FXML-based UI definitions

    // Third-party dependencies
    requires lombok;             // For reducing boilerplate code
    requires javafaker;          // For generating sample data
    requires org.slf4j;          // For logging with SLF4J

    // Java standard modules
    requires java.logging;       // For application logging
    requires java.sql;           // For database connectivity
```

```
    // Module exports and opens for JavaFX app 1 (simple example)
    exports fr.univtln.bruno.samples.jfx.fxapp1;               // Export APIs
     opens fr.univtln.bruno.samples.jfx.fxapp1 to javafx.fxml;   // Allow FXML
reflection access

    // Module exports and opens for JavaFX app 2 (MVC architecture)
     exports fr.univtln.bruno.samples.jfx.fxapp2;                    // Export
main package
     opens fr.univtln.bruno.samples.jfx.fxapp2 to javafx.fxml;   // Allow FXML
reflection access

    // View layer exports and opens
      exports fr.univtln.bruno.samples.jfx.fxapp2.view;              // Export
view components
     opens fr.univtln.bruno.samples.jfx.fxapp2.view to javafx.fxml;   // Allow
FXML reflection access

    // Model layer exports and opens
      exports fr.univtln.bruno.samples.jfx.fxapp2.model;             // Export
model classes
     opens fr.univtln.bruno.samples.jfx.fxapp2.model to javafx.fxml;  // Allow
FXML reflection access
}
```

> **Définition — Module JPMS et ses trois verbes**
>
> Un **module** est une unité de composition et d'encapsulation du système de modules Java (JPMS) : un paquetage de paquetages, plus un descripteur `module-info.java` qui déclare ses relations aux autres modules. Le descripteur s'analyse en trois verbes distincts :
>
> - **`requires`** — dépendance : le module courant a besoin des API du module nommé ; sans cette ligne, le compilateur et la machine virtuelle considèrent que la dépendance n'existe pas.
> - **`exports`** — offre : le paquetage nommé est accessible aux autres modules ; sans elle, ses types publics restent internes au module.
> - **`opens`** — ouverture à la réflexion : le paquetage nommé est rendu profondément accessible à un module donné (ici `to javafx.fxml`), sans quoi la réflexion d'un autre module sur ses types est refusée.
>
> Exemple : la sortie exécutée ci-dessus applique les trois verbes — `requires javafx.controls;`, `requires javafx.fxml;`, `requires lombok;` (dépendances) ; `exports …` (paquetages offerts) ; `opens … to javafx.fxml;` (ouverture au chargeur FXML).
>
> Cas limite : une déclaration peut survivre à son usage. La sortie exécutée montre un `requires java.sql;` alors que l'application ne touche aucune base de données — reliquat d'un état antérieur du dépôt. Ce n'est pas une erreur d'exécution : c'est une dette rendue **visible** par JPMS, donc attaquable — repérer ces dépendances devenues inutiles fait partie de la maintenance des dépendances explicites.
>
> Source : documentation Java SE du paquetage `java.lang.module` (classe `ModuleDescriptor`, modélisation de `module-info.java`). Déjà vu en sec-11 sous l'angle des dépendances ; les trois verbes (`requires`/`exports`/`opens`) sont posés intégralement ici parce que la liaison FXML en dépend.

Il en résulte une propriété vérifiable ligne à ligne sur la sortie lue ci-dessus : la maintenance des dépendances commence par la mise en correspondance des déclarations du descripteur avec les usages réels du code.

Comparez le descripteur avec le POM : Maven y résout des artefacts (coordonnées, version, jar) ; `module-info.java` déclare des capacités (telle bibliothèque est nécessaire, tel paquetage est offert). Les deux documents se recoupent ici, mais ils répondent à deux questions différentes — « comment obtenir la bibliothèque ? » contre « qui dépend de quoi, à l'exécution ? ».

Deux liaisons méritent un développement, parce qu'elles engagent le comportement du système.

La liaison FXML–réflexion. Le chargeur FXML instancie les contrôleurs par réflexion. C'est précisément ce que la ligne `opens … to javafx.fxml;` rend possible. Sans elle, le refus est attendu à l'exécution du chargement : le chargeur lève une exception (`javafx.fxml.LoadException`, dont la cause est le refus d'accès réflexif du système de modules) et l'application ne démarre pas. Ce cas limite ne peut pas être exécuté dans le rendu — le chargement FXML appartient à l'étape d'affichage, exclue par le tableau de la section 1 — ; il est énoncé ici avec sa source, la spécification d'encapsulation du JPMS (paquetage `java.lang.module`, ouverture réflexive conditionnelle).

Le cycle de vie de l'application. `App.java` de `fxapp1` montre le cycle de vie JavaFX : `main` appelle `launch`, qui rappelle `init`, puis `start` (chargement du FXML, création de la `Scene`, affichage du `Stage`), puis `stop` à la fermeture. Cet ordre est celui défini par la javadoc de `javafx.application.Application` (javadoc de la classe `Application`) ; il n'est pas observé dans le rendu, qui s'arrête à la construction.

1.4.a fxapp1, l'application minimale

`fxapp1` tient en quatre fichiers, ceux du listing `find` de la section 3 sous le paquetage `fxapp1`. `Controller.java` illustre la liaison déclarative : des champs annotés `@FXML` (dont `taskInputField`, `taskListView` et `addButton`) sont peuplés par le runtime, et la méthode `initialize()` les équipe de gestionnaires d'événements. Une liste de tâches, un champ de saisie, deux boutons — assez pour discerner chaque rôle, pas assez pour masquer le mécanisme.

1.5 5. MVC : fxapp2 sous la loupe

`fxapp2` applique la même trame, mais en séparant les rôles en paquetages. Visualisons la structure :

```
cd /tmp/javafx-tp/sample-javafx-client
ls     src/main/java/fr/univtln/bruno/samples/jfx/fxapp2     src/main/java/fr/
univtln/bruno/samples/jfx/fxapp2/view   src/main/java/fr/univtln/bruno/samples/
jfx/fxapp2/model
ls src/main/resources/fr/univtln/bruno/samples/jfx/fxapp2
```

```
src/main/java/fr/univtln/bruno/samples/jfx/fxapp2:
App.java  model  view

src/main/java/fr/univtln/bruno/samples/jfx/fxapp2/model:
Page.java  Person.java  PersonRepository.java

src/main/java/fr/univtln/bruno/samples/jfx/fxapp2/view:
Controller.java
fxapp02.fxml  styles.css
```

ℹ Définition — MVC (Modèle–Vue–Contrôleur)

MVC est un patron d'architecture applicative qui répartit le code en trois rôles distincts : le **modèle** — les données et les règles qui les régissent, indépendantes de toute interface ; la **vue** — la présentation de l'état du modèle ; le **contrôleur** — la traduction des entrées de l'utilisateur en demandes au modèle. Le critère du patron est le sens des dépendances : la vue et le contrôleur connaissent le modèle ; le modèle n'est censé connaître ni l'une ni l'autre.

Exemple : la sortie `ls` ci-dessus applique la répartition — `model/` (`Person`, `Page`, `PersonRepository`), `view/` (`Controller`), ressources (`fxapp02.fxml`, `styles.css`).

Cas limite : la frontière se juge à un critère observable, non à l'intention. Une valeur de couleur ou un calcul de présentation qui remonterait dans le contrôleur dissout le patron sans aucun message d'erreur : c'est exactement ce que les recherches `grep` de la section ci-dessous permettent de contrôler.

Source : Martin Fowler, Patterns of Enterprise Application Architecture, 2002, chapitre « Presentation Domain » (exposé moderne du patron) ; définition générale traitée en sec-05, non réouverte ici.

Trois étages, donc. Vérifions les trois points de couture entre eux — la vue ne connaît le modèle que par ces liaisons :

```
cd /tmp/javafx-tp/sample-javafx-client
grep   -n   "fx:controller\|stylesheets"   src/main/resources/fr/univtln/bruno/samples/jfx/fxapp2/fxapp02.fxml
grep  -n  "PersonRepository\|@FXML"  src/main/java/fr/univtln/bruno/samples/jfx/fxapp2/view/Controller.java | head -20
head -16 src/main/resources/fr/univtln/bruno/samples/jfx/fxapp2/styles.css
```

```
26:        fx:controller="fr.univtln.bruno.samples.jfx.fxapp2.view.Controller"
27:            stylesheets="@styles.css">
3:import fr.univtln.bruno.samples.jfx.fxapp2.model.PersonRepository;
48:               private    final    PersonRepository    personRepository    =
PersonRepository.newInstance();
54:    @FXML
57:    @FXML
60:    @FXML
63:    @FXML
66:    @FXML
69:    @FXML
71:    @FXML
```

```
73:    @FXML
81:    @FXML
282:                                                                    return
PersonRepository.newInstance().search(searchText).content();
/* General styles */
.root {
    -fx-font-family: 'Roboto', sans-serif;
    -fx-background-color: #f5f5f5;
}

/* Header styles */
.header {
    -fx-background-color: #b7dcea;
    -fx-background-radius: 20;
    -fx-padding: 20px;
    -fx-effect: dropshadow(three-pass-box, rgba(0,0,0,0.2), 10, 0, 0, 0);
}

.header .title {
    -fx-text-fill: white;
```

Les sorties exécutées établissent la liaison dans les deux sens : le FXML désigne son contrôleur par `fx:controller=…view.Controller` et sa charte graphique par `stylesheets="@styles.css"` ; le contrôleur consomme le modèle (`PersonRepository.newInstance()`) sur les champs annotés `@FXML` que le `grep` montre. Le mécanisme de cette désignation n'est pas nouveau : c'est la réflexion ouverte en section 4 (`opens … to javafx.fxml;`) — le chargeur FXML instancie le contrôleur nommé par l'attribut, sans que la vue importe sa classe. La feuille CSS, elle, ne parle qu'en classes de style (`header`, `toolbar`, `search-button`).

Il en résulte une séparation contrôlable par les sorties ci-dessus : le contrôleur ne contient aucune valeur graphique (le `grep` ne montre que des références au modèle et des annotations `@FXML`), et le FXML ne contient aucun calcul (il ne déclare que des composants et des classes de style). Par conséquent, une modification de charte graphique peut se limiter au fichier CSS sans toucher aux sources Java ; le contrôle en est la recherche des valeurs de couleur dans les paquetages Java, qui ne doit rien retourner.

Lisez le modèle pour clore le tour — une entité minimaliste, identifiée par son `UUID` :

```
cd /tmp/javafx-tp/sample-javafx-client
cat src/main/java/fr/univtln/bruno/samples/jfx/fxapp2/model/Person.java
```

```
package fr.univtln.bruno.samples.jfx.fxapp2.model;

import lombok.*;

import java.util.UUID;

@Getter
@ToString
@AllArgsConstructor(staticName = "of")
@NoArgsConstructor
@EqualsAndHashCode()
public class Person {
    @EqualsAndHashCode.Include
    private final UUID uuid = UUID.randomUUID();
    private String name;
    private String address;
}
```

La sortie attendue montre dix-sept lignes : une partie des membres de `Person` (comportement d'égalité, accèsseurs, fabrique `of`) est générée par Lombok. C'est une observation du code déposé, non un outil enseigné dans ce cours. Le dépôt simulé `PersonRepository` est de même peuplé par la bibliothèque `javafaker` — fait du tag `v1.0.0`, sans incidence sur les notions de cette section.

1.6 6. Un runtime qui se build : le profil jlink

Dernier étage, et le plus directement vérifiable pour la maintenabilité : le POM définit un profil nommé `jlink` (plugin `jigsaw-maven-plugin`) qui assemble, à la phase `package`, un runtime Java taillé pour la seule application. Lançons-le :

```
cd /tmp/javafx-tp/sample-javafx-client
./mvnw -q -P jlink package
```

```
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
WARNING: A restricted method in java.lang.System has been called
WARNING:       java.lang.System::load       has       been       called       by
org.fusesource.jansi.internal.JansiLoader  in  an  unnamed  module  (file:/home/
jovyan/.m2/wrapper/dists/apache-maven-3.9.9/3477a4f1/lib/jansi-2.4.1.jar)
WARNING: Use --enable-native-access=ALL-UNNAMED to avoid a warning for callers
in this module
WARNING: Restricted methods will be blocked in a future release unless native
access is enabled
```

```
WARNING: A terminally deprecated method in sun.misc.Unsafe has been called
WARNING: sun.misc.Unsafe::objectFieldOffset has been called by
com.google.common.util.concurrent.AbstractFuture$UnsafeAtomicHelper (file:/
home/jovyan/.m2/wrapper/dists/apache-maven-3.9.9/3477a4f1/lib/guava-33.2.1-
jre.jar)
WARNING: Please consider reporting this to the maintainers of class
com.google.common.util.concurrent.AbstractFuture$UnsafeAtomicHelper
WARNING: sun.misc.Unsafe::objectFieldOffset will be removed in a future release
```

> **i** Définition — Image de runtime (`jlink`)
>
> Une **image de runtime** est un répertoire produit par l'outil `jlink` du JDK : une machine virtuelle complète limitée à un ensemble choisi de modules, plutôt que le JDK entier installé sur la machine. L'ensemble est calculé par **résolution du graphe de modules** : à partir des modules racines indiqués, `jlink` inclut leur fermeture transitive de dépendances — ici le graphe de `module-info.java`, qui entraîne les modules JavaFX requis.
>
> Exemple : la vérification exécutée ci-dessous, `test -x target/image/bin/myapp`, confirme que l'image et son lanceur dédié (`myapp`, défini dans le POM par `<launcher>myapp=…</launcher>`) ont été produits ; `ls target/image/bin` en montre le contenu.
>
> Cas limite : le lanceur est **scellé au build**. `myapp` embarque la classe principale connue de `jlink` au moment de l'assemblage ; après un changement de `mainClass` dans le POM, l'ancien `myapp` continue de lancer l'ancienne application. Il faut relancer `./mvnw -q -P jlink package` pour que le lanceur suive — le runtime sur mesure est un artefact, et un artefact se régénère.
>
> Source : spécification des outils Java SE, page `jlink` (Java SE Tool Specifications, `jlink`).

Vérifions que l'image de runtime a bien été produite, avec son lanceur dédié :

```
cd /tmp/javafx-tp/sample-javafx-client
test -x target/image/bin/myapp && echo "image OK"
ls target/image/bin
```

```
image OK
java  jrunscript  keytool  myapp
```

La sortie attendue est la ligne `image OK`, suivie du contenu de `target/image/bin` : l'application ne dépend plus d'un `java` installé sur le poste cible, elle transporte le sien. Devenu artefact de la phase `package`, rattaché à l'étiquette du dépôt, le runtime se vérifie sur le disque par la commande ci-dessus — ce qui distingue une propriété vérifiable d'une simple promesse.

1.7 7. Sur votre poste : ouvrir la fenêtre (bloc non exécuté par le rendu)

⚠️ À faire — sur votre poste uniquement

Le README du dépôt fixe la condition : un JDK 23 ou plus récent (les modules JavaFX 25.0.2 l'exigent ; le POM compile avec `release` 21 — valeurs attribuées au README du tag `v1.0.0`, non des consignes du cours). La mise en place de ce JDK sur votre poste est détaillée dans le support d'environnement du cours (`90_prerequis_environnement`). Le rendu de ce support n'a pas d'affichage : les deux commandes ci-dessous ne sont pas exécutées ici. Deux chemins s'offrent à vous, selon que vous voulez développer ou livrer.

Développement — Maven lance l'application avec le JDK de votre poste :

```
# Bloc non exécuté par le rendu.
./mvnw javafx:run
```

Livraison — le runtime auto-suffisant construit à la section 6 :

```
# Bloc non exécuté par le rendu.
./target/image/bin/myapp
```

Critères de réussite (à observer sur votre écran) :

1. La fenêtre ouverte est celle de `fxapp2` : liste de personnes paginée visible — le POM pointe `mainClass` vers `fr.univtln.bruno.samples.jfx.fxapp2.App`.
2. Exercice de lecture du POM : changez temporairement la propriété `<mainClass>` vers `…fxapp1.App` et relancez `./mvnw javafx:run` ; critère atteint lorsque la fenêtre affiche la liste de tâches de `fxapp1` (section 4). Rétablissez ensuite le POM : ce changement est un exercice de lecture, pas une modification à remonter.
3. Si vous aviez construit `myapp` avant le changement, relancez `./mvnw -q -P jlink package` : sans cela, `myapp` garde la classe principale scellée au build précédent (cas limite de la section 6).

1.8 8. Ce que ce support apporte à la suite

Rien de nouveau n'a été introduit ici ; tout l'appareillage vient de briques déjà enseignées. **Maven et les modules JPMS** sont ceux du cours : le POM, le wrapper, le profil de build et `module-info.java` en sont l'application littérale (séquences Maven, sec-03, et JPMS, sec-11). Les **tests** que le dépôt pourrait porter relèvent des mêmes pratiques que celles du TP JDBC (sec-12). Le **clonage épinglé par étiquette** est celui du guide Git (sec-10). Ce que JavaFX ajoute n'est pas une technologie supplémentaire, mais un révélateur : une application graphique force les dépendances, les frontières et le runtime à devenir explicites, là où un programme en ligne de commande peut longtemps s'en dispenser.

Ce support établit trois propositions, chacune reliée à son critère d'observation :

1. Une dépendance non déclarée n'existe pas pour la machine : sans le `requires` correspondant, la compilation est rejetée (section 4).
2. Une frontière de rôles non tenue rend les modifications imprévisibles : les `grep` de la section 5 montrent exactement où passe la liaison modèle/vue/contrôleur, donc où une retouche graphique doit s'arrêter.
3. Un environnement qui n'est pas construit reste invérifiable : le `test -x` de la section 6 en donne la trace contraire sur le disque.

Ces trois propositions convergent vers la position que les leçons sur l'assistance automatisée réutilisent : maintenir un système, c'est rendre ses engagements explicites et vérifiables.