L’assistant de génération de code est un contributeur étranger

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

2026-10-31

Table des matières

1 Ce que l’assistant est, et n’est pas 1
2 Bien demander : contrat avant code 1
3 Le code généré est du code étranger 2
3.1 Porte 1 — la compilation 2
3.2 Porte 2 — les tests aux limites 5
3.3 Porte 3 — la maintenabilité 12
4 Ce que le TP vous demandera 15

1 Ce que l’assistant est, et n’est pas

Un assistant de génération de code est un **contributeur étranger** : il ne siège pas dans votre équipe, n’assiste à aucune de vos réunions de conception, et ne court jamais vos tests de son propre chef. Il produit du texte que nous choisissons, nous, de traiter comme du code — c’est-à-dire de le soumettre aux mêmes portes que n’importe quelle contribution humaine : la compilation, les tests, la revue. C’est le cours qui a construit ces portes ; l’assistant ne les contourne pas, il les franchit — ou il échoue devant elles, ce qui est tout aussi instructif.

Avertissement honnête sur ce qui suit : les réponses de l’assistant sont **épinglées** — aucun appel à une IA n’est passé par cette page ; les sorties de `javac` et de JUnit, elles, sont exécutées en direct par le rendu. Le choix pédagogique est ailleurs : nous ne choisissons pas ce que l’assistant répond, nous choisissons de soumettre ses réponses à nos portes.

2 Bien demander : contrat avant code

Une génération sans contrat, c’est une spécification vide : l’assistant remplira chaque blanc par sa propre hypothèse, silencieusement. Le prompt ci-dessous est volon-

tairement vague — il suffirait à produire « une classe playlist », au sens de l'assistant, pas au vôtre :

```
Écris-moi une classe playlist en Java.
```

Face à cela, un contrat : les signatures exactes attendues, les cas limites que la classe doit respecter, et le périmètre technique (« aucune dépendance hors JDK »). C'est ce contrat — et lui seul — qui a été soumis à l'assistant dans la suite de la leçon :

```
Écris en Java une classe Playlist représentant une liste de titres,
dans l'ordre d'ajout, sans aucune dépendance hors JDK.

API publique attendue :
  - void add(String morceau)         ajoute un morceau en fin de liste
  - void removeAt(int index)         retire le morceau à l'index donné ;
                                     IndexOutOfBoundsException si l'index est
                                     hors bornes
  - boolean contains(String morceau) vrai si le morceau figure dans la liste
  - List<String> topN(int n)         les n premiers morceaux, dans l'ordre ;
                                     moins si la liste est plus courte ;
                                     une copie indépendante de l'interne

Cas limites à respecter :
  - contains(null) renvoie false sans lever d'exception ;
  - topN(0) renvoie une liste vide ;
  - topN(n) avec n supérieur à la taille renvoie toute la liste.
```

Remarquez ce que le contrat ne dit pas : il n'impose aucune implémentation. Les contributions que nous allons juger satisfont toutes ce contrat à première vue — c'est exactement pour cela qu'il existe trois portes et non une.

3 Le code généré est du code étranger

Le domaine est le vôtre, familier depuis les TP d'objets : `Playlist`, quatre méthodes (`add`, `removeAt`, `contains`, `topN`), un état interne — une liste de titres dans l'ordre d'ajout. Cinq contributions successives de cette classe (quatre versions distinctes — la quatrième réintègre la première), produites par l'assistant en réponse au contrat ci-dessus, vont passer nos portes. Elles sont déposées sous `/tmp/ai-tp`, une arborescence par contribution (`f1`, `f2`, `f3`…), chaque cellule étant autonome.

3.1 Porte 1 — la compilation

Avant tout, l'outillage : un seul et même jar JUnit, réutilisé par toutes les cellules. Cette cellule le télécharge au besoin (trois tentatives, cinq secondes d'attente entre elles) et vérifie sa somme SHA-256 :

```
mkdir -p /tmp/ai-tp/lib && cd /tmp/ai-tp/lib && for i in 1 2 3; do [ -f junit-platform-console-standalone-1.11.4.jar ] || curl -sSLO https://repo1.maven.org/maven2/org/junit/platform/junit-platform-console-standalone/1.11.4/junit-platform-console-standalone-1.11.4.jar && break; sleep 5; done; echo "b016ef6b1c3454d6d7c2c88ce081dabf289699686af6622d6e4e2e1b54b4a2fc  junit-platform-console-standalone-1.11.4.jar" | sha256sum -c -
```

```
junit-platform-console-standalone-1.11.4.jar: OK
```

Un artefact réutilisé sans somme de contrôle n'est pas le même artefact : la vérification d'intégrité atteste que tout le monde exécute le même jar — c'est la leçon déjà vue à l'annexe JavaFX, appliquée ici à un simple téléchargement.

Première contribution. Elle compile, sans message, avec un code de sortie nul :

```
mkdir -p /tmp/ai-tp/f1
cd /tmp/ai-tp/f1
cat > Playlist.java <<'EOF'
import java.util.ArrayList;
import java.util.List;

// Playlist de morceaux, dans l'ordre d'ajout.
public class Playlist {

    private final List<String> morceaux = new ArrayList<>();

    // Ajoute un morceau en fin de liste.
    public void add(String morceau) {
        morceaux.add(morceau);
    }

    // Retire le morceau situe a l'index donne.
    public void removeAt(int index) {
        morceaux.remove(index);
    }

    // Vrai si le morceau figure dans la playlist.
    public boolean contains(String morceau) {
        return morceaux.contains(morceau);
    }

    // Les N premiers morceaux, dans l'ordre (moins si la liste est plus courte).
    public List<String> topN(int n) {
```

```
      return new ArrayList<>(morceaux.subList(0, Math.min(n, morceaux.size())));
    }
}
EOF
```

```
mkdir -p /tmp/ai-tp/f1/out
cd /tmp/ai-tp
javac -d f1/out f1/Playlist.java
```

```
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
```

Ici, **le silence est la sortie** : `javac` n'affiche rien quand tout va bien, et c'est son code de sortie qui parle. La compilation n'est pas une épreuve pour ce code-là ; elle en sera une pour le suivant.

Seconde contribution — une seule ligne la distingue de la première, dans `contains` :

```
mkdir -p /tmp/ai-tp/f2
cd /tmp/ai-tp/f2
cat > Playlist.java <<'EOF'
import java.util.ArrayList;
import java.util.List;

// Playlist de morceaux, dans l'ordre d'ajout.
public class Playlist {

    private final List<String> morceaux = new ArrayList<>();

    // Ajoute un morceau en fin de liste.
    public void add(String morceau) {
        morceaux.add(morceau);
    }

    // Retire le morceau situe a l'index donne.
    public void removeAt(int index) {
        morceaux.remove(index);
    }

    // Vrai si le morceau figure dans la playlist.
    public boolean contains(String morceau) {
        return morceaux.reverse().contains(morceau);
    }
```

```
    // Les N premiers morceaux, dans l'ordre.
    public List<String> topN(int n) {
      return new ArrayList<>(morceaux.subList(0, Math.min(n, morceaux.size())));
    }
}
EOF
```

```
mkdir -p /tmp/ai-tp/f2/out
cd /tmp/ai-tp
javac -d f2/out f2/Playlist.java
```

```
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
f2/Playlist.java:21: error: cannot find symbol
        return morceaux.reverse().contains(morceau);
                        ^
  symbol:   method reverse()
  location: variable morceaux of type List<String>
1 error
```

```
: 1
```

Le refus est précis, et il faut le lire comme une preuve, pas comme un obstacle :

```
f2/Playlist.java:21: error: cannot find symbol
        return morceaux.reverse().contains(morceau);
                        ^
  symbol:   method reverse()
  location: variable morceaux of type List<String>
```

Trois fragments à retenir : `cannot find symbol`, `symbol:   method reverse()`, `location: variable morceaux of type List<String>`. La leçon est typique des assistants entraînés sur tous les langages à la fois : depuis le JDK 21, `reversed()` est une méthode parfaitement légitime de `List` (héritée de `SequencedCollection`) — le fantôme ici est `reverse()`, un réflexe Python (`list.reverse()`), qui n'existe pas dans le JDK. Le compilateur ne condamne pas la classe entière : il pointe la ligne 21, nomme la méthode inconnue et la variable accusée. C'est précisément le niveau de détail qui rend la porte 1 si bon marché à franchir — et si difficile à contourner.

3.2 Porte 2 — les tests aux limites

Troisième contribution : elle compile (nous n'avons plus de `reverse()`), mais elle a modifié `removeAt` en silence — `morceaux.remove(index + 1)` retire le voisin de l'index

demandé — et a réécrit `contains` à la main avec une boucle. Le contrat, lui, reste satisfait en apparence. D'où la deuxième porte : la suite de tests aux limites, la même pour toutes les contributions.

```
mkdir -p /tmp/ai-tp/f3
cd /tmp/ai-tp/f3
cat > Playlist.java <<'EOF'
import java.util.ArrayList;
import java.util.List;

// Playlist de morceaux, dans l'ordre d'ajout.
public class Playlist {

    private final List<String> morceaux = new ArrayList<>();

    // Ajoute un morceau en fin de liste.
    public void add(String morceau) {
        morceaux.add(morceau);
    }

    // Retire le morceau situe a l'index donne.
    public void removeAt(int index) {
        morceaux.remove(index + 1);
    }

    // Vrai si le morceau figure dans la playlist.
    public boolean contains(String morceau) {
        for (String m : morceaux) {
            if (morceau.equals(m)) {
                return true;
            }
        }
        return false;
    }

    // Les N premiers morceaux, dans l'ordre.
    public List<String> topN(int n) {
      return new ArrayList<>(morceaux.subList(0, Math.min(n, morceaux.size())));
    }
}
EOF
cat > PlaylistTest.java <<'EOF'
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
```

```
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.List;
import org.junit.jupiter.api.Test;

class PlaylistTest {

    @Test
    void ajoutPuisPresence() {
        Playlist p = new Playlist();
        p.add("Alpha");
        assertTrue(p.contains("Alpha"));
    }

    @Test
    void topNRespecteLordre() {
        Playlist p = new Playlist();
        p.add("Alpha");
        p.add("Beta");
        p.add("Gamma");
        assertEquals(List.of("Alpha", "Beta"), p.topN(2));
    }

    @Test
    void removeAtRetireLeBonMorceau() {
        Playlist p = new Playlist();
        p.add("Alpha");
        p.add("Beta");
        p.add("Gamma");
        p.removeAt(0);
         assertFalse(p.contains("Alpha"), "Alpha etait a l'index 0, il ne doit
plus etre present");
    }

    @Test
    void containsTolerLeNull() {
        Playlist p = new Playlist();
         assertFalse(p.contains(null), "une playlist vide ne contient rien, pas
meme null");
    }
}
EOF
```

```
mkdir -p /tmp/ai-tp/f3/out
cd /tmp/ai-tp
javac -cp lib/junit-platform-console-standalone-1.11.4.jar -d f3/out f3/Playlist.java f3/PlaylistTest.java
java -jar lib/junit-platform-console-standalone-1.11.4.jar execute --disable-ansi-colors --disable-banner --class-path f3/out --scan-class-path f3/out
```

```
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
╷
├─ JUnit Platform Suite ✓
├─ JUnit Jupiter ✓
│  └─ PlaylistTest ✓
│     ├─ removeAtRetireLeBonMorceau() ✘ Alpha etait a l'index 0, il ne doit plus etre present ⟹ expected: <false> but was: <true>
│     ├─ containsTolerLeNull() ✓
│     ├─ ajoutPuisPresence() ✓
│     └─ topNRespecteLordre() ✓
└─ JUnit Vintage ✓

Failures (1):
  JUnit Jupiter:PlaylistTest:removeAtRetireLeBonMorceau()
            MethodSource [className = 'PlaylistTest', methodName = 'removeAtRetireLeBonMorceau', methodParameterTypes = '']
    ⇒ org.opentest4j.AssertionFailedError: Alpha etait a l'index 0, il ne doit plus etre present ⟹ expected: <false> but was: <true>
     org.junit.jupiter.api.AssertionFailureBuilder.build(AssertionFailureBuilder.java:151)
     org.junit.jupiter.api.AssertionFailureBuilder.buildAndThrow(AssertionFailureBuilder.java:132)
       org.junit.jupiter.api.AssertFalse.failNotFalse(AssertFalse.java:63)
       org.junit.jupiter.api.AssertFalse.assertFalse(AssertFalse.java:36)
       org.junit.jupiter.api.Assertions.assertFalse(Assertions.java:239)
       PlaylistTest.removeAtRetireLeBonMorceau(PlaylistTest.java:33)
       java.base/java.lang.reflect.Method.invoke(Method.java:565)
       java.base/java.util.ArrayList.forEach(ArrayList.java:1604)
       java.base/java.util.ArrayList.forEach(ArrayList.java:1604)

Test run finished after 131 ms
[         4 containers found      ]
[         0 containers skipped    ]
[         4 containers started    ]
[         0 containers aborted    ]
[         4 containers successful ]
[         0 containers failed     ]
```

```
[         4 tests found           ]
[         0 tests skipped         ]
[         4 tests started         ]
[         0 tests aborted         ]
[         3 tests successful      ]
[         1 tests failed          ]
```

```
: 1
```

Le rouge est réel, pas rhétorique : l'arbre du launcher marque `removeAtRetireLeBonMorceau() ✗` avec le message d'assertion exact (`expected: <false> but was: <true>`), et le summary affiche `[ 1 tests failed ]` — un seul, mais suffisant pour fermer la porte. Le décalage d'un cran dans `removeAt` est invisible à la compilation ; seul un test limites — retirer à l'index 0 puis vérifier la présence — le saisit.

Mais regardez de plus près le quatrième test : `containsTolerLeNull()` est **vert** ici, et c'est une leçon dans la leçon. Le `contains` de la version 3 déréférence `morceau` (`morceau.equals(m)`) ; sur une playlist vide, la boucle ne s'exécute jamais, donc le `null` n'est jamais touché. Sur une playlist non vide, `contains(null)` lèverait une `NullPointerException` — le défaut est là, latent, et la suite passe à côté parce qu'elle ne teste le `null` que sur liste vide. Retenez cette asymétrie : **une suite verte n'est pas une suite suffisante.** La porte 2 est plus solide que la porte 1, mais elle n'est pas imperméable — c'est l'argument même de la porte 3.

Quatrième contribution : la `Playlist` de la version 1, réintégrée, accompagnée de la copie stricte de la même suite. Elle repasse la porte :

```
mkdir -p /tmp/ai-tp/f4
cd /tmp/ai-tp/f4
cat > Playlist.java <<'EOF'
import java.util.ArrayList;
import java.util.List;

// Playlist de morceaux, dans l'ordre d'ajout.
public class Playlist {

    private final List<String> morceaux = new ArrayList<>();

    // Ajoute un morceau en fin de liste.
    public void add(String morceau) {
        morceaux.add(morceau);
    }
```

```
    // Retire le morceau situe a l'index donne.
    public void removeAt(int index) {
        morceaux.remove(index);
    }

    // Vrai si le morceau figure dans la playlist.
    public boolean contains(String morceau) {
        return morceaux.contains(morceau);
    }

    // Les N premiers morceaux, dans l'ordre (moins si la liste est plus courte).
    public List<String> topN(int n) {
      return new ArrayList<>(morceaux.subList(0, Math.min(n, morceaux.size())));
    }
}
EOF
cat > PlaylistTest.java <<'EOF'
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.List;
import org.junit.jupiter.api.Test;

class PlaylistTest {

    @Test
    void ajoutPuisPresence() {
        Playlist p = new Playlist();
        p.add("Alpha");
        assertTrue(p.contains("Alpha"));
    }

    @Test
    void topNRespecteLordre() {
        Playlist p = new Playlist();
        p.add("Alpha");
        p.add("Beta");
        p.add("Gamma");
        assertEquals(List.of("Alpha", "Beta"), p.topN(2));
    }

    @Test
    void removeAtRetireLeBonMorceau() {
```

```
        Playlist p = new Playlist();
        p.add("Alpha");
        p.add("Beta");
        p.add("Gamma");
        p.removeAt(0);
         assertFalse(p.contains("Alpha"), "Alpha etait a l'index 0, il ne doit
plus etre present");
    }

    @Test
    void containsTolerLeNull() {
        Playlist p = new Playlist();
         assertFalse(p.contains(null), "une playlist vide ne contient rien, pas
meme null");
    }
}
EOF
```

```
mkdir -p /tmp/ai-tp/f4/out
cd /tmp/ai-tp
javac  -cp  lib/junit-platform-console-standalone-1.11.4.jar  -d  f4/out  f4/
Playlist.java f4/PlaylistTest.java
java -jar lib/junit-platform-console-standalone-1.11.4.jar execute --disable-
ansi-colors --disable-banner --class-path f4/out --scan-class-path f4/out
```

```
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
╷
├─ JUnit Platform Suite ✓
├─ JUnit Jupiter ✓
│  └─ PlaylistTest ✓
│     ├─ removeAtRetireLeBonMorceau() ✓
│     ├─ containsTolerLeNull() ✓
│     ├─ ajoutPuisPresence() ✓
│     └─ topNRespecteLordre() ✓
└─ JUnit Vintage ✓

Test run finished after 113 ms
[         4 containers found      ]
[         0 containers skipped    ]
[         4 containers started    ]
[         0 containers aborted    ]
[         4 containers successful ]
```

```
[         0 containers failed     ]
[         4 tests found           ]
[         0 tests skipped         ]
[         4 tests started         ]
[         0 tests aborted         ]
[         4 tests successful      ]
[         0 tests failed          ]
```

L'arbre est intégralement coché, le summary affiche `[ 4 tests successful ]` et `[ 0 tests failed ]`. En CI, cette contribution serait mergée. Et pourtant — troisième porte.

3.3 Porte 3 — la maintenabilité

Cinquième contribution : conforme au contrat tel qu'il est écrit — qui ne dit nulle part si l'état est détenu par chaque instance ou partagé —, compilable, et une nouvelle suite (deux tests, plus légers) qui la valide :

```
mkdir -p /tmp/ai-tp/f5
cd /tmp/ai-tp/f5
cat > Playlist.java <<'EOF'
import java.util.ArrayList;
import java.util.List;

// Playlist "sans etat d'instance" : les morceaux sont partages.
public class Playlist {

    // Etat commun a toutes les instances de la JVM.
    private static final List<String> morceaux = new ArrayList<>();

    // Ajoute un morceau en fin de liste.
    public void add(String morceau) {
        morceaux.add(morceau);
    }

    // Retire le morceau situe a l'index donne.
    public void removeAt(int index) {
        morceaux.remove(index);
    }

    // Vrai si le morceau figure dans la playlist (parcours du topN complet).
    public boolean contains(String morceau) {
        for (String m : topN(morceaux.size())) {
            if (m.equals(morceau)) {
                return true;
```

```
            }
        }
        return false;
    }

    // Les N premiers morceaux, dans l'ordre.
    public List<String> topN(int n) {
        List<String> vus = new ArrayList<>();
        for (String m : morceaux) {
            if (vus.size() >= n) {
                break;
            }
            vus.add(m);
        }
        return vus;
    }
}
EOF
cat > MiniTest.java <<'EOF'
import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.Test;

class MiniTest {

    @Test
    void premierePlaylistRetrouveSesMorceaux() {
        Playlist a = new Playlist();
        a.add("Alpha");
        assertTrue(a.contains("Alpha"));
        assertTrue(a.topN(100).contains("Alpha"));
    }

    @Test
    void secondePlaylistRetrouveSesMorceaux() {
        Playlist b = new Playlist();
        b.add("Beta");
        assertTrue(b.contains("Beta"));
        assertTrue(b.topN(100).contains("Beta"));
    }
}
EOF
```

```
mkdir -p /tmp/ai-tp/f5/out
cd /tmp/ai-tp
javac -cp lib/junit-platform-console-standalone-1.11.4.jar -d f5/out f5/
Playlist.java f5/MiniTest.java
java -jar lib/junit-platform-console-standalone-1.11.4.jar execute --disable-
ansi-colors --disable-banner --class-path f5/out --scan-class-path f5/out
```

```
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
Picked up JAVA_TOOL_OPTIONS: -XX:MaxRAMPercentage=25
╷
├─ JUnit Platform Suite ✓
├─ JUnit Jupiter ✓
│  └─ MiniTest ✓
│     ├─ premierePlaylistRetrouveSesMorceaux() ✓
│     └─ secondePlaylistRetrouveSesMorceaux() ✓
└─ JUnit Vintage ✓

Test run finished after 109 ms
[         4 containers found      ]
[         0 containers skipped    ]
[         4 containers started    ]
[         0 containers aborted    ]
[         4 containers successful ]
[         0 containers failed     ]
[         2 tests found           ]
[         0 tests skipped         ]
[         2 tests started         ]
[         0 tests aborted         ]
[         2 tests successful      ]
[         0 tests failed          ]
```

Vert : `[ 2 tests successful ]`, `[ 0 tests failed ]`. Cette contribution passe donc les deux premières portes — et elle est quand même refusée par la troisième. La revue se conduit avec trois questions, et elles s'appliquent à tout code étranger, humain ou assisté.

Où est l'état ? Regardez le champ : `private static final List<String> morceaux`. Le `static` a glissé là, et il change tout : il n'y a plus une liste par playlist, mais **une seule liste pour toutes les instances de la JVM**. Deux objets `Playlist` distincts sont secrètement le même objet.

Que partage une deuxième instance ? Tout. `new Playlist()` deux fois, c'est la même collection mutée. Le plus pervers : la suite reste verte **en dépit de** ce défaut — aucun de ses tests ne fait intervenir deux playlists à la fois. Chacune retrouve son propre

ajout, ce qui serait aussi le cas avec une liste par instance ; rien, dans cette suite, n'observe l'état partagé. Deux playlists qui se partagent un même tour de chant ne sont pas deux playlists ; c'est une seule, mal nommée.

Que dit le diff ? Deux méthodes, `contains` et `topN`, se mettent à réécrire chacune leur propre parcours de la liste, alors qu'une ligne (`morceaux.contains(morceau)`) faisait le travail. La duplication de la logique d'itération est le second signal, visible uniquement à la lecture.

Assumons ce que cette porte est : **elle ne se franchit pas en CI, elle se franchit à l'œil** — et c'est pour ça qu'elle reste une porte. Aucun outil de cette leçon n'aurait rejeté la version 5 ; seule une lecture guidée par ces trois questions l'a fait. C'est exactement la pratique entraînée dans le guide Git (relire un diff ligne à ligne avant de l'approuver) : l'assistant peut produire le code, il ne peut pas produire le regard.

4 Ce que le TP vous demandera

Le fichier suivant de l'arc (`81_ai_P_…`) vous met en situation : vous êtes le reviewer. Trois nouvelles sorties de l'assistant, épinglées sur un autre domaine (`Reservation`), vous sont soumises. Vous compléterez une suite verte mais muette, écrirez le test qu'une autre suite n'a pas osé écrire, et vous rendrez un tableau de verdicts, sortie par sortie : laquelle franchit quelles portes.

Prérequis : le **TP 09** (mécanique JUnit : compiler avec le jar standalone, lancer une classe de test, lire l'arbre ✓/✗ et le summary) et le **guide Git** (relire un diff). Ni l'un ni l'autre n'est refait ici.