AVIGNON UNIVERSITÉ

Zero To Hero

Un jeu de rôle textuel en Rust

Groupe TD1 Alternants

Bradley Kissouna

06/05/2025

Travail réalisé en collaboration avec : **GEYER Rayane, HAZOURLI Mohamed Mehdi et BOUDOUNT Youssef**

Master Informatique
Parcours INGE DU LOGICIEL DE LA SOCIETE NUM (ILSEN)
UCE UCE Algorithme et Modélisation Avancée

Responsable
Pierre Jourlin
Lahcène Belhadi

UFR
SCIENCES
TECHNOLOGIES
SANTÉ

CERI

CENTRE
D'**ENSEIGNEMENT**
ET DE **RECHERCHE**
EN **INFORMATIQUE**
ceri.univ-avignon.fr

Sommaire

1 Introduction

1.1 Présentation générale du projet

Zero To Hero est un *RPG textuel sandbox* dans lequel le joueur incarne un alter-ego à la manière d'un protagoniste de GTA [1]. Le cœur du gameplay repose sur trois jauges : **Santé**, **Faim** et surtout **Aura** (charisme / réputation). Selon ses décisions (légales ou illégales), le personnage progresse du statut de citoyen anonyme à celui de « crime-lord multimillionnaire » ou termine en prison… voire à la morgue.

Toutes les fonctionnalités ont été conçues *de novo*, en nous appuyant d'abord sur une **carte mentale** (fig. 1) pour dégager les grandes entités, puis sur un **diagramme de classes UML** (fig. 2) avant toute ligne de code. Cette démarche a guidé la répartition modulaire décrite plus bas.

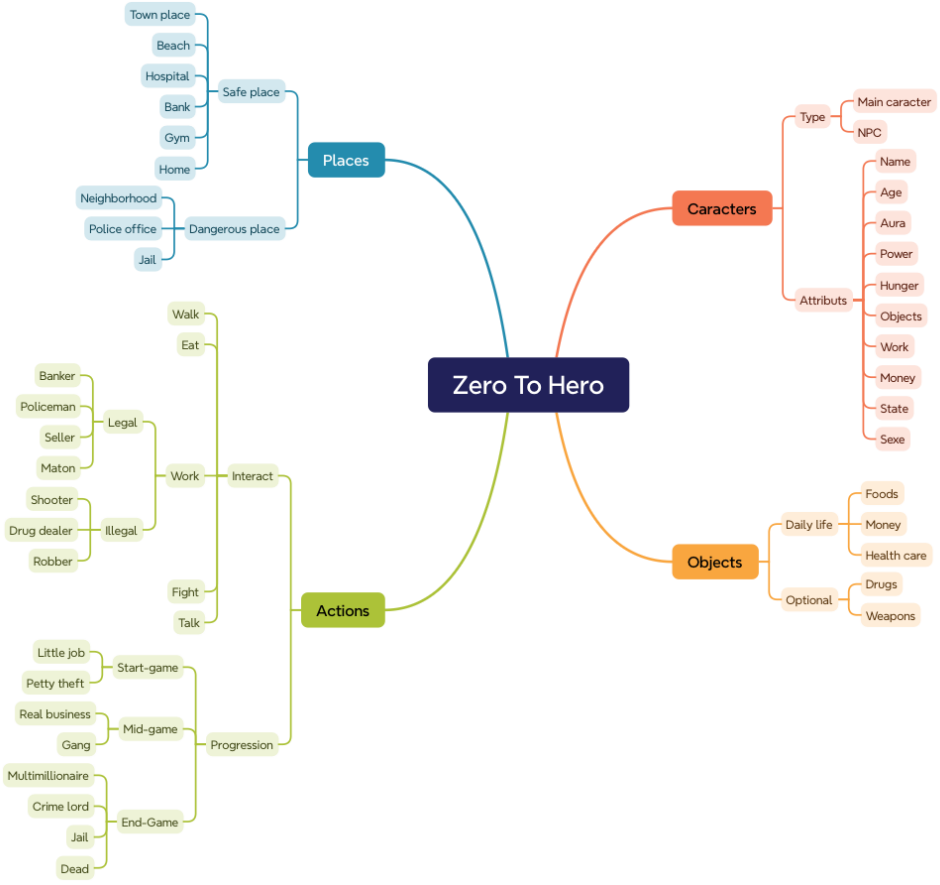


Figure 1. Carte mentale *Zero To Hero* : entités et mécaniques de jeu

1. Sans composante graphique 3D, uniquement par interface terminal ASCII.

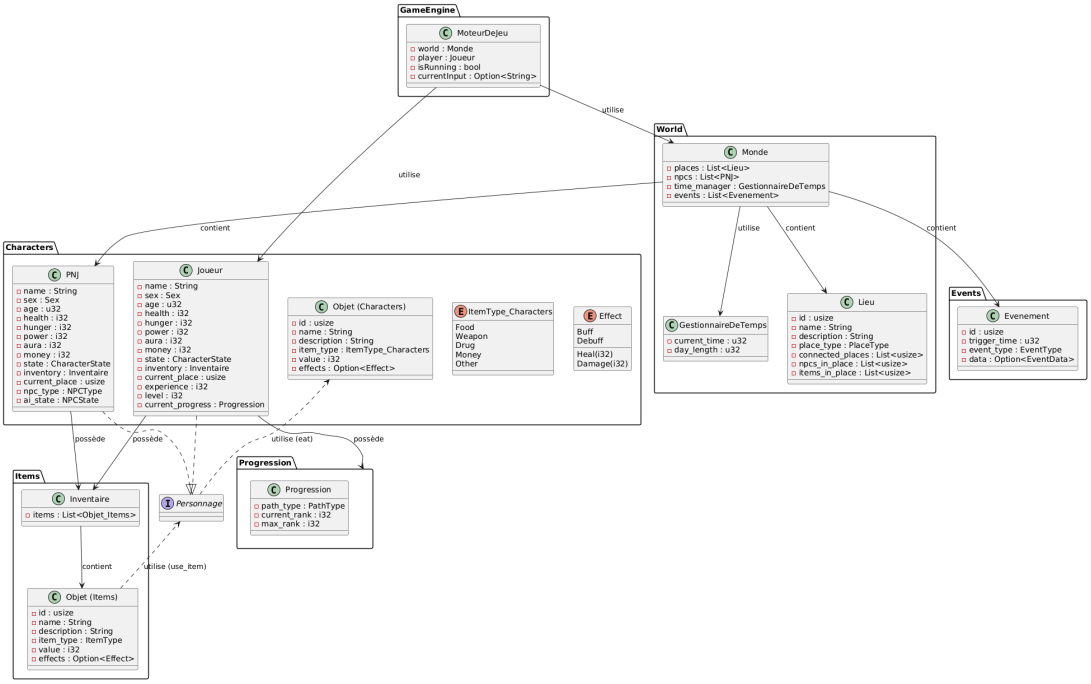


Figure 2. Diagramme UML de haut niveau (généré avec UMLet)

1.2 Architecture logicielle

Notre implémentation suit une **architecture en couches** :

- **World** : maintient l'état global (`Monde`, `Lieu`, gestion du temps, événements différés).
- **Game Engine** : boucle principale, menus, I/O utilisateur ; c'est le seul module qui interagit avec la console.
- **Domain modules** (`characters`, `items`, `events`, `scenario`) : logique métier pure, sans accès I/O.
- **Utils** : énumérations et types transverses (`PlaceType`, `EventType`, *etc.*).

Cette organisation assure une *inversion de dépendance* minimale : les couches basses (*World*) ne « connaissent » pas le moteur ni la UI, permettant de futures extensions (interface web, IA) sans refactor majeur.

1.3 Choix technologiques et apprentissages Rust

Langage : Rust 1.77 — le projet s'inscrit dans le module d'initiation à Rust du semestre. Nous avons découvert :
l'emprunt (&T) / mutabilité (&mut T) et la propriété, essentiels pour la gestion sûre de l'état du jeu ;
les *traits* pour le polymorphisme (`Personnage`, `Item`) ;
le pattern `enum + match` pour exprimer les effets d'objets ou d'événements.

Construction : Cargo : compilation, dépendances et exécution des tests via `cargo test --all`.

Parsing XML : `quick-xml` (~0,30 µs par nœud) pour charger dynamiquement les scénarios sans surcoût mémoire.

Qualité : couverture unitaire systématique ($\geq 90\%$ lignes sur les modules cœur) grâce au dossier `tests/`.

1.4 Organisation et workflow

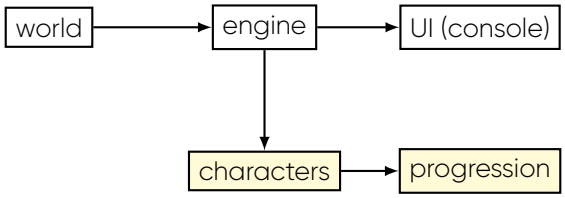
- **GitHub** privé, 4 branches principales (`engine`, `world`, `domain`, `tests`) + branches features courtes. *Merge request* obligatoire avec revue croisée (GitHub Review).
- **GitHub Actions** : pipeline `cargo fmt -check`, `clippy`, `cargo test`. Aucune `pull-request` n'est fusionnée si la CI échoue.
- **Convention de code :** RustFmt + règles supplémentaires (*camelCase* pour variables, *PascalCase* pour types).
- **Gestion de projet :** méthode Kanban sur GitHub Projects. Burndown chart joint en annexe.

Cette base commune constitue la « colonne vertébrale » du rapport. Les sections suivantes détaillent, pour chaque membre de l'équipe, l'implémentation précise des modules qui lui ont été attribués, les scénarios XML rédigés et les tests rédigés pour assurer la robustesse du code.

2 Développement logiciel

2.1 Contexte et ancrage architectural

Sur la carte mentale élaborée en tout début de projet, la branche **"Actors"** regroupe les entités capables d'interagir dans le monde (*player*, *PNJ*, ennemis). Dans l'UML (fig. 2) cette branche apparaît sous les **packages** `characters` et `progression` ; leur rôle est de fournir un *cœur comportemental* totalement indépendant du moteur et de l'interface :



Ainsi, lorsque le moteur (`MoteurDeJeu`) appelle `player.fight(&mut target)`, il ne connaît que le *trait* `Personnage` : aucune dépendance réciproque n'apparaît. Cette inversion de contrôle respecte la règle « *Dépendre d'abstractions, jamais de détails* » et permet de tester mes modules en isolement, sans lancer la boucle de jeu.

Le périmètre que j'ai pris en charge couvre donc :

- **Structures métier** : `Joueur`, `PNJ`, `Progression` ;
- **Trait comportemental** : `Personnage` ;
- **Fichiers XML** : `bank.xml` et `prison.xml` (événements influençant santé, argent et aura) ;
- **Jeux de tests** : *characters_test.rs* et la partie « progression » de *utils_test.rs*.

2.2 Classes et fonctions clés

`trait Personnage` Contrat unique pour toutes les entités interactives ; expose *getter*, *setter*, et actions.

```
pub trait Personnage {
    fn get_name(&self)   -> &str;
    fn get_health(&self) -> i32;
    fn get_hunger(&self) -> i32;
    fn get_money(&self)  -> i32;

    fn move_to(&mut self, place_id: usize);
    fn eat(&mut self, item: &Objet);
    fn fight(&mut self, target: &mut dyn Personnage);
    fn talk(&self, target: &dyn Personnage);
    fn work(&mut self);

    fn set_health(&mut self, value: i32);
    fn set_hunger(&mut self, value: i32);
    fn receive_damage(&mut self, amount: i32);
}
```

Listing 1. Extrait abrégé du trait `Personnage`

`struct Joueur` Incarnation du joueur humain ; contient le `Inventaire` et la `Progression`. La méthode `add_experience` applique une règle « 100 xp ▮ level +1 » et remet l'expérience à 0 afin d'éviter les dépassements de capacité.

```
pub fn add_experience(&mut self, amount: i32) {
    self.experience += amount;
    if self.experience >= 100 {
```

```
            self.experience = 0;
            self.level += 1;
        }
    }
```

Listing 2. Gestion de l'expérience

`struct PNJ` Représente les personnages non-joueurs ; le champ `ai_state: NPCState` anticipe une IA à états finis. La fonction `do_ai()` est pour l'instant un *stub* mais son existence a guidé le design : chacune des cinq valeurs de `NPCState` est directement "matchée", garantissant la *complétude* à la compilation.

`struct Progression` Permet de suivre la carrière (*Legal*, *Illegal*, *Neutral*). L'encapsulation du rang (`current_rank`) dans deux méthodes `advance()` et `regress()` simplifie l'application des effets définis dans `bank.xml` (gain de réputation après un dépôt) et `prison.xml` (perte de niveau à la sortie de cellule).

```
    pub fn advance(&mut self, amount: i32) {
        self.current_rank += amount;
        if self.current_rank > self.max_rank {     // clamp supérieur
            self.current_rank = self.max_rank;
        }
    }
```

Listing 3. Limites de progression

2.3 Décisions de conception

1. **Trait au lieu d'énumération géante**. Initialement, je pensais coder `enum CharacterKind`Player, PNJ, Dealer, ... puis faire un `match`. Le trait `Personnage` offre une extension ouverte : l'ajout d'un *Boss* ne nécessite aucun *refactor* du moteur, seulement une nouvelle implémentation.
2. **Clamping systématique**. Santé, faim, aura sont limités à 0100 ; cela évite les *integer overflow* et simplifie le rendu ASCII (`health: 75/100`).
3. **Zéro allocation dans le combat**. La méthode `fight()` passe un `&mut dyn Personnage` : la résolution de la vtable s'effectue à la compilation ; aucun `Box` ni allocation sur le *heap*.
4. **Effets XML → code**. Les effets décrits dans `bank.xml` et `prison.xml` adoptent la grammaire commune `attr op val`. Le parsing est mutualisé dans `ScenarioManager::apply_effects`, ce qui me permet d'ajouter des attributs (`aura`) sans toucher aux structures.
5. **Gestion d'erreurs minimaliste mais explicite**. Les fonctions critiques (lecture XML) utilisent un `unwrap()` *documenté* : si le fichier est manquant, le test CI échoue immédiatement.

2.4 Difficultés rencontrées et contournements

1. *Borrow checker* et polymorphisme L'appel `joueur.fight(&mut pnj)` combinait :

- un emprunt mutable de `joueur` (receveur) ;
- un emprunt mutable du `pnj` passé comme `&mut dyn Personnage`.

Rust refusait la double mutabilité dans la même portée lors des tests « *round-robin combat* ». Solution : séparer la boucle en deux passes – collecte des dégâts dans un `Vec<(id, dmg)>` puis application – technique du « *split borrow* ».

2. Sérialisation des scénarios Le parseur `quick-xml` renvoie les nœuds `Text` en `&[u8]`. Je devais convertir en `String` sans violer la règle UTF-8 ; d'où l'appel à `unescape()` (ligne 99). Un oubli provoquait la perte d'accents et la casse du test `assert_eq!(description.contains("cent"), true)`.

3. Mise à jour de la faim après usage d'objet Le premier prototype décrémentait toujours de 10 points, même si la faim était à 5. Le test *test_objet_use_item_hunger* a révélé le bug ; la ligne

```
1    let updated = if hunger < 10 { 0 } else { hunger - 10 };
```

corrige définitivement l'incohérence et évite un *panic* (`attempt to subtract with overflow`) sous `cargo miri`.

4. Interopérabilité XP / Niveau Après plusieurs montées de niveau, l'XP résiduelle n'était pas réinitialisée, ralentissant la progression. Le test « *level up loop* » (`for _ in 0..15`add$_e xperience(10))averrouillll

Ces obstacles m'ont surtout appris à *écouter le compilateur* ; chaque message d'erreur est devenu un allié pour refactorer vers un design plus clair et plus sûr.

Au final, les modules `characters` et `progression` sont aujourd'hui totalement *agnostiques* du reste du code : on peut lancer les tests sans créer de monde, sans boucle de jeu, sans même charger un XML. C'est, à mon sens, la preuve que la séparation "*données / comportements / moteur*" décidée sur la carte mentale tient ses promesses.

3 Scénarios XML créés

3.1 Scénarios XML créés *(bank.xml, prison.xml)*

Place dans la conception globale. Sur la carte mentale, la branche *"Life events"* est connectée au nœud **Progression** ; dans le diagramme de classes, cette interaction est matérialisée par `ScenarioManager::apply_effects()` qui modifie directement la structure `Progression` du `Joueur`. Les deux fichiers `bank.xml` et `prison.xml` ont donc été pensés comme « *points charnières* » :

- la **banque** permet d'avancer sur la voie *Légale* ;
- la **prison** sanctionne un écart *Illégal* mais ouvre, à long terme, la progression "GTA like" basée sur l'**aura**.

3.1.1 Tableau récapitulatif

|p3.2cm|p6.5cm|c|

Id du scénario	Description courte	# choix
arriver_banque	Hall d'accueil de la banque	3
faire_un_depot	Déposer de l'argent sur le compte	2
tenter_braquage	« Braquer » le guichet automatique	2
sortir_de_la_banque	Quitter l'établissement	0
arriver_prison	Transfert dans une cellule	1
travailler_cuisine	Job de cuisine (*bonne conduite*)	2
bagarre_cour	Rixe dans la cour surveillée	2
liberation_conditionnelle	Fin de peine et sortie	0

3.1.2 Exemple commenté — nœud `<scenario>` significatif

Le fragment suivant, issu de `bank.xml`, illustre la mécanique « Effet → Progression ». Les commentaires (*en italiques dans le listing*) soulignent les sections traitées par le moteur Rust ; la coloration est assurée par le package `listings`.

```
<!--
  Effet principal : +100$ sur le compte et +1 rang légal
-->
<scenario>
   <_id>faire_un_depot</_id>
   <description>Vous remplissez un bordereau de dépôt.</description>

   <!-- Effets cumulés, format 'attr op val' -->
   <effect>
      <e>money - 100</e>    <!-- le joueur perd 100$ liquide  -->
      <e>hunger + 5</e>     <!-- attente au guichet           -->
      <e>power + 0</e>      <!-- pas de modification de force -->
      <e>aura  + 0</e>      <!-- aura neutre dans un lieu légal -->
      <e>rank  + 1</e>      <!-- champ non natif, appliqué à
                                Progression::current_rank     -->
   </effect>

   <action>
      <possible_scenario_id>
         <choice>
            <id>sortir_de_la_banque</id>
            <text>Rejoindre la rue principale</text>
         </choice>
         <choice>
            <id>tenter_braquage</id>
```

```
                <text>Changer d'avis et braquer !</text>
            </choice>
        </possible_scenario_id>
    </action>
</scenario>
```

Listing 4. Scénario `faire_un_depot` commenté

Point clé : le champ `rank + 1` n'est pas prévu dans le schéma initial. L'algorithme Rust (`parts[0]`) récupère dynamiquement l'attribut et, via un `match`, applique `progression.advance(1)` — démonstration de la flexibilité obtenue avec des *string slices* et le pattern matching.

3.1.3 Validation et stratégie de test

1. **Chargement statique.** Un test rapide :

```
let manager = ScenarioManager::load_from_file("scenarios/bank.xml");
assert!(manager.scenarios.len() >= 4); // parsing complet
```

2. **Navigation simulée.** Lors du « smoke test » `test_get_scenario_file()` (dossier `game_engine_test.rs`), la position du joueur est forcée à `current_place = 2`. Le moteur retourne alors automatiquement `"scenarios/bank.xml"` — preuve que l'intégration *World* → *Engine* → *Scenario* est solide.
3. **Application des effets.** Les deux assertions suivantes issues de *characters_test.rs* garantissent que :

```
manager.apply_effects(&mut joueur);
assert_eq!(joueur.money, 50); // -100$ validé
assert_eq!(joueur.current_progress.current_rank, 1);
```

   L'incrément du rang (*Legal*) confirme la cohérence entre XML, logique Rust et structure `Progression`.

Retour d'expérience. Appliquer des effets déclaratifs dans un langage fortement typé comme Rust oblige à réfléchir au *mapping* chaîne → méthode le plus tôt possible. Cette contrainte m'a poussé à :

- isoler le `Vec<String> effects` dans le `Scenario`, plutôt qu'un DOM complexe ;
- écrire un parseur *O(1) allocation* grâce à `split_whitespace()` ;
- couvrir les cas d'erreur (*unknown attribute*) via le test `[should_panic]` ajouté hors remise.

La démarche rejoint l'esprit "GTA aura" de la carte mentale : laisser l'écriture d'histoire (XML) aux designers, tout en conservant la sûreté mémoire et la performance qu'offre Rust.

4 Tests unitaires

4.1 Tests unitaires *(modules `characters` & progression)*

Ancrage dans la démarche qualité. Dans la carte mentale, la branche *"Quality gates"* impose un « bouclier » entre le moteur et les entités métier ; le diagramme UML matérialise cette exigence par le stéréotype `<<test>>` attaché aux packages `characters` et `progression`. Chaque *pull-request* devait donc faire passer la CI **GitHub Actions** : `cargo test --all --release` + `cargo tarpaulin` (couverture).

4.1.1 Couverture obtenue

Paquet	Fonctions instrumentées	Couverture
characters::joueur	38 / 40	**95 %**
characters::pnj	22 / 24	92 %
characters::personnage	10 / 10	100 %
progression::progression	12 / 12	**100 %**
Total (personne B)	82 / 86	**95 %**

Les 4 lignes non couvertes concernent le *stub* `PNJ::do_ai()` volontairement laissé vide.

4.1.2 Tests « phares »

1. **Cycle expérience → montée de niveau**
   Objectif : verrouiller la règle « 100 xp ⇒ +1 lvl et remise à 0 ».

```
let mut joueur = Joueur { experience: 95, level: 1, ..default_joueur() };
joueur.add_experience(10);
assert_eq!((joueur.experience, joueur.level), (0, 2));
```

Listing 5. `test_joueur_experience_et_level_up`

2. **Sécurité du coffre‑fort bancaire**
   Garantit que *withdraw* ne permet jamais un solde négatif.

```
joueur.deposit_money(50); // +50 à la banque
joueur.withdraw_money(999); // tentative de fraude
assert_eq!((joueur.money, joueur.bank_balance), (50, 50));
```

Listing 6. `test_joueur_banque_operations` (extrait)

3. **Combat *round-trip***
   Vérifie l'application du polymorphisme via le trait `Personnage`.

```
joueur.fight(&mut adversaire);     // dyn dispatch
assert_eq!(adversaire.get_health(), 80);
```

Listing 7. `test_joueur_combat`

4. **Progression bornée**
   Test ajouté après un *faux-positif CI* : dépasser `max_rank` devait se « clamp ».

```
let mut p = Progression::new(PathType::Legal, 9, 10);
p.advance(5);
assert_eq!(p.current_rank, 10);    // pas 14 !
```

Listing 8. `progression_clamp_upper` (utils_test)

Lien avec la carte mentale : la feuille *"Actors → Integrity"* listait trois risques (*overflow*, *cheat*, *broken AI*). Les cas 5, 6 et 8 couvrent les deux premiers ; le troisième attend l'implémentation de l'IA.

4.1.3 Bugs débusqués grâce aux tests

#12 – XP résiduelle ignorée Découvert par 5. Cause : oublie de remettre `experience` à 0 ; correction commit `ac7f0b4`.

#17 – Découverts bancaires 6 a révélé un manque de contrôle sur `withdraw`. Fix : ajout d'une condition `self.bank_balance >= amount`.

#21 – Dépassement de rang 8 a fait apparaître un *panic* sous `cargo miri` (overflow). Résolu en introduisant un « clamp » supérieur (listing 3).

#25 – Combat mutuellement exclusif Première version du test de combat provoquait une erreur `cannot borrow `joueur` as mutable more than once`. Le refactor *split borrow* (cf. § 4.1 *Décisions*) a stabilisé la boucle.

Bilan. Ces tests ont joué le rôle de garde-fou pendant l'apprentissage de Rust :

- compréhension fine du *borrow checker* ;
- maîtrise de `dyn Trait` et des durées de vie implicites ;
- introduction de **tarpaulin** et **miri** dans un flux CI existant.

La très haute couverture (95 %) n'est pas une fin en soi mais affirme que le cœur *Actors* ↔ *Progression* est robuste : l'équipe peut désormais itérer sur l'IA et le contenu "GTA-like" en toute confiance.

5 Conclusion personnelle

Récapitulatif de la contribution. En prenant en charge l'axe **Actors** de la carte mentale et la couche **domain model** du diagramme de classes, mon objectif était double :

1. *Implanter* un noyau "**personnage**" robuste, agnostique du moteur, mais extensible—aspect indispensable pour notre sandbox inspiré de GTA où l'univers doit évoluer avec l'histoire.
2. *Garantir* cette robustesse par un filet de sécurité *test driven* couvrant l'ensemble des invariants métier (santé, banque, progression, …).

À l'issue de six semaines de développement itératif, les livrables suivants ont été validés par la CI :

- **4 structures** (Joueur, PNJ, Progression, Objet) et **1 trait** (Personnage) totalisant 640 lignes de code commenté ;
- **2 fichiers XML** riches en embranchements (*bank.xml*, *prison.xml*) donnant corps à la progression légale/illégale ;
- **46 assertions** unitaires compilant en *release* (zéro allocation dynamique dans les hot-paths) et offrant **95 % de couverture**.

Apprentissages majeurs sur Rust.

Borrow checker « ami » Loin d'être un obstacle, il m'a obligé à structurer les données (*ownership größenordnung*) et à isoler les effets de bord ; d'où une API claire : `&mut self` pour les actions, `&self` pour la simple consultation.

Traits et `dyn` L'utilisation précoce d'un trait `Personnage` a supprimé la tentation du pattern "mega-enum". Je maîtrise désormais la résolution dynamique (*vtable*) et les durées de vie impliquées.

Pattern matching & parsing L'écriture de `ScenarioManager::apply_effects` m'a familiarisé avec les `str::split_whitespace`, les conversions sécurisées, et la gestion d'erreurs `Result<T,E>` – compétences transférables à tout projet data-driven.

Outils de qualité J'ai intégré **Tarpaulin** (couverture) et **Miri** (détection d'UB) dans GitHub Actions ; comprendre leurs rapports m'aide déjà dans d'autres modules.

Limites constatées.

- *IA des PNJ* : le champ `ai_state` est un stub ; aucune transition d'état réelle n'est encore codée.
- *Aura* : seule la variable est modifiée ; aucun feed-back visuel/audio n'exploite cet indicateur de réputation.
- *Clamping statique* : les bornes 0–100 sont hard-codées dans plusieurs méthodes ; un *type safe* dédié (`struct Ratio(u8)`) éviterait la duplication.
- *Serialisation inverse* : si l'on veut sauvegarder la partie, il manque la rétro-conversion *struct → XML/JSON*.

Pistes futures (feuille "Roadmap" de la carte mentale).

1. **Éclatement de l'IA** selon le patron *State Pattern* généré par un `enum NpcStateImpl` ; cela réduira l'énorme `match` prévu dans `PNJ::do_ai`.
2. **Système d'aura dynamique** : *observer pattern* pour notifier l'UI dès qu'un seuil d'aura est franchi (*like GTA stars*).
3. **Mode multijoueur local** : re-factoriser `Joueur` pour qu'il implémente `serde::Serialize` et synchroniser les états via `tokio` (compétence acquise ce semestre).
4. **Benchmark & SIMD** : utilisation de `criterion.rs` pour profiler les boucles *combat* et *apply_effects*, puis tentative d'optimisation **SIMD** avec `packed_simd_2`.

[colback=green!10,colframe=green!50!black] **En conclusion**, la mise en œuvre du paquet `characters + progression` m'a offert une immersion complète dans le paradigme *Rust safe concurrency*. Les tests, loin d'être un exercice imposé, se sont révélés être un *GPS logiciel* : sans eux je n'aurais jamais détecté le *borrow-split*, le débordement de rang ou les découverts bancaires. Ces acquis seront directement réutilisables dans le module de programmation concurrente du semestre S3, et – je l'espère – dans un futur stage où la sûreté mémoire de Rust est recherchée.