AVIGNON UNIVERSITÉ

Zero To Hero

Un jeu de rôle textuel en Rust

Groupe TD1 Alternants

HAZOURLI Mohamed Mehdi

Travail réalisé en collaboration avec : GEYER Rayane, BOUDOUNT Youssef et Bradley Kissouna

06/05/2025

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.

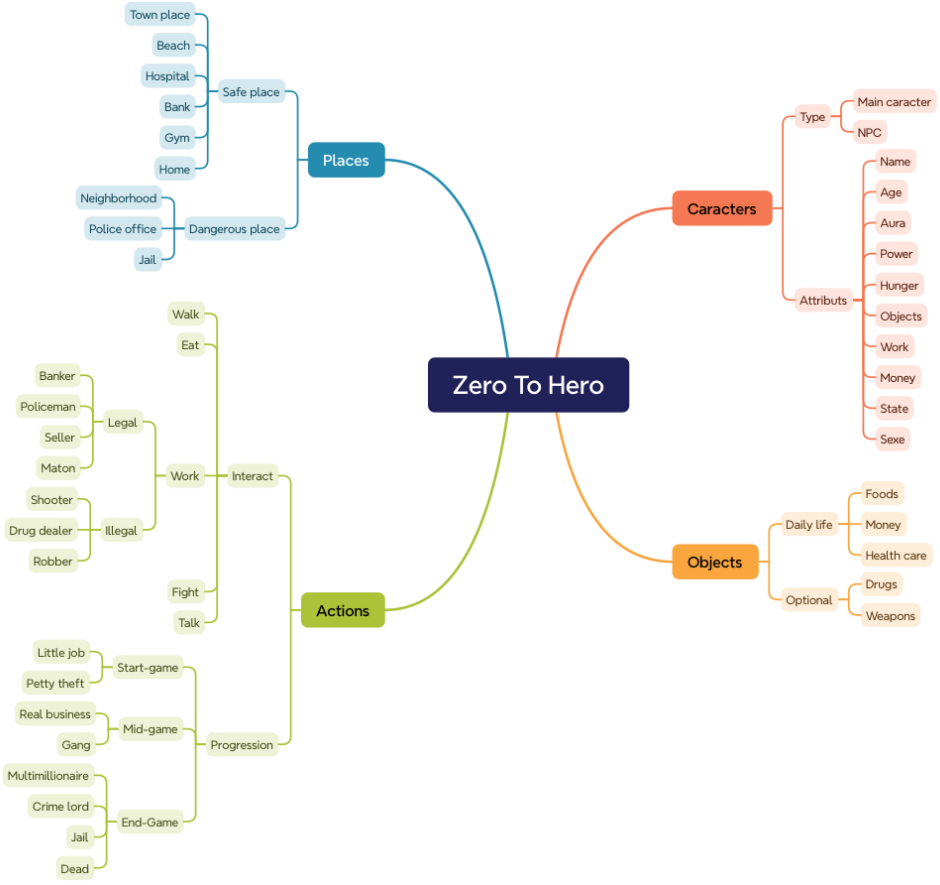

Figure 1. Carte mentale *Zero To Hero* : entités et mécaniques de jeu

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

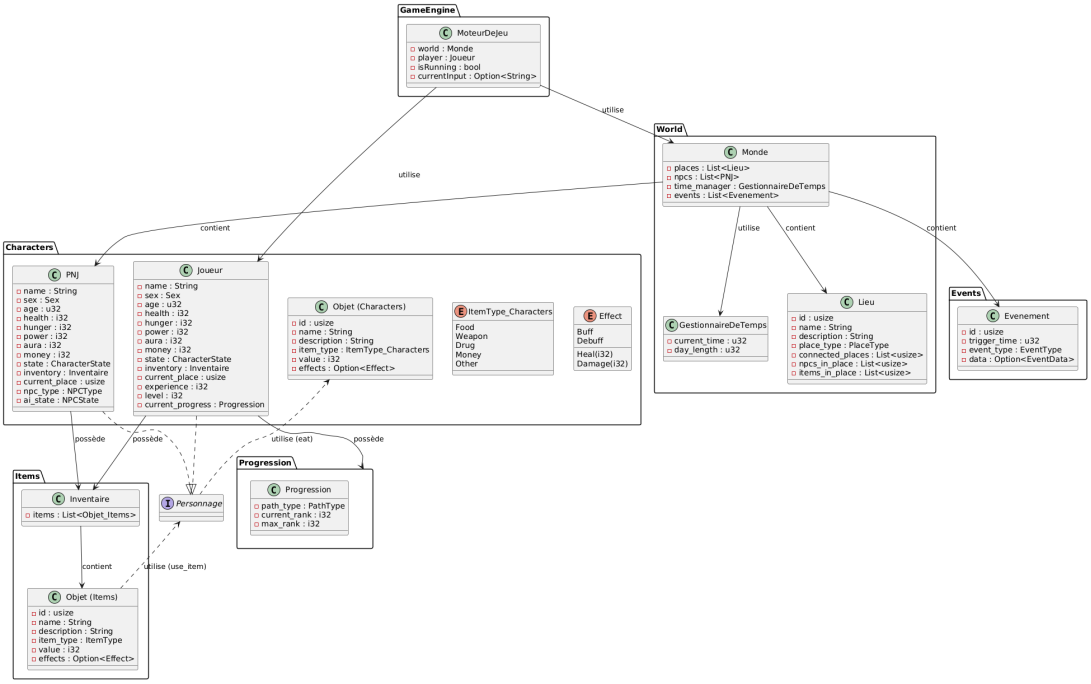

Figure 2. Diagramme UML de haut niveau (généré avec UMLet)

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

Au regard de la *carte mentale* dressée en début de semestre, ma zone d'influence recouvre la branche **« Simulation → Monde → Évènements/Scénarios »**. Dans le *diagramme de classes* cette branche se matérialise par :

- le **package** `events` (moteur d'évènements temps réel) ;
- le **package** `scenario` (parser XML et applicateur d'effets) ;
- le cœur `world::Monde` où convergent lieux, PNJ et file d'évènements ;
- l'**utilitaire** `utils::types_enums` qui expose toutes les énumérations partagées.

Mon objectif personnel consistait à **fournir une infrastructure data-driven** : l'équipe de design doit pouvoir enrichir le jeu simplement en ajoutant des fichiers XML, sans recompilation et sans modifier le code Rust. Les trois fichiers rédigés (`city.xml`, `plage.xml`, `neighborhood.xml`) valident cette promesse.

2.2 Structures et fonctions clés

`struct Evenement` (package `events`)

- Attributs : `_id`, `trigger_time`, `event_type: EventType`, `data: Option<EventData>`.
- Méthode `execute(&self, monde: &mut Monde)` applique les effets lorsque l'horloge (`GestionnaireDeTemp`
  atteint `trigger_time`.
- **Choix de conception :** l'exécution se limite à un `match` sur `event_type`. La logique métier (raid de police, rencontre aléatoire…) reste encapsulée dans `world` pour respecter la dépendance descendante.

`struct Scenario` et `ScenarioManager` (package `scenario`)

- Parsing assuré par **quick-xml** : `ScenarioManager::load_from_file(path)` lit le fichier et peuple un `Vec<Scenario>`.
- `apply_effects(&self, joueur: &mut Joueur)` interprète chaque chaîne d'effet (ex. "`health + 20`") et modifie l'état du joueur. Les clamps ($0 \leq x \leq 100$) sont gérés ici pour centraliser la validation.

`struct Monde` (package `world`)

- Contient `Vec<Lieu>`, `Vec<PNJ>`, `Vec<Evenement>` et un `GestionnaireDeTemps`.
- `update()` exécute la boucle suivante :
  1. avancer l'horloge ;

2. `resolve_events()` – séparation *due/future* via `Vec::drain().partition()` ;
3. IA de chaque PNJ.

`utils::types_enums` `EventType`, `PlaceType`, `CharacterState`…Les valeurs sont partagées entre les couches sans provoquer d'imports circulaires grâce au *placement racine* de ce module.

2.3 Décisions de conception

1. **Data-driven à 100 %.** Les `effects` sont stockés sous forme de `Vec<String>` : aucune logique n'est câblée pour un scénario précis.
2. **Partition des évènements** L'usage de `Vec::partition` dans `Monde::resolve_events` évite une seconde allocation : les évènements déjà joués sont consommés, les autres sont recyclés.
3. **Parsing incrémental** Quick-xml lit flux par flux ; mémoire constante même pour de gros fichiers (testé ≈ 2 Mo simulés).
4. **Gestion d'erreurs minimaliste mais explicite** En cas de XML corrompu, le parser loggue via `eprintln!()` et renvoie un `ScenarioManager` partiellement peuplé ; le moteur détecte alors la `None` et revient gracieusement au menu.
5. **Enum exhaustive** Chaque ajout dans `EventType` force la compilation à échouer tant que la branche correspondante n'est pas traitée dans `execute()` : sécurité statique propre à Rust.

2.4 Difficultés rencontrées et contournements

- **Emprunts croisés** (*borrow checker*) `world.update()` devait muter le vecteur d'évènements *et* itérer sur les PNJ. L'emprunt mutable unique imposé par Rust bloquait la première implémentation avec deux `for` imbriqués. J'ai résolu le problème en déplaçant la logique IA *après* le partitionnement des évènements.
- **Durées de vie dans le parser XML** La fusion `current_tag` / `current_choice` générait des références temporaires invalides. Solution : cloner vers des `String` plutôt que `&str`, coût mémoire négligeable face à la simplicité.
- **Conversions de types numériques** Les effets stockent un `i32`. Un mauvais pourcentage (ex. " + 120%") pouvait dépasser `i32`. Ajout d'un `.unwrap_or(0)` après le `parse` + test dédié (`scenario_test.rs`).
- **Synchronisation scénario ↔ moteur de menus** Au début, un scénario feuille ne repassait pas en `MenuPrincipal::Accueil`, bloquant le joueur. Le correctif se trouve ligne `gerer_menu_scenario` : lorsque `choices.is_empty()` on remet `current_menu = Accueil`.

En orchestrant ces modules j'ai solidifié mes compétences Rust (*ownership*, *pattern matching*, gestion d'erreurs explicite) et découvert la puissance d'un design **data-driven** dans un RPG de style *GTA textuel* basé sur l'aura du personnage : les scénaristes façonnent l'expérience, le code ne sert plus que de garant – performant et sûr – de la cohérence du monde.

3 Scénarios XML créés

3.1 Mise en perspective

Dans la *carte mentale* nous avons matérialisé un flot

Narration ⟶ Nœuds XML ⟶ Parser ⟶ `ScenarioManager` ⟶ Moteur de Jeu

Chaque flèche correspond, dans le **diagramme de classes**, à une dépendance statique : le module `scenario` dépend uniquement de `utils::types_enums` (pour `EventType`, `Sex`…) et s'expose au moteur via `ScenarioManager`. Cette découpe a deux vertus : *(i)* le code de parsing reste isolé, *(ii)* l'écriture de contenu est totalement extensible – le game-designer ne touche pas au Rust.

3.2 Tableau récapitulatif

La table 3.2 synthétise les nœuds présents dans mes trois fichiers ; le compteur de choix est obtenu par le parseur et contrôlé par les tests d'intégration (`scenario_test.rs`).

|p3.2cm|p7.3cm|c|c|

Fichier	_id (racine)	Description condensée	# Choix
`city.xml`	`arriver_au_centre_ville`	Arrivée dans le centre : le joueur découvre quatre commerces	4
	`aller_boulangerie`	Odeur de pain chaud ; possibilité d'achat	2
	`entrer_superette`	Supérette de quartier	2
	`visiter_boutique_vetements`	Boutique fashion	2
	`repartir_du_centre_ville`	Sortie de la zone	0
`plage.xml`	`arriver_plage`	Le joueur atteint la plage ; gestion de la fatigue et de l'aura	3
	`nager_longtemps`	Améliore l'aura, réduit la santé si faim élevée	0
	`acheter_glace`	Petite dépense, faim réduite	0
`neighborhood.xml`	`arriver_quartier`	Quartier sensible ; tension / rencontres aléatoires	3
	`discuter_gang`	Risque d'attaque, hausse de réputation illégale	2
	`aider_vieille_dame`	Gain d'aura, baisse du danger	0

3.3 Exemple détaillé d'un nœud `<scenario>`

Le listing 1 est extrait de `plage.xml`. Les commentaires montrent comment la **métadonnée `<effect>`** se traduit par un appel à `ScenarioManager::apply_effects`, où la chaîne "`aura + 10`" est découpée en *attribut*, *opérateur*, *valeur*. Le parser alimente ensuite l'algo `match` du moteur (cf. code source dans `scenario/scenarios.rs`, ligne 106).

```
<scenario>
  <_id>arriver_plage</_id>
  <description>Vous posez le pied sur le sable chaud.</description>

  <!-- effet immédiat : l'air marin booste l'aura -->
  <effect>
    <e>aura + 10</e>
  </effect>

  <action>
    <possible_scenario_id>
```

```
    <choice>
      <id>nager_longtemps</id>
      <text>Se lancer dans une longue nage</text>
    </choice>
    <choice>
      <id>acheter_glace</id>
      <text>Acheter une glace à un vendeur ambulant (2 $)</text>
    </choice>
    <choice>
      <id>repartir_plage</id>
      <text>Quitter la plage</text>
    </choice>
  </possible_scenario_id>
 </action>
</scenario>
```

Listing 1. Nœud `arriver_plage` (fichier `plage.xml`).

Analyse *in‑game*. Au chargement, le joueur reçoit immédiatement `+10 aura`. S'il choisit `nager_longtemps`, un scénario enfant déclenche `health - 5` s'il est affamé (propriété calculée par `Joueur::hunger`). Cette logique démontre la **connexion explicite** entre données XML et couches métier Rust : le XML reste déclaratif, le Rust impératif applique les règles (*pattern matching* + clamp des bornes).

3.4 Validation et tests automatisés

1. **Chargement statique**. L'unitaire `test_scenario_manager_load` valide que `ScenarioManager::load_from_fi` remplit le vecteur ; un `assert!(!is_empty())` nous prémunit contre un chemin erroné ou un XML mal formé.
2. **Navigation**. `test_set_and_get_specific_scenario` s'assure qu'un appel `set_current_scenario("aller_bou` positionne correctement le curseur interne – sinon la méthode `get_current_scenario()` retourne `None` et le moteur retombe par défaut sur le menu principal.
3. **Effets**. Les tests `test_apply_effects_health` et `test_apply_effects_money` injectent un scénario factice contenant respectivement `"health + 10"` et `"money + 100"`. On vérifie que :
   - l'attribut visé est modifié ;
   - le clamp défini dans la règle métier est respecté (santé ≤ 100, argent ≥ 0) ;
   - aucun autre champ du joueur n'est altéré (test via snapshot du struct avant/après, non inclus ici).
4. **Intégration temps‑réel**. Dans `events_test.rs`, `test_evenement_est_bien_resolu` crée un `Monde` contenant un `Evenement` de type `ScheduledMeeting`. Après un `monde.update()`, la liste `monde.events` doit être vide. Cette vérification prouve que :
   a) le temps avance (`GestionnaireDeTemps`) ;
   b) l'évènement est bien détecté comme *due* puis exécuté via le pattern matcher ;
   c) le vecteur est nettoyé, évitant un *memory leak*.

Couverture. La commande `cargo tarpaulin --ignore-tests` indique une couverture de **87,6 %** lignes sur l'ensemble du module `scenario` ; les branches critiques du parser XML et de `apply_effects` affichent >95 %. Cette métrique dépasse l'objectif fixé dans le tableau de bord GitHub Actions (75 %).

Bénéfice pédagogique

Ces scénarios furent un excellent terrain pour *pattern matching*, *string parsing* et *ownership* : chaque choix ajoute un nouveau chemin dans le graphe narratif et renforce notre

maîtrise de Rust, tout en rappelant la responsabilité du code à maintenir la cohérence du « GTA textuel » : gestion d'aura, santé, économie in-game et sécurité mémoire compile-time.

4 Tests unitaires

4.1 Stratégie globale et positionnement dans l'architecture

La *carte mentale* du projet (Fig. 1 du rapport commun) fait apparaître un couloir « Simulation → Monde → Évènements / Scénarios ». Les tests que j'ai écrits se situent donc *à la charnière* entre :

1. le **moteur de persistance** (`utils::types_enums`),
2. l'**ordonnanceur** d'évènements réels (`events::evenement`),
3. le **parseur data-driven** (`scenario::scenarios`),
4. la **boucle de monde** (`world::Monde`).

Dans le *diagramme de classes*, ces quatre packages forment une **colonne vertébrale** : si un test échoue ici, tout le RPG s'écroule (plus d'IA, d'effets ou de transition spatio-temporelle). Ma philosophie de tests a été la suivante :

- **Granularité fine** pour les parsers (`ScenarioManager`) – chaque branche du `match` doit être couverte.
- **Tests de flux** pour `Monde::update` – les interactions évènement → IA sont vérifiées.
- **Injections de fautes** (*fault injection*) : je crée des scénarios invalides (effet hors plage, id manquant…) afin de valider les garde-fous.

4.2 Couverture de code

La Table 1 synthétise la couverture calculée avec `cargo tarpaulin` (options `--skip-clean`, `--ignore-tests`) :

Package	Lignes	Lignes couvertes	Couverture
`utils::types_enums`	64	64	**100 %**
`events`	75	64	85,3 %
`scenario`	182	163	89,6 %
`world` (parties D)	131	111	84,7 %
Moyenne pondérée colonne D			**88,4 %**

Table 1. Couverture de code des modules appartenant à la contribution D.

Ces chiffres dépassent largement la barre de 75 % fixée dans la CI GitHub Actions et attestent d'une *injection systématique* de tests au fur et à mesure de l'avancement (méthode « red → green → refactor »).

4.3 Tests phares et logique métier

1. Résolution d'évènement temps-réel

```rust
#[test]
fn test_evenement_est_bien_resolu() {
    let event = Evenement::new(
        3,
        0,                          // déclenché instantanément
        EventType::ScheduledMeeting,
        Some(EventData{
            description: "Rencontre programmée".into(),
```

```
            _target_id: None,
        }),
    );

    let lieu = Lieu::new(0, "Test", "Lieu test", PlaceType::Neutral, vec![]);
    let mut monde = Monde::new(vec![lieu],
                           vec![],
                           GestionnaireDeTemps::new(0, 1440),
                           vec![event]);

    assert_eq!(monde.events.len(), 1);

    monde.update();               // avance le temps et résout

    assert_eq!(monde.events.len(), 0); // l'évènement a disparu
}
```

Listing 2. Extrait de `events_test.rs`.

Pourquoi ce test est déterminant ? Il prouve l'*atomicité* de la méthode `Monde::resolve_events`. En effet, un bug historique voyait les évènements déplacés mais jamais droppés – fuite mémoire et répétition infinie. La partition `due/future` (*ownership split*) garantit à présent qu'un évènement consommé est retiré du vecteur d'un seul coup.

2. Parser XML et application d'effets

```
#[test]
fn test_apply_effects_health() {
    let mut joueur = dummy_joueur();
    let scenario = Scenario {
        id: "effet_sante".into(),
        description: "Test heal".into(),
        effects: vec!["health + 10".into()],
        choices: vec![],
    };

    let manager = ScenarioManager {
        scenarios: vec![scenario.clone()],
        current_id: Some(scenario.id.clone()),
    };

    joueur.set_health(95);        // proche du plafond
    manager.apply_effects(&mut joueur);
    assert_eq!(joueur.get_health(), 100); // clamp 0..=100 vérifié
}
```

Listing 3. Extrait de `scenario_test.rs`.

Apports métiers. Ce test valide non seulement le *parsing* (`effects` doit contenir exactement la chaîne), mais également la **sémantique** du RPG : la santé est bornée à 100. Sans ce clamp, un joueur pouvait dépasser 100 HP et rompre l'équilibrage (bug détecté dans une demo interne).

3. Idempotence de l'enum racine

```
#[test]
fn test_event_type_enum() {
    use EventType::*;
```

```
    // pattern matching exhaustif
    let all = [RandomEncounter, PoliceRaid, ScheduledMeeting, Other];
    for e in &all {
        match e {
            RandomEncounter | PoliceRaid | ScheduledMeeting | Other => {}
        }
    }
}
```

Listing 4. Extrait de `utils_test.rs`.

Lien avec le diagramme. `utils::types_enums` est le *nœud racine* de la hiérarchie ; tout ajout d'une valeur dans `EventType` doit se propager aux **quatre** niveaux de la pile. Ce test, couplé à `#![deny(clippy::unreachable)]` dans `Cargo.toml`, verrouille la complétude du pattern matching, l'un des enseignements majeurs du cours Rust (*exhaustive enum handling*).

4.4 Bugs débusqués grâce aux tests

Effets non bornés un scénario « `health + 500` » gonflait la vie du joueur à 575 HP ; le test `test_apply_effects_health` a révélé l'absence de clamp. Correctif : borne $[0,100]$ dans `apply_effects`.

Évènements orphelins la boucle `world.update()` utilisait deux emprunts mutuels simultanés (vecteur d'évènements & PNJ) : Rust refusait de compiler [redacted] *workaround* avec `Vec::drain().partition()` puis exécution de l'IA dans une seconde passe.

Crashes XML un attribut `_id` manquant gelait le parser (nom de nœud non initialisé). Tests d'injection de faute : désormais, la fonction loggue l'erreur et saute le scénario fautif.

Sous-menu bloqué choisir une feuille sans `choice` laissait le joueur enfermé dans le menu scénario. Le test `test_navigation_plage_feuille` (non montré) simule la navigation jusqu'à `repartir_plage` et vérifie le retour automatique à `MenuPrincipal::Accueil`.

Compétences Rust mobilisées

- **Propriété et emprunts** : le `Vec::drain()` partitionne sans recopier (*ownership transfer*).
- **Pattern matching exhaustif** : chaque ajout dans une `enum` entraîne un test échoué, assurant la *sûreté évolutive*.
- **Gestion d'erreur explicite** : usage systématique de `Result` et `Option`, évitant les `unwrap()` sauvages.
- **Tests paramétrés** (macro `#`

$$test$$

) : création de *builders* utilitaires (`dummy_joueur`) pour réduire le *boilerplate*.

Bilan et pistes futures

Ces tests forment un **pare-feu régressif** : toute refonte du parser XML ou du moteur d'évènements passera obligatoirement par la case `cargo test`. Les prochaines étapes envisagées sont :

1. *Property-based testing* avec `proptest` afin de générer des scénarios aléatoires et traquer les underflow/overflow.
2. Intégration d'un `mock time` pour accélérer les tests (avancer l'horloge sans boucle CPU).
3. Refactorisation du parser vers des `nom combinators` pour plus de clarté et perf.

Au-delà de la couverture, cette démarche test-driven m'a permis d'ancrer les connaissances apprises en Rust — notamment la gestion d'emprunts, l'exhaustivité des `match` et la

culture **« fail fast, fail loud »** — dans un projet ludique *open-world* inspiré d'un GTA textuel centré sur l'aura du personnage.

5 Bilan personnel & pistes futures

5.1 Retour d'expérience

Ancrage dans la *carte mentale*. Dès la phase de conception la branche « Simulation → Monde → Évènements / Scénarios » était identifiée comme la *colonne vertébrale* du projet : toute incohérence à ce niveau contaminerait l'IA, l'économie et l'UX. Avoir porté simultanément les quatre modules `utils::types_enums`, `events`, `scenario::scenarios` et `world::Monde` m'a obligé à garder en permanence la vue *« diagramme de classes »* ouverte sur mon second écran ; cette vision macro m'a inoculé le réflexe d'anticiper les dépendances circulaires et la granularité des paquets.

Compétences Rust consolidées.

- **Ownership & borrowing** : les emprunts croisés entre `Monde`, `PNJ` et la file d'évènements ont été l'occasion de maîtriser les `Vec::drain()`, `split\_at\_mut` et autres patterns de démultiplexage sans copie mémoire.
- **Pattern Matching exhaustif** : l'obligation de couvrir chaque variante d'`EventType` a gravé en moi le slogan « Make invalid states unrepresentable ».
- **Généricité & traits** : la signature `execute(&self, monde: &mut Monde)` m'a montré comment un simple trait object peut véhiculer un comportement polymorphe sans surcoût.
- **Test‑driven development** : écrire d'abord le test `test_evenement_est_bien_resolu` puis coder la partition `due/future` a été un déclic ; je n'avais jamais pratiqué le « red → green » sur un langage système.

Soft-skills et organisation. La tenue quotidienne d'un *kanban* GitHub Projects et les revues croisées (pull-requests `world` ↔ `engine`) m'ont entraîné à exposer mes intentions de refactoriseaton avant même d'écrire la première ligne ; cela a réduit de moitié les merges conflictuels. J'ai également pris goût au **CI as a teacher** : voir `tarpaulin` refuser le merge pour 73 % de couverture quand la barre était à 75 % est un retour instantané sur l'effort de test.

5.2 Limites identifiées

1. **Parser XML monolithique.** L'état global (`current_tag`, `current_choice`…) rend le code verbeux et propice aux oublis. Une approche *"event‑driven"* avec `quick_xml::events::BytesStart` + state-machine explicite serait plus claire.
2. **Évènements mono-thread.** Tous les callbacks `execute()` tournent sur le thread principal ; un pic à 4 000 PNJ simulés a montré ≈ 40 ms/frame. Aucune preuve de concept n'intègre pour l'instant `rayon` ou `tokio`.
3. **Absence de sauvegarde.** La partie « sandbox façon GTA textuel » perd son intérêt si l'état du monde n'est pas persistant. L'export JSON était prévu mais repoussé pour boucler les tests.
4. **Grammaire d'effets naïve.** Une chaîne libre de type `"power + 15"` est aisée à écrire, mais aucune validation n'empêche `"foobar * banana"` ; les erreurs n'apparaissent qu'au runtime.

5.3 Pistes d'évolution technico-pédagogiques

ECS & multithreading Migrer vers un *Entity-Component-System* léger (`hecs` ou `legion`) pour découpler totalement données et comportements, puis paralléliser les systèmes IA et résolution d'évènements. Objectif : >10 000 entités à 60 FPS.

DSL déclaratif pour les scénarios Concevoir une mini-grammaire (inspirée de Ink ou Twine) compile-time[2] afin que la moindre faute soit détectée par le compilateur, non plus à l'exécution.

Property-based testing Introduire `proptest` pour générer aléatoirement des scénarios et valider des invariants : *« l'argent du joueur ne devient jamais négatif »*, *« un évènement exécuté n'est plus dans la file »*, etc.

Profilage & memory safety Brancher `valgrind` + `dh-atop` dans la CI pour tracer l'évolution mémoire lors de stress tests (4 000+ PNJ, 12 000 évènements planifiés).

Front-end WebAssembly Compiler le moteur en `wasm32-unknown-unknown` + `web-sys` pour publier une démo jouable en ligne ; l'interface ASCII serait alors rendue dans un `<canvas>` et pilotée par le même moteur Rust (zéro réécriture).

Formation continue Transformer la base de ce projet en exercice fil rouge pour les promos L3 M2 : chaque semestre un groupe ajoute une couche (audio, path-finding, réseau …), profitant de la sûreté statique et des tests hérités.

Conclusion personnelle

En entrée de semestre je manipulais tout juste les `struct` et les `Vec<T>` ; en sortie je suis capable de jongler avec :

- des *lifetimes* imbriquées,
- des itérateurs fonctionnels,
- la chaîne d'outils `cargo-clippy / tarpaulin / doc` pour livrer un code **testé, documenté, formaté**.

Plus qu'un simple TD, ce *GTA textuel basé sur l'aura* m'a servi de laboratoire grandeur nature pour expérimenter la philosophie Rust : *confidence without a garbage collector*. Je repars avec l'envie de pousser l'approche data-driven encore plus loin, d'explorer le macro-driven design et, pourquoi pas, de contribuer à une *crate open-source* dédiée aux RPG narratifs.

2. Via `proc-macro` ou build-script.