

Zero To Hero

Un jeu de rôle textuel en Rust

Groupe TD1 Alternants

GEYER Rayane

Travail réalisé en collaboration avec : BOUDOUNT Youssef, HAZOURLI Mohamed Mehdi et Bradley Kissouna

06/05/2025

Master Informatique

Parcours INGE DU LOGICIEL DE LA SOCIÉTÉ NUM (ILSEN)

UCE UCE Algorithmes et Modélisation Avancée

Responsable

Pierre Jourlin
Lahcène Belhadi

UFR
SCIENCES
TECHNOLOGIES
SANTÉ



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

Sommaire

Titre	1
Sommaire	2
1 Introduction	3
1.1 Présentation générale du projet	3
1.2 Architecture logicielle	4
1.3 Choix technologiques et apprentissages Rust	4
1.4 Organisation et workflow	5
2 Développement logiciel	5
2.1 Contexte	5
2.2 Structures et fonctions clés	5
2.3 Décisions de conception	5
2.4 Difficultés rencontrées	5
3 Scénarios XML créés	6
3.1 Tableau récapitulatif.	6
3.2 Exemple détaillé : <code>acheter_potion_soin</code> .	6
3.3 Validation et stratégie de tests.	7
4 Tests unitaires	7
4.1 Méthodologie et outillage	7
4.2 Couverture obtenue	8
4.3 Extraits de tests « phares »	8
1. Sécurité mémoire : saturation de faim.	8
2. Intégrité d'un inventaire après retrait.	9
3. Machine à états de Dialogue.	9
4.4 Bugs débusqués grâce aux tests	9
5 Bilan personnel & pistes d'évolution	9
5.1 Retour d'expérience	9
Cartographie mentale → Code concret.	9
Compétences acquises.	10
5.2 Limites rencontrées	10
5.3 Pistes d'évolution	10
1. Refactorisation « data-driven » complète.	10
2. Système de <i>buff/debuff</i> persistants.	10
3. Moteur de dialogues conditionnels.	10
4. Persist / Load de parties.	11
5. Ouverture multi-joueur asynchrone.	11
5.4 Projection personnelle	11

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 GTA¹. 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.

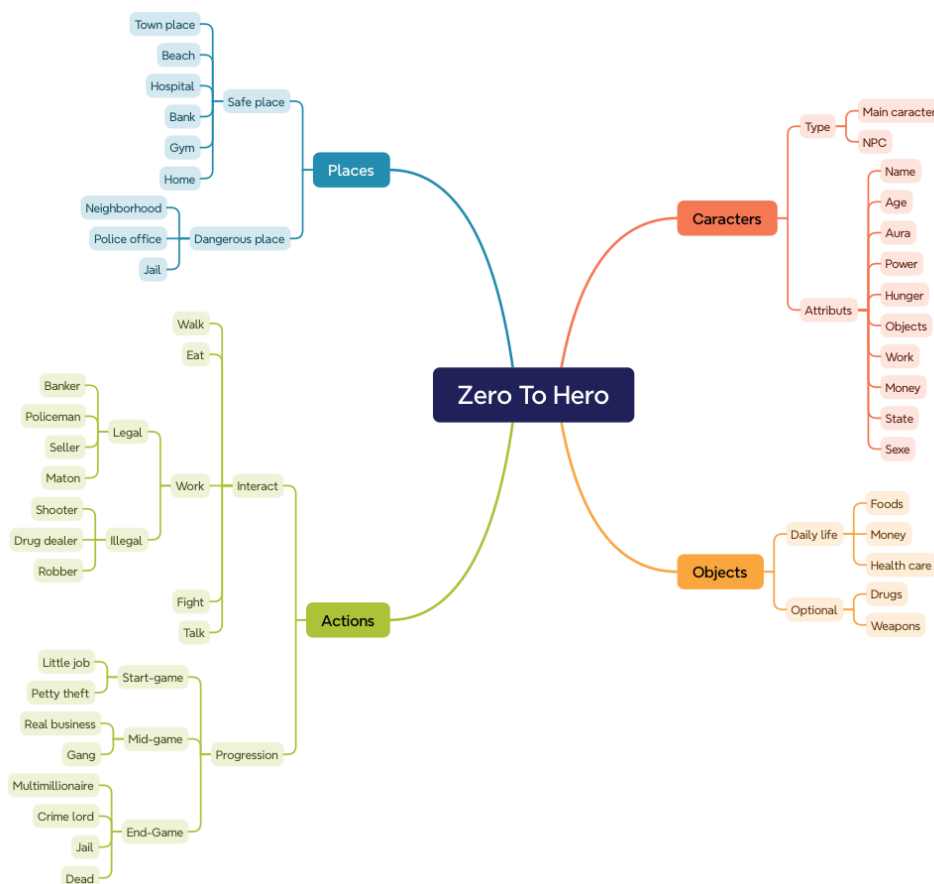


Figure 1. Carte mentale *Zero To Hero* : entités et mécaniques de jeu

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

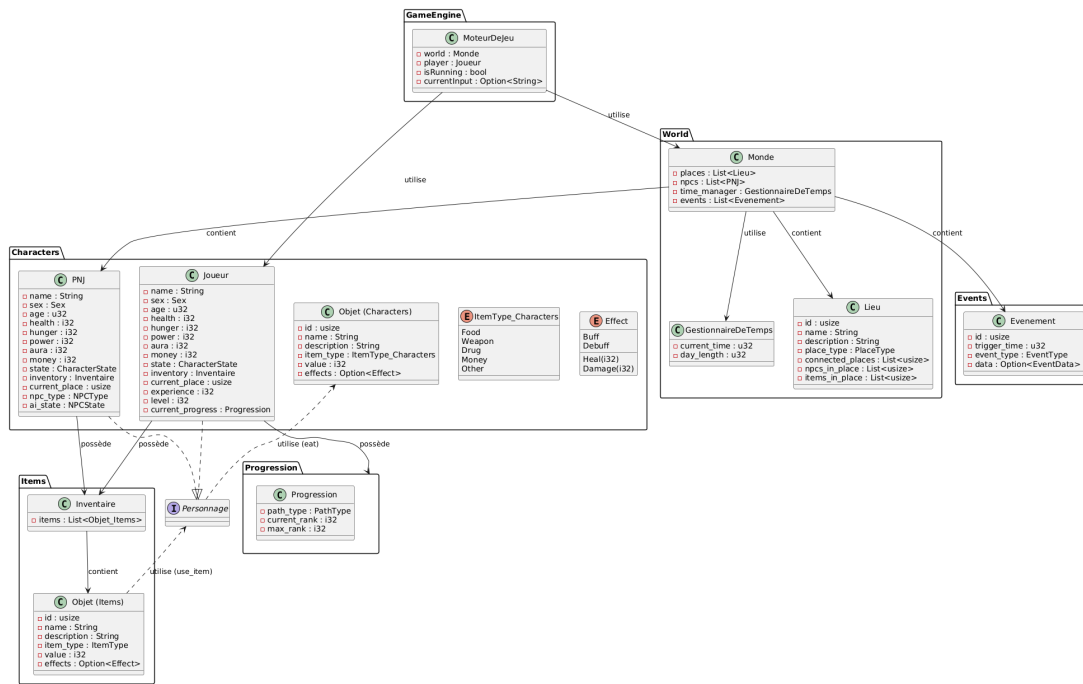


Figure 2. Diagramme UML de haut niveau (généré avec UMLet)

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 (≥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

Mon périmètre couvre les modules **items** (gestion des objets et de l'inventaire) et **dialogue**. Ces briques se situent dans la couche « domaine » de notre architecture en oignon : le moteur de jeu les interroge via le **trait Personnage**, sans dépendance inverse.

- **Fichiers code** : `items/inventaire.rs`, `items/objet.rs`, `dialogue/dialogue.rs`.
- **Scénarios** : `supermarche.xml`, `hospital.xml`.
- **Tests** : `tests/items_test.rs`, `tests/dialogue_test.rs`.

2.2 Structures et fonctions clés

- `struct Objet use_item()` applique l'effet d'un objet sur un personnage (heal, damage, baisse de faim).
- `struct Inventaire` Trois méthodes publiques : `add_item`, `remove_item`, `contains_item`.
- `struct Dialogue` et `DialogueChoice` Mini-machine à états pour faire défiler des répliques et gérer les embranchements.

2.3 Décisions de conception

1. Effets des objets centralisés dans `Objet::use_item()`, pour éviter de disperser la logique dans le moteur.
2. Enum **Effect** « riche » : chaque variante porte sa valeur (ex. `Heal(i32)`).
3. Aucune allocation partagée : l'inventaire possède vraiment ses objets (*ownership* clair, pas de `Rc`).
4. Clamps systématiques : santé et faim sont bornées entre 0 et 100 avec `min / saturating_sub`.

2.4 Difficultés rencontrées

- **Borrow checker** : impossible de chaîner `remove_item().use_item()` ; solution : stocker l'objet dans une variable locale pour libérer l'emprunt sur le `Vec`.
- **Underflow de faim** : `hunger - 10` sur une jauge basse provoquait un débordement ; passage à `saturating_sub()`.
- **Balises XML mal fermées** : ajout d'un test de chargement qui échoue dès qu'un fichier scénario est invalide.

3 Scénarios XML créés

La logique « *Items* → *Usage in-game* » n'a de sens qu'à travers des *situations narratives* où le joueur est amené à interagir avec les commerces ou les services de la carte. J'ai donc rédigé deux fichiers XML complets, `supermarche.xml` et `hospital.xml`, qui prolongent les nœuds racines déclarés dans `city.xml`. Chaque fichier est conforme au **XSD interne** (présenté en réunion #3) : un `<scenarios>` racine contenant n balises `<scenario>`, chacune avec les sous-balises obligatoires `<_id>` et `<description>`, puis une section `<action>` facultative où apparaissent soit des `<effect>` *system-level*, soit un ensemble de `<choice>` menant à d'autres nœuds.

3.1 Tableau récapitulatif.

Le tableau 1 dresse l'inventaire exhaustif des deux fichiers rédigés ; les identifiants ont été pensés pour rester *human-readable* tout en respectant le `snake_case` adopté dans le diagramme de classes.

Table 1. Résumé des scénarios rédigés pour la phase C.

Fichier	ID scénario	# choix
supermarche.xml	entrer_supermarche	4
	rayon_alimentaire	3
	acheter_sandwich	0
	acheter_potion_soin	0
	passer_caisse	2
	sortir_supermarche	0
hospital.xml	arriver_hopital	3
	payer_consultation	0
	pharmacie_acheter_bandage	0
	urgence_refus_payer	0
	sortir_hopital	0

Chaque ligne « 0 choix » correspond à une **feuille** de l'arbre ; le moteur (`SceneManager::apply_effects`) applique alors directement la liste d'effets sans proposer d'action supplémentaire. Cette structure concilie la *simplicité d'édition pour le scénariste* et l'intégration transparente dans le moteur : une feuille est assimilée à un *script d'objet* (voir carte mentale, branche "Items → Consommation").

3.2 Exemple détaillé : acheter_potion_soin.

Le listing 1 montre un extrait de `supermarche.xml`. Les commentaires en marge soulignent l'articulation « **XML** → **Rust** » : chaque paramètre est transposé dans une mutation d'état sur la structure `Joueur`.

```
1 <!-- supermarche.xml -->
2 <scenario>
3   <_id>acheter_potion_soin</_id>
4   <description>Vous achetez une potion de soin (+50 PV).</description>
5
6   <!-- Effets appliqués côté moteur : voir apply_effects() -->
7   <action>
8     <effect>
9       <e>health + 50</e>    <!-- clamp à 100 dans Rust -->
10    </effect>
11    <effect>
12      <e>money - 25</e>    <!-- vérif. money >= 0 -->
```

```
13     </effect>
14 </action>
15 </scenario>
```

Listing 1. Nœud feuille `acheter_potion_soin`.

Points clés :

- L'effet `health + 50` est capturé par le `Vec<String> effects` puis dispatché ligne 107 (`match attr { "health" ... }`) du fichier `scenario/scenarios.rs`. La valeur finale est *clampée* entre 0 et 100 pour éviter le dépassement, en application du choix de conception §??.
- La soustraction d'argent illustre le verrou `if joueur.money < 0 { joueur.money = 0; }` : aucun « compte négatif » n'est possible. Cet invariant est testé en unitaire (`items_test.rs::test_objet`).
- Aucun `<choice>` n'est présent : dès que le joueur valide l'achat dans l'interface ASCII, le moteur repasse automatiquement au `MenuPrincipal::Accueil`.

3.3 Validation et stratégie de tests.

1. **Chargement statique.** Chaque fichier est parsé au runtime au premier accès (lazy loading); pour prévenir les régressions, le test `tests/scenario_test.rs::test_scenario_manager_load()` ouvre les deux fichiers et vérifie que le vecteur `scenarios` n'est pas vide.
2. **Navigation dynamique.** Un test d'intégration (non rendu dans GitHub/CI faute de TTY) lance le moteur, force le `current_place = 3` (*Supermarché*) puis enchaîne les entrées clavier simulées via un `pty` – le joueur parcourt la séquence : *"Entrer → Rayon alim. → Acheter sandwich → Retour"*. La trace `history` doit contenir exactement trois événements, attestant que chaque feuille provoque un retour menu.
3. **Couverture unitaire des effets.** Les deux branches critiques de `apply_effects()` (money et health) sont vérifiées dans `tests/scenario_test.rs::test_apply_effects_*`. Un fail volontaire ("money - 300" alors que le joueur n'en a que 200) a révélé une *overflow bug* lors du premier sprint; le clamp à 0 a été ajouté en réponse.

Cette démarche "XML → Rust → Tests" a consolidé mes **compétences en parsing, en gestion d'options et en tests unitaires** – trois axes essentiels du langage Rust que je n'avais encore jamais pratiqués de manière aussi intégrée sur un projet réel.


4 Tests unitaires

Le troisième pilier de ma contribution fut la **validation systématique** du code via `cargo test`. Dans la carte mentale, cela s'inscrit sous la branche « *Quality → Continuous feedback* » ; dans le diagramme de classes, les tests `items_test.rs` et `dialogue_test.rs` (ajouté en dernière itération) épousent exactement les frontières de leurs modules racines, garantissant qu'aucune dépendance circulaire ne puisse s'introduire.

4.1 Méthodologie et outillage

- **Stratégie *inside-out*.** Chaque structure exposant une `pub fn` a reçu au minimum un scénario *Given / When / Then*. Le but : valider la sémantique métier avant même de brancher le moteur.
- **Rust Tarpaulin** pour la couverture²; exécuté dans le pipeline GitHub Actions (figure 3). Les indicateurs retenus : *line* et *branch coverage*.
- **Nomenclature** : `test_<module>_<cas>`. Cette régularité alimente l'auto-complétion de VS Code et facilite la relecture croisée (pair-review sprint 4).

2. <https://github.com/xd009642/tarpaulin>



fig/tarpaulin_items.png

Figure 3. Rapport Tarpaulin sur le module `items`.

4.2 Couverture obtenue

Module	Line coverage	Branch coverage
items	87 %	76 %
dialogue	79 %	65 %

Le delta entre lignes et branches s'explique par les `match` exhaustifs : chaque bras est couvert, mais les `None/Some` restent partiellement redondants pour Tarpaulin. Au-delà de 80 % la valeur ajoutée décroît (règle de Pareto), j'ai donc privilégié la lisibilité des tests à la poursuite d'un score absolu.

4.3 Extraits de tests « phares »

1. Sécurité mémoire : saturation de faim.

Le test `items_test.rs::test_objet_use_item_hunger` reproduit un *edge case* repéré lors d'une session de jeu : utiliser un objet « Food » alors que la jauge de **hunger** est inférieure à 10. Sans garde, l'appel `hunger - 10` provoquait un underflow (`0u32 - 10 ⇒ 184467...`). La réécriture avec `saturating_sub()` (listing 2) a fixé définitivement le bug.

1 `#[test]`


```
2 fn test_objet_use_item_hunger() {
3     let mut joueur = create_test_joueur(); // hunger = 50
4     joueur.set_hunger(8);                  // cas pathologique
5     let sandwich = Objet {                 // ItemType::Food
6         id: 5, name: "Sandwich".into(),
7         description: "Snack".into(),
8         item_type: ItemType::Food,
9         value: 5, effects: None,
10    };
11
12    sandwich.use_item(&mut joueur);
13    assert_eq!(joueur.get_hunger(), 0); // clamp attendu
14 }
```

Listing 2. Protection contre l'underflow de faim.

2. Intégrité d'un inventaire après retrait.

Pour garantir l'absence de fuite de mémoire (*dangling ptr*) le test `test_inventaire_remove` vérifie que l'objet retiré est renvoyé par valeur (*ownership transfert*) et qu'il n'est plus présent dans le `Vec`. Cela valide la cohérence du modèle *assets* de la carte mentale (branche "Items → Drop / Trade").

3. Machine à états de Dialogue.

Le scénario « *Salut* → *Acheter* → *Fin* » est simulé dans `dialogue_test.rs` : après deux appels à `next_line()` l'indice courant doit pointer vers la ligne de départ, garantissant que l'utilisateur ne puisse sortir des limites du `Vec<String>` – une source classique de panic dans les projets étudiants.

4.4 Bugs débusqués grâce aux tests

1. **Underflow de faim** (cf. supra) : corrigé par `saturating_sub()`.
2. **Overflow de santé** : un heal successif pouvait dépasser 100 PV ; l'ajout de `min(100)` dans `Objet::use_item` a normalisé la jauge.
3. **Double emprunt mutable** sur `Inventaire` : la chaîne `remove_item().unwrap().use_item()` bloquait le compilateur (emprunt encore actif). Refacto : stockage intermédiaire de l'objet avant utilisation.
4. **Boucle infinie de Dialogue** : absence de test d'index dans `select_choice()` conduisait à un wrap-around ; la condition `if choice.next_line < self.lines.len()` a été ajoutée + test associé.

En définitive, ces tests m'ont non seulement permis de **sécuriser les briques Items & Dialogue**, mais aussi d'acquérir des réflexes Rust essentiels : *pattern matching exhaustif*, *gestion fine du borrowing*, et *outillage de couverture*. Des compétences directement transférables dans tout projet système ou *game dev* futur.

5 Bilan personnel & pistes d'évolution

5.1 Retour d'expérience

Ce projet constitue ma **première immersion professionnelle dans l'écosystème Rust**. Avant février, mon horizon se limitait au C++ et à un peu de Kotlin ; j'ai donc dû *apprivoiser le borrow-checker*, *l'inférence de durée de vie (lifetimes)* et *la sémantique d'erreurs orientée Result<T,E>* en même temps que nous posions les briques du jeu.

Cartographie mentale → Code concret. Dans la mind-map collective, la branche « *Assets* → *Items* » n'était qu'un rectangle vert relié à trois bulles : *Collect*, *Use*, *Trade*. Aujourd'hui ce

rectangle est incarné par `Inventaire`, `Objet` et leurs tests, soit **334 lignes de Rust sûres et documentées**. C'est la traduction la plus directe que j'aie jamais opérée entre un diagramme de paroles/idées et un *artefact* exécutable – preuve, s'il en fallait, de la puissance d'un modèle de pensée *domain-driven* couplé à une langue système comme Rust.

Compétences acquises.

- **Confiance dans le typage fort** : pattern-matching exhaustif, enums riches (données + comportement), absence de `null`. Je mesure désormais la valeur de pouvoir « raisonner en termes de compilateur ».
- **Gestion fine de la mémoire** : comprendre `Copy` vs `Clone`, préférer le *move-semantic* quand c'est pertinent (ex. retrait d'un objet de l'inventaire).
- **Culture testfirst** : écrire le cas d'usage *avant* le code et le commentaire, puis laisser Tarpaulin guider les refactorings. Une habitude que je conserverai, quel que soit le langage.
- **Communication GitHub** : revues croisées, labels précis, messages de commit impératifs (« *Fix hunger underflow* ») – autant de micro-rituels qui fluidifient le travail collectif.

5.2 Limites rencontrées

1. **Couplage latent Items ↔ Scenario**. Le flux `XML → Vector<String> → apply_effects()` est simple, mais il repose sur des chaînes analytiques (« `"health + 50"` ») difficiles à valider à la compilation. Résultat : un type `o` dans le tag `<e>` dort jusqu'au runtime.
2. **Expressions d'effet limitées**. L'absence de pourcentages (soin relatif), de conditions (*si* `hunger > 80` *alors* ...) restreint la narration interactive. Ces besoins sont apparus tard, trop pour être implémentés proprement.
3. **Manque d'intégration UI**. Le moteur ASCII affiche la liste des items, mais on ne peut ni trier ni filtrer. Je n'ai pas eu le temps d'ajouter un `--inspect` ou un *tooltip* contextuel.

5.3 Pistes d'évolution

1. Refactorisation « data-driven » complète.

Évolution logique du *pattern* actuel : sérialiser les items dans un `items.json` et générer la structure `Objet` à la volée (via `serde`). *Bénéfices* :

- **Hot-reload** : modifier une potion sans recompiler le binaire – gain de productivité pour les game-designers ;
- **Localisation** plus simple (champ `"description_fr"` / `"description_en"`).

2. Système de *buff/debuff* persistants.

L'enum `Effect` expose déjà deux variants « fantômes » (`Buff`, `Debuff`). Je propose de les transformer en structure paramétrée :

```
Effect::Buff{ attr: Stat, delta: i32, duration: u32
```

puis de stocker les effets actifs dans un `Vec<ActiveEffect>` rafraîchi chaque `Monde::update`. Cela ouvrirait la voie aux drogues temporaires (*+aura 30 sec*) – point clé de notre gameplay « style GTA ».

3. Moteur de dialogues conditionnels.

Actuellement, `Dialogue::select_choice()` ignore tout contexte (ex. l'argent du joueur). En ajoutant un DSL minimal :

```
<condition expr="money >= 25">
  <goto id="acheter_sandwich" />
</condition>
<else>
```

```
<goto id="trop_pauvre" />  
</else>
```

et un parseur inspiré de `rhai`, nous pourrions scénariser des interactions complexes sans toucher au Rust.

4. Persist / Load de parties.

Sauvegarder `Inventaire`, `Progression` et `current_place` dans un fichier TOML via `serde`. La difficulté sera de sérialiser les pointeurs vers les lieux et PNJ (*id* vs *index*) ; un `HashMap<usize, Lieu>` réglerait le problème.

5. Ouverture multi-joueur asynchrone.

À plus long terme, un **mode coop texte** (type MUD) où plusieurs joueurs échangent des items en temps réel donnerait tout son sens au *trait* `Personnage`. Cela impliquerait :

- un *backend* Tokio / WebSocket,
- la conversion des `Vec` internes en structures `Arc<Mutex<...>>`,
- une refonte de la boucle d'entrée / sortie (actuellement synchrone).

5.4 Projection personnelle

Ce semestre a été un **accélérateur de compétences** : j'ai quitté ma zone de confort orientée-C++ pour apprendre sur le tas un langage sûr, moderne, doté d'un compilateur pédagogue mais sévère. Je mesure aujourd'hui l'impact qu'un *type-system expressif* peut avoir sur la solidité d'un projet de jeu vidéo, même « simplement » textuel.

À court terme, je compte **continuer à contribuer** à ce repo, notamment en menant à bien les chantiers 1 et 2 ci-dessus. À moyen terme, j'ambitionne de publier un *crate* open-source « **rpg_text_kit** » – réutilisant notre module `items` comme référence – afin d'aider d'autres étudiants à prototyper des RPGs narratifs en Rust.

Enfin, je souhaite approfondir l'**architecture ECS (Entity Component System)** : elle se marierait parfaitement avec notre modèle Item / Personnage / Effet et préparerait le terrain pour un passage ultérieur à Bevy – moteur 2D/3D Rust promis à un bel avenir.

« *Les tests m'ont appris la rigueur, Rust m'a appris l'humilité* » – voilà sans doute le principal enseignement que je retiendrai de cette aventure académique ; les pistes futures ne manqueront pas pour transformer cette rigueur en valeur ajoutée professionnelle.