Rapport de développement – Simulation hospitalière (PCO)

Auteur : Sofian Ethenoz & Gatien Jayme

Ce document présente la démarche suivie pour implémenter la simulation hospitalière en appliquant le Test-Driven Development (TDD) et décrit les décisions techniques adoptées pour respecter l'intégralité des tests fonctionnels et de concurrence.

1. Méthodologie

Phase	Description
Exécution initiale des tests	Les tests fournis ont été lancés dès le départ afin d'identifier l'ensemble des tests *FAILED*.
Boucle TDD	Pour chaque test *FAIL*: 1. *Comprendre* le comportement attendu. 2. Implémenter la modification minimale permettant de faire passer ce test. 3. Relancer la suite complète afin de vérifier que nous n'avons pas cassé d'autres tests. 4. Refactoriser le code et ajouter les garde-fous nécessaires (vérification d'arguments, conditions de sortie, etc.).
Documentation incrémentale	Des commentaires ont été ajoutés au fur et à mesure directement dans le code, afin d'expliciter l'intention de chaque changement (verrouillage, facturation, contrôle de stock). Cette discipline évite la problématique de « code muet » observé lors du laboratoire précédent.
Intégration finale	Une fois la totalité des tests dans l'état *PASSED*, le point d'entrée `main` a été exécuté. Les erreurs restantes (incohérences de fonds, problèmes de concurrence) ont alors été corrigées lors d'un dernier passage.

2. Synchronisation : choix des **mutex**

Les zones critiques concernent tous les accès concurrents aux variables partagées :
`money`, `stocks`, `unpaidBills`, mais aussi `patientsInfo` et `nbFreed` côté hôpital.
Toutes les lectures/écritures potentiellement concurrentes sur ces attributs sont protégées par un `PcoMutex` propre à chaque entité.

Dans une première version, nos mutex étaient déclarés comme `static` dans les fichiers `.cpp`.
Cela avait deux effets indésirables :

- toutes les instances d'un même type (toutes les ambulances, tous les hôpitaux, etc.) partageaient un mutex global, ce qui réduisait fortement le parallélisme ;

- il devenait plus difficile de respecter l'esprit de la consigne (« sections critiques les plus courtes possibles »).

Nous avons donc opté pour un mutex membre privé par classe pour (`Ambulance`, `Clinic`, `Hospital`, `Supplier`, `Insurance`).
Ce choix permet :

- de protéger uniquement les données partagées propres à chaque instance (`money`, `stocks`, `unpaidBills`, `patientsInfo`, `nbFreed`, …) ;
- de laisser plusieurs instances d'un même type progresser en parallèle sans se bloquer inutilement.

Conformément au cahier des charges, nous avons également pris soin de ne jamais détenir deux mutex à la fois.

Avant chaque appel à une méthode d'un autre acteur (`transfer`, `invoice`, `buy`, `pay`), le mutex local est systématiquement relâché. Lorsque c'est nécessaire, les données sont copiées dans des variables locales (p. ex. liste de factures à payer) afin de traiter la logique métier hors section critique.

Enfin, certaines fonctions utilitaires comme `Hospital::getNumberPatients()` ou `Hospital::addPatients()` ne verrouillent pas elles-mêmes le mutex, car elles sont toujours appelées depuis des méthodes déjà protégées. Cela évite les double-verrouillages sur le même `PcoMutex` et donc les deadlocks, tout en restant conforme aux règles de concurrence du laboratoire.

Un mutex suffit à garantir l'exclusion mutuelle dans tous ces cas, l'utilisation d'un sémaphore, plus lourde, n'apporterait pas de bénéfice supplémentaire dans ce contexte.

---

3. Validation

Invariant vérifié	**Valeur attendue**	**Valeur obtenue**
Somme globale des fonds (hors cotisations assurance)	**4 700 €**	**4 700 €**
Nombre total de patients	**900**	**900**
Suite Google Test	**31 / 31** tests verts	✔

Ces résultats correspondent aux attentes et valident l'implémentation. Cependant, le simple fait d'obtenir ces résultats une fois ne garantit pas à lui seul une gestion correcte et durable de la concurrence.

---

4. Utilisation de l'IA

Aucune portion de code n'a été générée automatiquement. L'intelligence artificielle (ChatGPT 5) a uniquement été sollicitée pour la mise en forme ainsi que la reformulation de la présente documentation. Le but étant de rendre la documentation concise et professionnelle.

---

5. Mot de la fin

Nous avons fais l'erreur sur le labo précédant de ne pas commenter nôtre code. Nous avons fait tout l'inverse sur ce labo. Lorsque l'on travail sur le code écrit par quelqu'un d'autre, comprendre ce qu'a voulu

faire le développeur avant nous, n'est pas chose aisée. L'utilisation systématique des commentaires permet de se plonger dans le code plus facilement et cela rend le code plus facile à lire et à comprendre.

Aussi, les occasions de faire du vrai TDD ne sont pas nombreuses. Sur ce laboratoire, les tests on rendu bien plus simple l'implémentation des fonctionnalités. Si le temps le permettait, une plus grande quantité de testes pourraient être ajoutés par exemple sur les facturations et l'argent de manière générale.

Nous avons dû investir un temps conséquent qui aurait pu être évité si ces tests avaient été implémnentés dés le début lorsque nous avons réalisé que les fonds n'étaient pas conformes en fin de programme.

Aussi, les commentaires todo ont été laissé dans le code pour distinguer plus facilement nôtre code de celui fourni