Gestion des risques et des rendus

Équipe gl56

14 janvier 2026

Table des matières

1 Gestion des risques

1.1 Classification des risques

On peut répartir les risques dans ce projet en deux grandes catégories :

— **Risques organisationnels et humains :** Liés à la gestion de l'équipe, la communication, la répartition du travail et le respect des délais.

— **Risques techniques :** Liés au développement du compilateur.

1.2 Risques organisationnels et humains

1.2.1 Risque 1 : Oubli des dates de rendus

Description : Ne pas respecter les échéances du projet.
Niveau de criticité : ★★★★ (4/5) - Critique
Impact potentiel :

— Pénalités importantes sur la note finale

— Stress intense et démotivation de l'équipe

— Risque d'échec du projet

Solutions et actions de préventions :

☐ Configurer des rappels automatiques 3 jours et 1 jour avant chaque échéance

☐ Rappeler les dates pour chauqe discussion de groupe.

☐ Donner des délais d'anticipation pour ne pas attendre à la dernière minute(par exemple finir les docs du suiv deux jours avant le suivi)

1.2.2 Risque 2 : Mauvaise répartition du travail

Description : Certains membres de l'équipe font plus de travail que d'autres.
Niveau de criticité : ★★★ (3/5) - Modéré
Impact potentiel :

— Conflits et tensions au sein de l'équipe

— Démotivation des membres surchargés

— Sentiment d'injustice.

Solutions et actions préventives :

☐ donner clairement les responsabilités pour chaque tâche

☐ S'assurer que chaque membre comprend ses tâches et leurs échéances

☐ Rééquilibrer les charges de travail si nécessaire (par exemple si on rend compte qu'une tâche est très exigeante)

1.2.3 Risque 3 : Démotivation d'un membre de l'équipe

Description : Un membre de l'équipe perd sa motivation, participe moins aux discussions de l'équipe et contribue moins au projet.
Niveau de criticité : ★★★ (3/5) - Modéré
Impact potentiel :

— Surcharge de travail pour les autres membres

— Ralentissement global du projet

— Retard potentiel pour le rendu

Solutions et actions de préventions :

☐ Adapter les tâches aux préférences et compétences de chacun quand c'est possible (par exemple si quelqu'un aime l'assembleur, on lui donne la responsablité pour la partie C)

☐ Faire une nouvelle répartition des tâches

☐ Donner le soutien si le démotivation est liée à des raisons personnelles pressantes.

1.2.4 Risque 4 : Mauvaise communication au sein du groupe

Description : Manque de communication entre les membres(rare)
Niveau de criticité : ★★★★ (4/5) - Critique
Impact potentiel :

— plusieurs personnes font la même tâche, donc perte du temps

— Incompatibilité entre les différentes parties du code

— Frustration et tensions dans l'équipe

— Impossibilité de rendre le projet dans les délais

Solutions et actions de prévention :

☐ Créer un groupe pour la communication (Discord, WhatsApp) et échanger régulièrement

☐ Prévenir l'équipe avant de travailler sur une partie critique du code

☐ ne jamais prendre des décisions sans discuter avec le groupe

1.2.5 Risque 5 : Inactivité d'un membre du groupe

Description : Un membre de l'équipe est inactive pendant une période prolongée (pas de réponses, pas de prévention...)
Niveau de criticité : ★★★ (3/5) - Modéré
Impact potentiel :

— Retard sur les tâches assignées à ce membre

— Surcharge de travail pour les membres présents

— Risque de ne pas comprendre le code écrit par la personne absente

— Désorganisation temporaire de l'équipe

Solutions et actions de prévention :

☐ Tenir les profs au courant dans ces situations critiques

☐ Une nouvelle partage des tâches pour avancer

1.3 Risques techniques

1.3.1 Risque 6 : Perte de code ou de travail

Description : Perte de fichiers, de code ou de documentation suite à un problème technique ou une erreur de manipulation.
Niveau de criticité : ★★★★★ (5/5) - Catastrophique
Impact potentiel :

— Perte de plusieurs jours voire semaines de travail

— Impossibilité de livrer le projet dans les délais

— Démotivation l'équipe et conflits internes

— Échec potentiel du projet

Solutions et actions de prévention :

☐ Faire `git commit` et `git push` après chaque session de travail.

☐ Faire un `git pull` toujours et obligatoirement avant de commencer à travailler

☐ Ne jamais utiliser `git force push` sans l'accord de toute l'équipe

1.3.2 Risque 7 : Compilateur non-fonctionnel avant un rendu

Description : Découvrir que le compilateur ne compile plus ou ne fonctionne plus correctement juste avant une échéance.

Niveau de criticité : ★★★★★ (5/5) - Catastrophique

Impact potentiel :

— Impossibilité de rendre un projet fonctionnel

— Panique au sein de l'équipe

— Risque d'échec du projet complet

Solutions et actions de prévision :

☐ Exécuter le script `common-tests.sh` systématiquement à chaque fois qu'on ajoute du code.

☐ Tester manuellement le code ajouté avant de faire push.

☐ Faire une brache temporaire et ne fusionner dans la branche principale qu'après validation des tests

1.3.3 Risque 8 : Retard majeur sur une partie bloquante du projet

Description : Prendre du retard sur une partie du projet dont dépendent d'autres parties (par exemple, l'étape B qui bloque le reste).

Niveau de criticité : ★★★★★ (5/5) - Catastrophique

Impact potentiel :

— toutes les parties dépendantes sont bloquées

— Impossibilité d'avancer sur le projet dans son ensemble

— Stress et pression sur les membres responsables de cette partie

— Risque de rendre un projet incomplet

Solutions et actions préventives :

☐ Identifier clairement les dépendances entre les différentes parties du projet

☐ Prioriser absolument les parties bloquantes

☐ En cas de retard détecté, donner de l'aidre aux responsables de la partie en retard pour avancer

☐ Ecrire du code qui foncionne même si il n'est pas optimisé au début (ne chercher pas à perfectionner au début)

1.3.4 Risque 9 : Retard sur une partie non-bloquante

Description : Prendre du retard sur une fonctionnalité optionnelle qui n'empêche pas le reste du projet d'avancer.

Niveau de criticité : ★★ (2/5) - Faible

Impact potentiel :

— Perte de points bonus

— Projet moins complet que prévu mais fonctionnel

— Déception par rapport aux objectifs initiaux

Solutions et actions préventives :

☐ Prioriser clairement : d'abord le compilateur de base, ensuite les bonus

☐ Définir un planning réaliste

☐ Communiquer en équipe sur les priorités et accepter de ne pas tout faire

1.4 Risques liés à la soutenance et aux présentations

Cette catégorie couvre les incidents techniques et organisationnels pouvant survenir lors de la restitution orale du projet (soutenance finale, revues de projet).

1.4.1 Risque 10 : Défaillance du matériel de projection ou de visio-conférence

— **Description :** Il est fréquent que des problèmes techniques surviennent au moment de la présentation.

— **Stratégie préventive (Avant) :**

  — Tester le matériel (projecteur, câbles HDMI, son) dans la salle de soutenance spécifique plusieurs jours avant le jour J, en utilisant l'ordinateur qui servira à la présentation.

  — Exporter impérativement les slides au format **PDF** statique pour éviter les problèmes de compatibilité de police ou de logiciel (type PowerPoint/Canva).

  — Avoir les slides stockés sur plusieurs supports locaux (Clé USB, Disque dur) pour ne pas dépendre d'une connexion Internet.

— **Stratégie corrective (Pendant) :**

  — En cas d'impossibilité totale de projection, nous basculons sur une présentation sans support visuel, en nous appuyant sur des notes papiers imprimées servant de fil conducteur pour chaque orateur.

1.4.2 Risque 11 : Échec de la démonstration technique ("Effet Démo")

— **Description :** Le code ou l'exécutable, bien que fonctionnel lors des tests, plante ou ne se lance pas lors de la démonstration en direct devant le jury (bugs imprévus, environnement différent).

— **Stratégie préventive (Avant) :**

  — Figer une version "stable" du code (code freeze) dédiée à la démo 24h avant la soutenance, qui ne sera plus modifiée.

  — Enregistrer une **vidéo de la démonstration** (capture d'écran commentée) montrant le fonctionnement nominal du projet.

— **Stratégie corrective (Pendant) :**

  — Si la démo en direct échoue et que le problème n'est pas résolu en moins d'une minute, nous basculons immédiatement sur la diffusion de la vidéo de secours pour prouver le fonctionnement du projet sans perdre de temps sur le débogage.

On constate que les risques techniques sont potentiellement plus graves mais les risques organisationnels sont plus nombreux et nécessitent une gestion continue de l'équipe.

2 Gestion des rendus

La gestion des rendus (« Release Management ») est une étape critique pour garantir que le livrable final correspond aux attentes de qualité et de fonctionnalité, tout en minimisant le stress de dernière minute. Notre stratégie repose sur l'automatisation via Maven et une procédure de vérification manuelle stricte avant chaque échéance.

2.1 Procédure automatisée

Afin d'assurer la reproductibilité des livrables, le processus de construction est entièrement automatisé.

— **Construction (Build) :** Le projet utilise `Maven` pour gérer le cycle de vie du logiciel. La commande `mvn package` garantit que le projet est compilé proprement, que les tests unitaires sont passés avec succès, et que l'archive finale (JAR) est générée avec toutes ses dépendances.

— **Validation continue :** L'exécution systématique du script fourni `common-tests.sh` permet de vérifier que le compilateur ne régresse pas sur la base de tests commune. Ce script est intégré dans notre workflow de développement pour être lancé avant toute fusion majeure.

2.2 Checklist de pré-rendu

Malgré l'automatisation, certaines vérifications nécessitent une intervention humaine ou une validation contextuelle spécifique avant la soumission finale. Nous avons établi la checklist suivante, à exécuter impérativement avant chaque date de rendu (Intermédiaire, Final, etc.) :

1. **Synchronisation du dépôt :**
   — *Action :* Vérifier l'état du dépôt Git avec `git status` et s'assurer que tous les fichiers nécessaires (nouveaux tests, classes Java, documentation) sont bien ajoutés et commités.
   — *Utilité :* Évite le syndrome du « ça marche chez moi » causé par des fichiers locaux non partagés sur le dépôt distant.
2. **Test en environnement neutre :**
   — *Action :* Cloner le dépôt dans un répertoire temporaire vierge (`/tmp/test-rendu`) et lancer la compilation complète (`mvn package`).
   — *Utilité :* Confirme que le projet est autonome et ne dépend pas de configurations spécifiques à la machine d'un développeur.
3. **Validation des consignes spécifiques :**
   — *Action :* Relire le document `[A-Rendre]` et vérifier point par point que le format de l'archive, le nom des exécutables (ex : `decac`) et l'arborescence respectent scrupuleusement les demandes.
   — *Utilité :* Prévient les pénalités "bêtes" liées au non-respect du format de soumission, qui rendraient le projet inévaluable par les scripts de correction automatique.
4. **Documentation à jour :**
   — *Action :* S'assurer que la documentation technique (et notamment cette section de gestion des risques) est à jour par rapport au code effectivement livré.
   — *Utilité :* Garantit la cohérence entre ce qui est décrit et ce qui est implémenté.

2.3 Script d'automatisation global (test_all.sh)

Afin de fluidifier les vérifications avant rendu et d'éviter les erreurs manuelles, nous avons développé et mis en place un script shell unique : `test_all.sh`. Ce script est le point d'entrée principal pour la validation technique du projet.

Il exécute séquentiellement les trois étapes critiques de notre chaîne de validation :

1. **Nettoyage et Tests Unitaires :** Le script lance la commande `mvn clean test` en activant explicitement l'agent JaCoCo . Cela garantit que le projet compile proprement depuis zéro et que tous les tests unitaires Java sont valides. Si cette étape échoue, le script s'arrête immédiatement pour empêcher un rendu cassé.
2. **Tests d'Intégration (Compilateur + Machine Abstraite) :** Le script parcourt récursivement tous les fichiers `.deca` présents dans `src/test/deca`. Pour chaque fichier trouvé :
   — Il tente de le compiler avec notre exécutable `decac`.
   — Si la compilation réussit (génération d'un fichier `.ass`), il lance automatiquement l'exécution via `ima` pour vérifier le comportement à l'exécution.
   — Il signale visuellement les échecs de compilation (ce qui permet aussi de valider les tests qui sont *censés* échouer, comme les tests syntaxiques invalides).
3. **Analyse de la Couverture :** Enfin, il déclenche la génération du rapport de couverture de code via `src/test/script/jacoco-report.sh`, permettant à l'équipe de vérifier instantanément dans `target/site/jacoco/index.html` si les nouvelles fonctionnalités sont suffisamment testées.

Cette procédure de mise en production est la garante de notre sérénité lors des échéances. Elle transforme le rendu d'une source de stress imprévisible en une suite d'étapes maîtrisées et routinières.