Rapport d'utilisation de GitHub Copilot

1. Contexte

Dans le cadre du développement de notre jeu (clone de Ketchapp Circle), j'ai utilisé **GitHub Copilot** comme assistant de programmation intégré à VS Code. Ce rapport décrit mon expérience, les méthodes utilisées, et les limites rencontrées.

2. Utilisation des commentaires `/** */` (Copilot Tab)

Ma première approche a été d'utiliser les **commentaires Javadoc** (`/** */`) comme prompts pour déclencher les suggestions automatiques de Copilot (complétion par tabulation).

Fonctionnement

En écrivant un commentaire décrivant la fonction souhaitée juste au-dessus d'une méthode, Copilot propose automatiquement une implémentation. Par exemple :

```
/** Fait sauter le joueur en appliquant une impulsion verticale négative
*/
public void jump() {
    // Copilot propose le corps de la méthode
}
```

Résultats

- **Cas simples** : Pour des méthodes isolées (getters, setters, calculs simples), les suggestions étaient souvent correctes du premier coup.
- **Cas complexes** : Dès que la méthode dépendait de classes définies dans d'autres fichiers (par exemple `Parcours`, `Position`, `Affichage`), Copilot Tab faisait régulièrement des erreurs :
  - Noms d'attributs incorrects ou inventés
  - Mauvaise signature de méthodes existantes
  - Imports manquants ou erronés
- **Ajustements nécessaires** : Il fallait fréquemment reformuler le commentaire-prompt, être plus précis, ou corriger manuellement le code généré, surtout pour les parties impliquant des dépendances inter-fichiers.

Exemple de limitation

Pour la méthode `checkCollision` dans `Parcours.java`, le commentaire :

```
/** Vérifie si le joueur entre en collision avec la ligne brisée */
```

... produisait du code qui ne correspondait pas à la structure réelle de nos `ArrayList<int[]>` de points. Il fallait corriger manuellement la logique de calcul point-segment.

3. Passage à GitHub Copilot Chat

Étant habitué à utiliser **Claude** (Anthropic) comme assistant IA, j'ai rapidement basculé vers **GitHub Copilot Chat** plutôt que de me reposer uniquement sur les commentaires et la complétion Tab.

Différence clé : le contexte

La grande différence est la possibilité de **fournir explicitement le contexte** au chat :

- En ajoutant les fichiers pertinents à la conversation (par exemple `Position.java`, `Parcours.java`, `Affichage.java`)
- En décrivant précisément ce qu'on veut par rapport au code existant

Cela a **tout changé** : les réponses étaient beaucoup plus précises et contenaient beaucoup moins d'erreurs, car le modèle avait accès à la structure réelle des classes, aux noms exacts des attributs et méthodes.

Comparaison

Critère	Copilot Tab (commentaires)	Copilot Chat (avec contexte)
Code isolé / simple	Bon	Très bon
Code avec dépendances inter-fichiers	Souvent incorrect	Correct dans la majorité des cas
Temps de correction manuelle	Moyen	Faible
Effort de rédaction du prompt	Commentaire concis mais limité	Plus détaillé mais plus efficace
Compréhension de l'architecture	Aucune (fichier courant uniquement)	Bonne (si on fournit les fichiers)

4. Bilan

- **Copilot Tab** est utile pour accélérer l'écriture de code répétitif ou de méthodes simples. Les commentaires `/** */` servent de prompt efficace dans ce cas.
- **Dès que le code implique plusieurs fichiers**, la complétion Tab atteint ses limites : elle ne « voit » pas les autres classes et fait des suppositions souvent fausses.
- **Copilot Chat avec contexte explicite** est nettement supérieur pour les tâches complexes. Fournir les bons fichiers au chat réduit considérablement les erreurs et le temps de correction.
- L'approche la plus productive a été de **combiner les deux** : Copilot Tab pour le code courant simple, et Chat pour les fonctionnalités impliquant de la logique inter-classes.