**ESIL**

08

**Rapport PSR**

**WarThread**

**Florent Mas,Sylvain Paillé,Tony Quesnel,Aurélien Fiol,Michaël Lanöe, Jean-Marc Muzi**

Sommaire

[1 Introduction 3](#_Toc199003135)

[2 Présentation du jeu 3](#_Toc199003136)

[2.1 But 3](#_Toc199003137)

[2.2 Règles 4](#_Toc199003138)

[3 Entités du jeu 5](#_Toc199003139)

[3.1 Dieu 5](#_Toc199003140)

[3.2 Fidele 5](#_Toc199003141)

[3.3 Warrior 5](#_Toc199003142)

[3.4 Diagramme UML 5](#_Toc199003143)

[4 Création du réseau et déploiement 7](#_Toc199003144)

[4.1 Création du graphe 7](#_Toc199003145)

[4.2 Déploiement du graphe 7](#_Toc199003146)

[4.3 Visualisation du jeu 7](#_Toc199003147)

[5 Déroulement du Jeu 7](#_Toc199003148)

[5.1 Migration des warriors 7](#_Toc199003149)

[5.2 IA des warriors 7](#_Toc199003150)

[5.2.1 Récolte des informations 7](#_Toc199003151)

[5.2.2 Détermination d’une stratégie 7](#_Toc199003152)

[5.3 Gestion des batailles 8](#_Toc199003153)

[6 Gestion des pannes 8](#_Toc199003154)

[6.1 Simulation d’une panne 8](#_Toc199003155)

[6.2 Détection des pannes 8](#_Toc199003156)

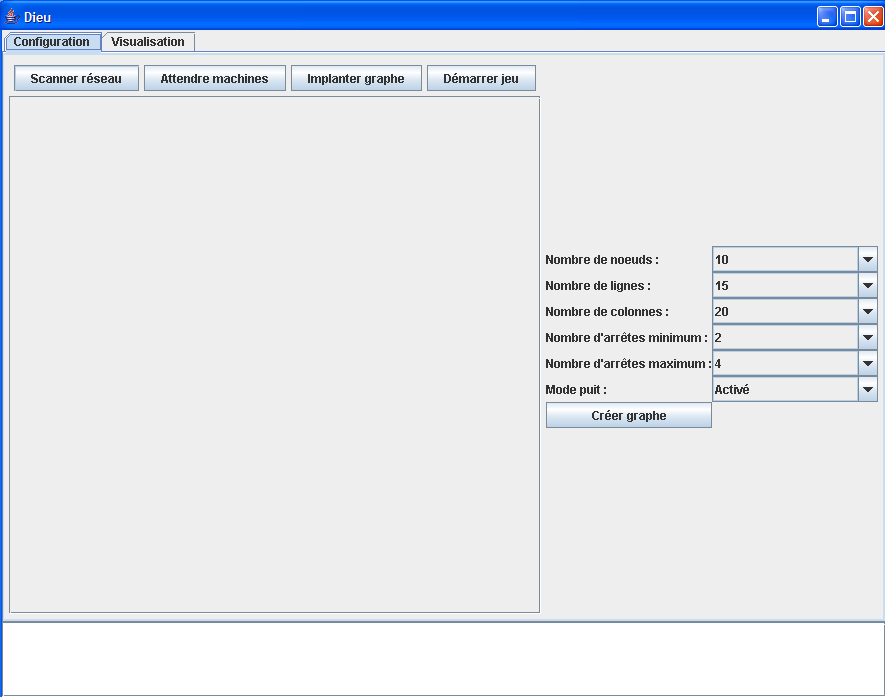
[6.3 Reconstruction d’un nœud 9](#_Toc199003157)

# Introduction

Notre objectif était de réaliser un système représentant une bataille entre thread. Le projet a été développé par une équipe de 6 personnes. Le projet aura pour but de mettre en œuvre les différents mécanismes vus en cours. Le projet reposera sur plusieurs parties notamment les entités du jeu, la création du graphe et son déploiement, les conditions de déroulement du jeu et une gestion des pannes. Ces parties seront présentées dans la suite de ce document.

# Présentation du jeu

## But



Le but du jeu est de remporter une bataille qui oppose plusieurs équipes constituées de thread. Pour cela, chaque équipe établit sa stratégie en décidant du comportement qu’auront ses threads face aux situations qu’ils peuvent rencontrer. Le Jeu se déroulera sur un graphe constitué d’un certain nombre de nœud. La fenêtre ci-dessus permet à l’un des joueurs de créer un graphe en lui donnant les options permettant son déploiement. Les joueurs auront la possibilité de visualiser le graphe et les déplacements des threads en sélectionnant l’onglet visualisation.

## Règles

Chaque équipe dispose d’un type de thread, ayant une stratégie qui lui est propre. Le jeu comporte deux catégories de nœud avec une base pour chaque équipe et des nœuds à capturer. La différence entre les bases et les nœuds est que les bases produisent à intervalles réguliers des threads pour appuyer l’armée des équipes.

Pour remporter une partie, les équipes ont plusieurs possibilités qui leur sont proposées :

-soit les deux équipes tentent de conquérir un maximum de nœud et le chronomètre de la partie se termine. Dans ce cas, l’équipe qui remporte la partie est celle qui a capturé le maximum de nœud.

-soit les deux équipes capturent les nœuds et au fil du jeu l’une des deux équipes se retrouve avec la totalité des nœuds de la carte en sa possession.

-soit l’une des deux équipes réussit à capturer la base de son adversaire.

# Entités du jeu

## Dieu

## Fidele

## Warrior

## Diagramme UML

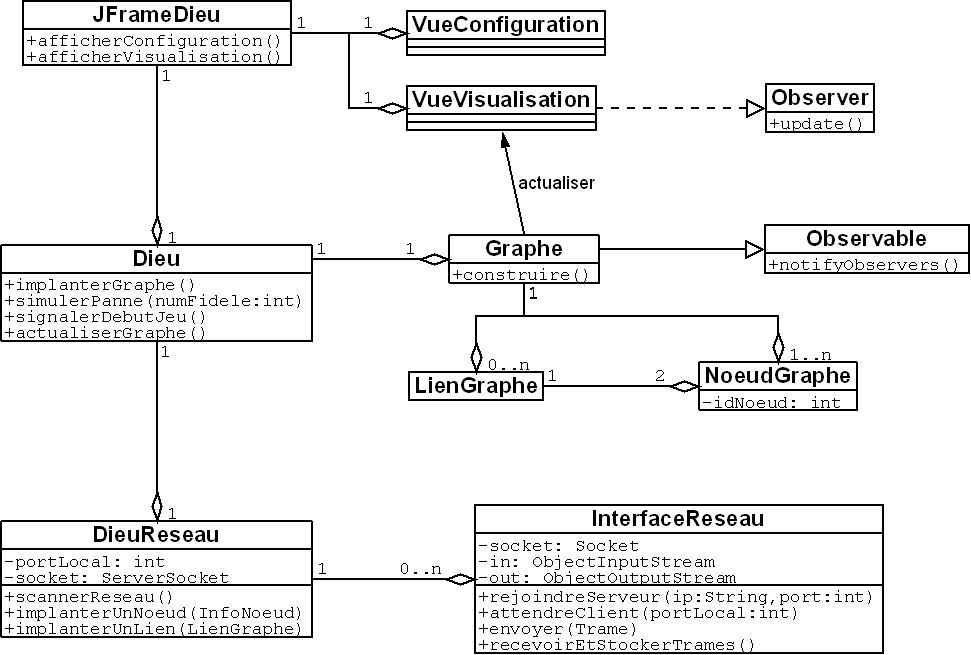


Figure : UML du programme DIEU

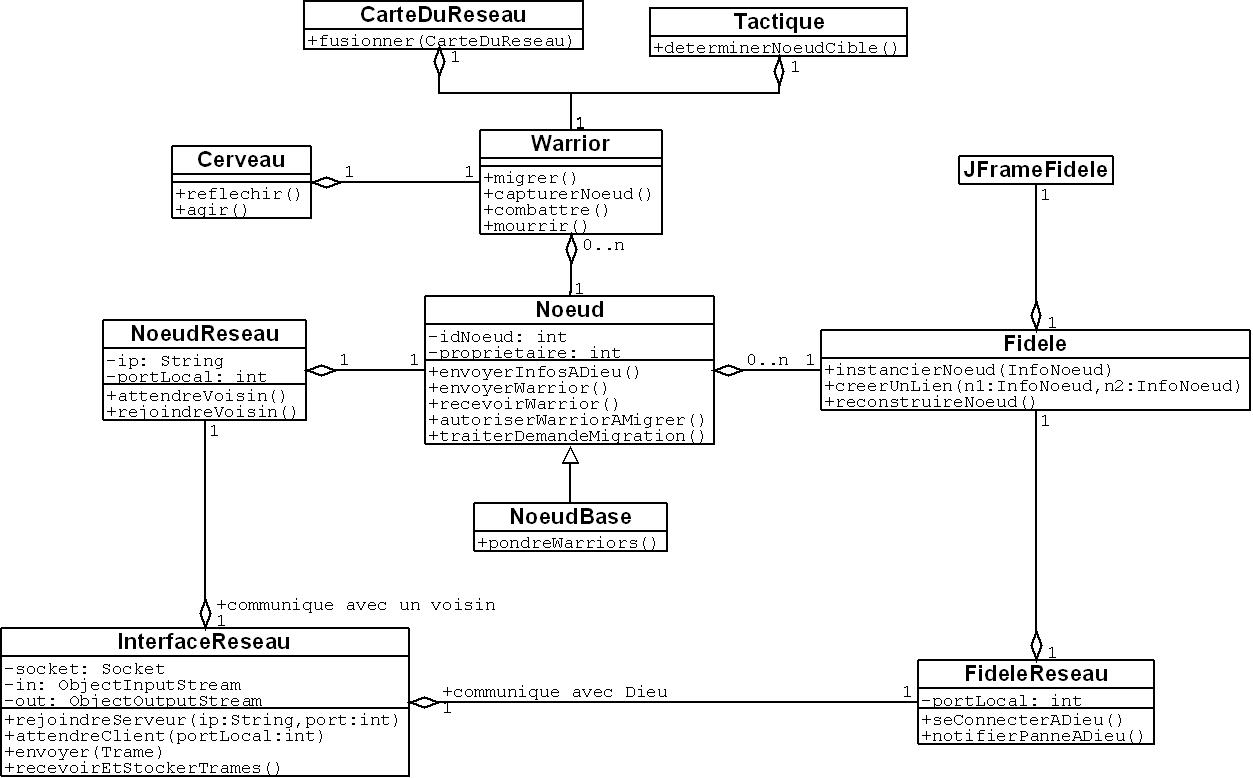


Figure : UML du programme Fidèle

# Création du réseau et déploiement

## Création du graphe

## Déploiement du graphe

## Visualisation du jeu

# Déroulement du Jeu

## Migration des warriors

## IA des warriors

Le déroulement du jeu nécessite un comportement intelligent des warriors (agents mobiles).Nous avons donc implémenté une IA respectant le schéma classique de réflexion d’une entité intelligente :

Cette IA va donc récolter des informations sur la carte présente et agir en conséquence.

### Récolte des informations

### Détermination d’une stratégie

Après la récolte des différentes informations sur la carte. L’IA va choisir un type de stratégie. Il en existe 4 :

* Stratégie indéterminée
* Stratégie Exploratrice
* Stratégie Offensive
* Stratégie Défensive

Le choix va se faire sur différent facteurs : le risque des nœuds aux alentours, le coefficient d’agressivité de l’IA du joueur et le nombre de nœud connus par le warrior.

1. La stratégie Indéterminée : elle permet au warrior d’évoluer sur le réseau de manière aléatoire au début du jeu ou dans le cas ou aucune autre stratégie ne serait applicable.
2. La stratégie Exploratrice : Cette stratégie choisit toujours d’explorer les nœuds inconnus : c’est-à-dire que parmi les voisins du nœud sur lequel se trouve le warrior, elle va choisir un nœud qui n’est pas présent dans la carte du warrior.
3. La stratégie offensive : Elle permet au warrior d’attaquer des nœuds qui possèdent un coefficient « risque » plus ou moins élevé : c'est-à-dire un nœud qui a été capturé par l’ennemi ou accueillant un ou plusieurs warrior ennemis.  
   Ce choix se fait parmi toute la carte en possession du warrior .La stratégie calcule donc **le chemin le plus court** pour accéder au nœud grâce à **l’algorithme de Dijkstra.**Si plusieurs warriors sont nécessaires pour la capture du nœud, le thread attendra un certains laps de temps que d’autres warriors de son camp viennent l’aider.
4. La stratégie offensive : Cette stratégie se met en place lorsque le risque de la majorité des nœuds connus est beaucoup trop élevé. Le warrior va alors défendre sa base puis les nœuds aux alentours.  
   De la même manière que la stratégie offensive les chemins sont calculés avec dijkstra.

Chaque stratégie définit donc une ou plusieurs actions qui sont :

* Migrer directement vers le nœud cible
* Migrer vers le nœud cible en passant par un chemin donné
* Attaquer le nœud
* Attendre l’arrivé de renfort

## Gestion des batailles

### Surveillance de nœuds

Chacun des nœuds composant le terrain du jeu, à sa création lance une fonction de surveillance, chargée d’analyser en permanence (à de très faibles intervalles) l’activité présente sur son nœud afin de juger si un combat est nécessaire. Cette surveillance s’effectue dans un thread externe, afin de ne pas interférer ou bloquer l’action du jeu courante.

Cette tache de surveillance consiste à parcourir l’ensemble des threads Warriors présents actuellement sur le nœud, d’observer leur joueur propriétaire et de contrôler leur état : veulent-ils capturer le nœud ou sont-ils seulement de passage. En fonction de ces informations, la fonction prend une décision :

* Se trouvent présents sur le nœud au moins deux Warriors de propriétaire différents, voulant capturer le nœud. On appelle alors la fonction de résolution de combat qui prendra le relais.
* Des Warriors veulent capturer le nœud, et ils sont tous du même camp : si ils respectent toutes les conditions requises pour capturer le nœud, alors celui-ci change de camp.
* Dans toutes autres situations, la fonction de surveillance n’agit pas et continue de surveiller.

### Combats de threads

Lorsque des Warriors d’au moins deux camps différents désirant prendre contrôle du nœud se trouvent côte à côte, un combat s’engage pour déterminer quel sera le camp gagnant, et quels seront les perdants.

Afin de décider l’issue de la bataille, chacun des Warriors de chaque camp lance un dé. Le nombre aléatoire de 1 à 100 sortants de ce lancer représente la force que les Warriors mettent dans la bataille. Après avoir fait la somme des lancers des Warriors pour chaque camp, on rajoute un bonus pour le camp possédant actuellement le nœud (dans le cas où le nœud n’est pas neutre et que le camp propriétaire fasse partie du combat en cours) : on ajoute un lancer de dés d’autant de fois que du montant de l’attribut ‘coefficient’ du nœud sur lequel l’action se déroule. Cet attribut sert à diverses choses, notamment à savoir la quantité de Warriors nécessaires pour capturer le nœud, ou comme ici de bonus au camp qui a déjà capturé le dit nœud.

Une fois que le compte est fait, il est facile de déterminer le vainqueur du combat : il s’agit du camp totalisant le plus grand score – s’il y a égalité, ce sera celui de plus petit identificateur.

Le camp gagnant capture le nœud. On parcourt une dernière fois la liste des Warriors, et tout ceux appartenant au camp ayant totalisé le plus petit résultat doivent mourir. Il n’est pas nécessaire de continuer à se battre jusqu’à ce qu’il ne reste plus qu’un camp : si cela est nécessaire la fonction de surveillance du nœud rappellera la fonction de résolution de combat.

Nous avons choisi de résoudre les batailles de cette manière, car elle offre un grand intérêt stratégique ainsi qu’un bon compromis : plus un camp dispose de Warriors de son côté et plus elle a de chances de remporter la bataille ; mais rien n’est jamais joué à l’avance et la chance et le hasard interviennent tout de même pour rendre les bataille toujours intéressantes et inattendues. De plus, grâce au bonus apporté au camp propriétaire, plus un nœud est difficile à capturer, et plus le camp gagnant aura de facilité à le défendre et de chance de conserver sa propriété.

### Capture du nœud

Lorsqu’un combat se termine ou qu’un camp seul sur un nœud décide de le conquérir, il faut alors décider si le nœud est capturé ou non.

L’attribut ‘coefficient’ du nœud sert principalement à cela : il représente le nombre nécessaires pour contrôler le nœud. Si la fonction de surveillance ne détecte que des Warriors d’un même camp, et que ces Warriors veulent capturer le nœud courant, alors elle compare leur nombre au ‘coefficient’ du nœud ; si ils sont supérieur ou également nombreux, alors le nœud prend comme propriétaire celui de ces Warriors, et cela qu’il soit neutre ou appartienne déjà à un camp adverse.

Dans le cas d’un combat de Warriors, c’est assez différent : la fonction de résolution de combat, à l’issue de la bataille, donne le nœud au camp victorieux, et cela sans prendre en compte le nombre de Warriors ou le ‘coefficient ‘ du nœud. Le fait de s’être battu à la fois les un contre les autres mais aussi parfois contre le terrain lui-même affranchit les joueurs d’une telle règle. Cela peut-être très intéressant d’un point de vue stratégique…

# Gestion des pannes

## Simulation d’une panne

## Détection des pannes

Pour détecter une panne, nous avons mis en place un système pour migrer un thread d’un nœud A vers un nœud B. La migration d’un thread nécessite l’envoie d’une demande d’autorisation de migration. Si la demande a bien été reçue, le nœud B va envoyer une réponse au nœud A. S’il n’y a aucune réponse, alors nous considérons que le nœud B est en panne.

Quand la panne est détectée, le nœud qui l’a découvert envoie une trame à Dieu pour le prévenir de l’incident. Cette trame contient l’identifiant du nœud en panne. Une fois que Dieu est informé de la panne, il envoie une trame à chaque voisin du nœud en panne ainsi qu’au fidèle sur lequel était le nœud en panne pour leur spécifier que le nœud d’identifiant k est en panne. A ce moment là, il y a une partie de l’IA qui s’occupe de traiter comment les voisins du nœud en panne vont traiter les threads notamment pour les migrations. Le but étant d’éviter de faire migrer un thread vers un nœud en panne et de surcharger le réseau en prévenant Dieu à plusieurs reprises que qu’un nœud k est en panne.

## Reconstruction d’un nœud

La reconstruction du nœud est faite par le fidèle sur lequel se trouvait le nœud avant de tomber en panne. Pour cela Dieu va lui envoyer une trame avec l’identifiant du nœud et la valeur du propriétaire avant la panne. Le fidèle reconstruira un nœud identique à celui en panne c'est-à-dire en respectant les liens qui le reliaient à ses voisins.