Organisation de l‘émulateur

Petit résumé du découpage de l‘émulateur, ainsi que de son fonctionnement.

Packages

- common : Contient les objets communs entre le serveur de jeu et de connexion
- core : Framework de l‘émulateur
  - config : gestion de la configuration
  - dbal : gestion de la base de données
  - di : conteneur d‘injection
  - events : event dispatcher et listeners
- data : Gestion de la base de données
  - constant : Contantes et enums utilisées en base
  - living : Gestion des données dynamiques (qui changent)
    - constraint : Contraintes sur les données (équivalent à un formulaire)
    - entity : Objets représentant la donnée en base, organisés en sous packages
    - repository : DAOs gérant les entités dans les bases de données
    - transformer : Normalise et dénormalise les données en base
  - transformer : transformers communs
  - value : Contient les value objects
  - world : Données statiques (données de l‘univers, qui ne changent pas)
    - entity : Comme pour living (mais en immutable)
    - repository : Comme pour living
    - transformer : toujours pareil
- game : Contient tout la logique du jeu (objets du domaine, services, intéractions…) **Non exhaustif**
  - handler : Contient les gestionnaires de packets entrant (équivalent à un contrôleur)
  - listener : Tout les listeners, permettant de gérer la communication inter-services et l‘envoi de packets
  - world : objets commun au game **déprécié**
- network : Gestion du réseau et des packets
  - adapter : Adapte le système de packets de l‘émualteur à une bibliothèque réseau
  - game : Gestion des packets du jeu
    - in : packets entrant et parsers
    - out : packets sortant et formatteurs
  - in : packets communs entrant et parsers
  - out : packets communs sortant et formatteurs
  - realm : Gestion des packets du serveur de connexion
    - in : packets entrant
    - out : packet sortant
- realm : Logique du serveur de connexion
- util : fourre-tout de tool box

Gestion des données

La gestion des données de l'émulateur est assez particulière et précise, et ne correspond pas totalement à une gestion de données plus classique. Elle reste assez bas niveau, et toute la gestion plus haut niveau (par exemple relations) sont faite à part.

Entity

Les entity sont des objets le plus simple possible, ne servant que de simple conteneur des données en base. Ils ne contiennent que des attributs, avec getters et setters, ainsi que des factory method si nécessaire.

Elles ne doivent pas gérer les relations, et doivent donc stocker directement les clés étrangères plutôt que les entitées liées.

Les champs constants, tels que les clés primaires, ou valeurs ne devant jamais changer doivent être mis en **final**. Dans le cas de la clé primaire en **autoincrement**, une méthode withId, ou setId doit être faite, qui va recréer l'objet avec la valeur de l'id, mais en aucun cas le champ id doit être modifiable.

Un constucteur initalisant tout les champs doit être présent. Différents constructeurs, ajoutant des valeurs par défaut peuvent être ajoutés pour simplifier l'instantiation, ou la rétrocompatibilité avec les tests.

Repository

Les dépôts gère les entitées décrites plus haut. Un dépôt ne gère qu'une seule et unique entitée, et est le seul à la gérer. C'est lui qui contient les méthodes de lecture et d'écriture en base.

Chaque dépôts doit **obligatoirement** avoir son interface contenant **la totalité** des méthodes. Seule l'interface est accessible à l'extérieur (et donc utilisable). L'implémentation doit se trouver dans un package, avec une visibilité „défaut" (visible qu'à l'intérieur du package). Il existe actuellement deux types d'implémentation :

- sql : pour gérer via une base de données sql
- local : pour sotcker les données en cache dans java

L'implémentation est choisie par le module des dépôts.

Les dépôts doivent implémenter le minimum de méthodes. Si une méthode est non nécessaire, elle ne doit pas être ajouté „au cas où", ni pour respecter l'interface (à l'exception de l'interface Repository).

Pour l'implémenation des dépôts SQL, il faut déclarer un Loader, contruisant l'entitée avec le ResultSet (en utilisant potentiellement un ou plusieurs transformeur), et remplissant les clés générées (en cas d'insert, avec auto-increment). La méthode initialize fait le create table (actuellement utilisé pour les tests, donc format SQLite), et le destroy s'occupe quand à elle du drop (idem, pour les tests en SQLite). Puis chaques méthode sont implémenter en utilisant directement des requêtes SQL. **Toutes les requêtes nécessitant des paramètres doivent être préparées, et aucune concaténation d'est autorisée !** Les dépôts SQL ne doivent pas gérer de cache, celui-ci doit être gérer par un autre dépôt **si nécessaire**.

Les dépôts de cache quant-à eux doivent tout déléguer à l'implémentation réelle. Ils doivent être totalement transparent à l'utilisation (pas de modification de la données ni du comportement). Le cache doit être utilisé avec parcimonie : il peut être source de bug, de problèmes de synchronisation, ainsi que de fuite mémoire. C'est pourquoi il est fortement déconseillé de l'utiliser sur des données dynamiques. Les performances de MySQL sont largement correct s'il est utilisé

intelligemment. De plus les requêtes complexes sont difficiles à mettre en cache, et peut être contre-productif.

Transformer

Ce n‘est pas un vieau dessin animé pourris, ou un film tout aussi médiocre uniquement là pour en mettre plein la vu avec plus d‘effets spéciaux que d‘acteurs…

Le transformer permet de normaliser et dénormaliser la données en base. Il permet ainsi que gérer des champs complexes sans avoir à créer 50 tables. C‘est par exemple `ItemEffectsTransformer` qui s‘occupe d‘analyser et de sauvegarder les stats des items.

Création d‘une nouvelle entité

Voici une petite procédure pour créer une nouvelle entité :

- Créer l‘entité avec tout les attributs nécessaires
- Créer l‘interface du dépôt avec les méthodes qui seront utilisées
- Créer l‘implémentation SQL
- Enregistrer l‘implementation dans le module
- Faire les tests unitaires du dépôt
  - toutes les méthodes doivent être testés !
  - L‘entité n‘ayant pas de code, il est inutile de la tester
- Tester le module : le bon dépôt doit être correctement instancier
- Si pertinent, créer le dépôt de cache :
  - Créer l‘implementation de cache
  - La tester unitairement
  - La déclarer dans le module

Au contraire des méthodes du dépôt, lors de la création de l‘entitée, elle doit être la plus complète possible, en déclarant la totalité des champs qui seront nécessaire dans le future, le changement d‘une entitée étant assez complexe à réaliser.

Gestion du réseau et des packets

Description du protocol de Dofus

Dofus 1.29 utilise un protocol texte, en passant par du TCP/IP. Les échanges entre le serverur et le client passent par des packets. Les packets **entrant** sont délimités par la suite de caractère “LF NUL“ (i.e. en C like “\n\0“), et les packets **sortant** se terminent par “NUL“ (“\0“). Les packets n‘ont pas de taille maximale. Les packets sont composés de deux parties : l‘entête et les données.

L‘entête permet de savoir qu‘elle action appliquer, c‘est l‘équivalent d‘une commande. Sa taille est variable, de 2 à 5 caractères (avec une majorité de packets à 2 caractères) , contenant exclusivement des lettres, minuscules ou majuscules (WC et Wc ne sont pas les même packets!).

L‘entête n‘ayant pas de taille fixe et de séparateur, il faut parcourir toutes les entêtes possibles pour savoir laquelle correspond: il n‘est pas possible d‘utiliser une table avec la forme header → action. La meilleur solution pour l‘analyse de l‘entête est de passer par un arbre lexicographique, la taille de l‘entête étant limité (profondeur de l‘arbre), ainsi que le charset (largeur de l‘arbre). De plus cette structurte permet de coller à la sémantique des packets Dofus, car chaque niveau de lettre permet de présiser l‘action et ainsi créer des famille de packets (ex: GKK est plus proche de GKE que de GI, GI est plus proche de GC, que de AA, etc…; On peut ainsi créer les familles des packets “G“ et “A“). De manière générale (tout les packets ne suivent pas cette logique !), la première lettre désigne de domaine sur lequel agir, la deuxième est l‘action à effectuer, et la troisième (si présente) pour préciser le comportement de l‘action. Par exemple pour les packets GKK et GKE :

- G représente le domaine “Game“ : c‘est une action à effectuer dans le jeu
- K pour action terminée (acquitement)
- K pour terminée avec succès, et E pour annulée

Les différents domaines sont (non exchaustif) :

- H pour la connexion au serveur
- B pour différentes actions
- A pour le compte et personnage courant
- G pour les actions dans le jeu
- c pour le chat
- D pour les dialogues avec le PNJ
- I pour les infos du monde
- S pour la gestion des sorts
- O pour la gestion des objets
- F pour la liste d‘amis
- i pour la liste d‘enemis
- J pour les métiers
- E pour les échanges
- h pour les maisons
- s pour les coffres
- e pour les emotes
- g pour les guildes
- W pour les zaaps
- f pour les combats
- Q pour les quêtes
- P pour les groupes
- R pour les montures

Attention : les packets entrant ne correspondent pas toujours avec ceux sortant !

Les données (ou paramètres) du packets sont situées directement après l‘entête. Il n‘y a pas de format bien définie, mais une structure globale peut être tiré :

- Le premier caractère peut valoir K pour un succès et E pour une erreur (ex: AAE pour une erreur lors de la création du personnage, AAK si le personnage est bien créé)
- Souvent les erreurs sont complétées par un caractère définissant le code d‘erreur. Si ce caractère n‘est pas présent, c‘est pour une erreur indifinie (ex: AAEa pour création de personnage avec nom déjà pris)
- Les différentes sections sont séparées par des pipes “|“, les sous-sections par des point-virgules “;“, puis les virgules “,“, et enfin des tildes “~“. (ex: GDM|123|456|key pour les données de la map 123)
    - Certains packets sont séparés par des “;“ au lieu de “|“
    - Certains packets peuvent avoir le pipe “|“ directement après le header, ou non. Il faut bien faire attention à ça !
    - Dans la majorité des cas, les listes sont séparées par des pipes “|“
- Dans les listes de données permettant de modifier les objets dans le jeu, un caractère modifieur est présent au tout début de chaque sections:
    - + Pour ajouter un objet
    - - Pour supprimer l‘objet
    - ~ Pou modifier l‘objet
- Les données numériques de faible valeurs sont encodées en décimale, pour celles plus complexes en hexadecimale

Les packets qui transitent peuvent être chiffrés, en utilisant un simple XOR, et en encodant dans leur implémentation propriétaire du Base64 (Voir util.Base64). Le chiffrement n‘est actuellement pas supporté par l‘émulateur.

Packets sortant

Chaques packets sortant sur l‘émulateur a sa propre classe, et se trouvent dans le package “out“. Ces objets doivent être immutable, et doivent contenir la méthode toString. Leur but est de formater les objets java pour générer le packet. Pour simplifier leur instantiation, il est aussi possible d‘ajouter des factory methodes.

Packets entrant

Les packets entrant quant-à eux se trouvent dans le package “in“. Il doivent implémenter l‘interface Packet, et doivent contenir une classe interne Parser implémentant SimplePacketParser. Ces objets sont eux aussi immutables. Le parser transforme la données du packet en valeur java, et le packet stocke ces données. Le parser peut envoyer une exception si le packet est mal formé.

Pour pouvoir enregistrer le packet dans l‘émulateur, celui-ci doit être ajouté dans le ParserLoader de son package (GameParserLoader ou RealmParserLoader).

Packet Handlers

Les packets handlers (game.handler, ou realm.handler) sont les points d‘entrée de

l‘application gérant les packets entrant. Ils sont l‘équivalent d‘un contrôleur sur un site MVC. Ils écoutent un type de packet (définie par la classe), et reçoivent le packet, ainsi que la session courante. Comme un contrôleur, ils valident les données du packet, le traite et envoi la réponse au client. Les gestion des erreurs peuvent être fait de différentes façopns :

- Via une exception ErrorPacket qui envoie un packet d‘erreur, et arrête le traitement
- Via une exception CloseImmediately qui ferme la session sans envoyer de packet
- Via une exception CloseWithPacket qui envoi un packet et ferme la session
- Manuellement (attention à ne pas oublier le **return** !)

Le packet handler doit être sans état (ne doit pas contenir de variables mutable), et ne prends en paramètres que des services. Sont rôle est d‘être un pont entre le client et les services.

Ils sont regroupés par domaine (celui-ci peut différer du domaine du packet !), et ils sont enregistrés dans les différents Loader. Pour gérer la sécurité (par exemple qu‘un simple joueur ne puisse pas avoir accès à la console), ils sont enveloppés dans différents handler de sécurité, nommées EnsureXXX (ex: EnsureAdmin pour tester si le joueur est admin, EnsureFighting pour tester si le joueur est dans un combat…). Ces enveloppes sont ajoutées automatiquement par le Loader.

Envoi de packets

La structure est assez stricte pour ce qui est de l‘envoi de packets :

- Ne **jamais** envoyer le packet en mode **string**. Le packet doit **toujours** être un objet
- Les packets en réponse d‘un packet entrant doivent être envoyés par le handler
- Les packets envoyés par synchroniser le client avec les données serveur doivent être envoyés par les listeners
- Seul un nombre restreint de classes peuvent envoyer des packets :
  - Les handlers
  - Les listeners
  - Les intéractions (game action, fight action)
  - Les effects handler (fight)

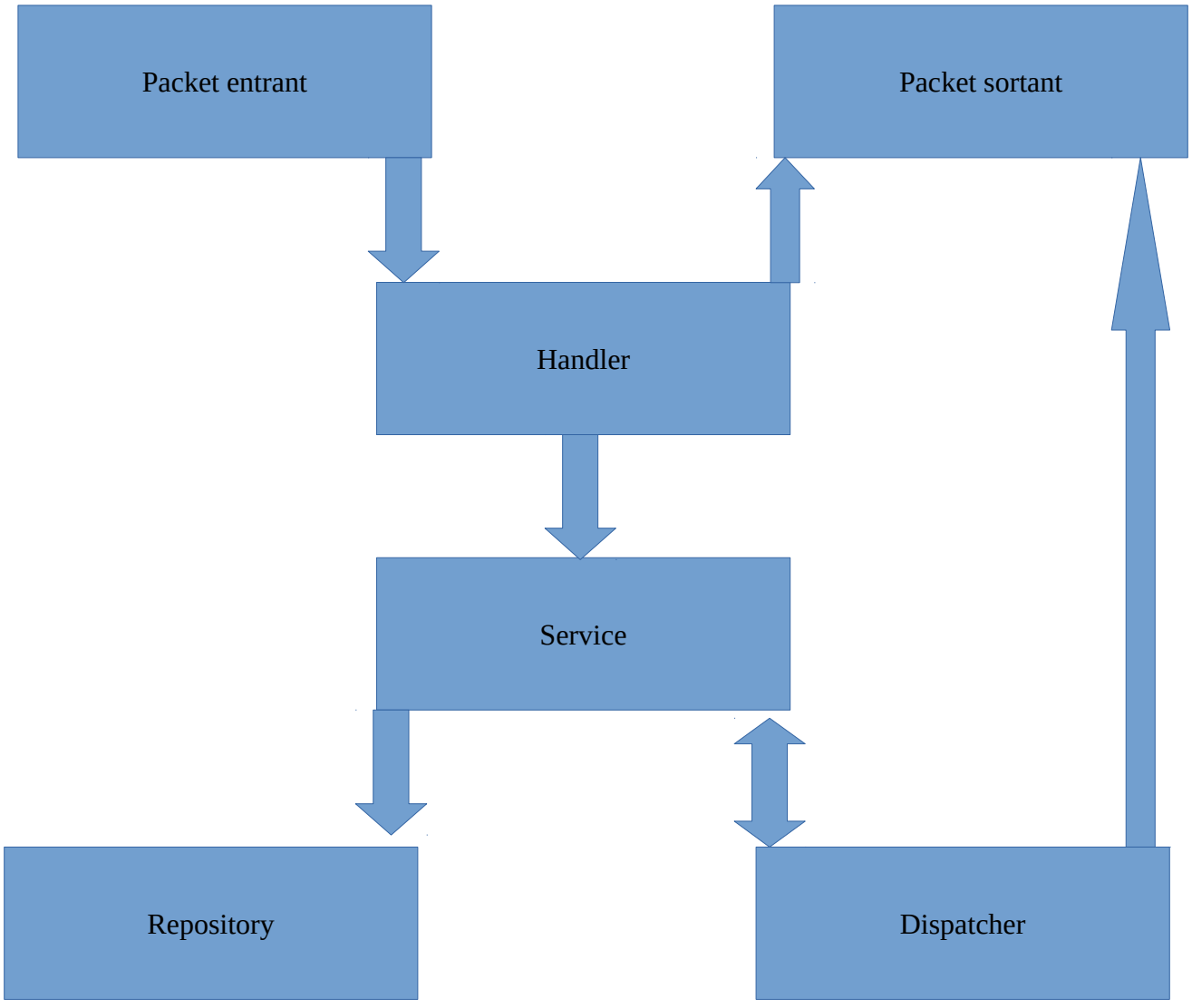
Procédure pour gérer un packet (entrée + sortie)

- Créer le packet entrant et sont parser
- L‘enregistrer dans le ParserLoader
- Ajouter un test unitaire du parser (il faut aussi tester les erreurs !)
- Compléter le test du ParserLoader pour voir s‘il a bien pris en compte le packet
- Créer le packet sortant
- Tester le packet sortant (y compris s‘il se résume à un simple return d‘une constante)

- Ajouter le packet handler
- Tester le packet handler (test fonctionnel)
- Enregistrer le packet handler dans le loader
- Tester que le loader a bien enregistré le handler
- Lancer tout les tests, et voir la couverture du code : elle doit être à 100% si possible
- Ne pas oublier de tester directement en jeu ;)

Services et objets du domaine

Les services et objets du domaine sont le coeur de l'émulateur. Ce sont eux qui gère toute la logique et les contraintes du jeu. Ils se situent entre le reseau (handlers et packets), et la base de données (entity et repository). Ils sont donc appelés par les handlers, ils appellent le base de données et dispatch des events.



Service

Le service est à l‘objet du domaine, ce que le dépôt est à l‘entitée : c‘est lui qui gère, et crée et objets du domaine. Le service est donc unique au sein de l‘application, et n‘est pas stateless.

Il prend en paramètre les dépôts liés à son domaine (et uniquement lié à son domaine), et si nécessaire le dispatcher et la configuration. Il ne doit par contre pas prendre d‘objets ou de services en dehors de son domaine.

Pour communiquer entre les différents services, des listeners doivent être enregistrés pour écouter les changement sur les autres domaines, et des events doivent être dispatch en cas de changement. Pour simplifier l‘enregistrement des listeners, le service peut implémenter l‘interface EventSubscriber, et ainsi enregistrer ses listeners.

Pour initialiser le service au lancement de l‘émulateur, l‘interface PreloadableService avec la méthode preload peut être implémentée. C‘est de cette façon que les maps sont par exemple préchargée au lancement.

Pour enregistrer le service dans l‘émulateur, pour qu‘il puisse être utilisé par un handler, il suffit de l‘enregistrer dans le GameModule ou RealmModule. Le service doit être enregistré en “persist“. S‘il est **preloadable**, il faut l‘enregistrer dans le 6e paramètre du constructeur du GameService (attention à l‘ordre !), et s‘il défini des **listeners**, il faut l‘ajouter dans le 7e paramètre.

Objets du domaine

Les objets du domaine sont créés par un service. Ils contiennent les entitées, ainsi que les différentes relations. Ils ont aussi les différentes méthods permettant de faire vivre l‘objet. La majorité des actions du domaine sont implémenter dans les objets du domaine. Les services quant-à eux servent principalement pour gérer les intéractions entres les différents objet, et faire des actions du des collections d‘objets (par exemple équiper un item se fait sur les objets du domaine, mais la création et la récupération de cet objet est fait par le service).

Ces objets peuvent avoir accès au service (enregistré dans un attribut), et ne sont pas stateless.

Interactions et actions du jeu

Dofus est basé sur ce qui est appellé des Game actions. Ce sont elles qui sont à l‘origine des déplacement, et autres actions du personnage. Ces actions sont communes au combat ainsi qu‘à l‘exploration, mais elles sont séparées au niveau de l‘émulateur.

Intéractions

Les intéractions sont ce que le personnage fait avec les autres objets du monde. Ceci peut être un échange, un dialogue, etc… Il ne peut y avoir qu‘une seule intéraction par personnage, et celle ci doit être arrêtée si le joueur veut faire une autre action, ou que les autres joueurs veulent intéragir avec lui. Il est par exemple impossible de demander un défie si le joueur est en plein échange avec un autre joueur. L‘intéraction peut être démarrée, arrêtée, et selon le type d‘intéraction, celle-ci peut être acceptée ou refusée (dans le cas d‘une demande, comme un défie).

Les intéractions peuvent être démarrées via une GameAction, mais pas uniquement.

GameActions

Les GameActions représente les actions que font notre personnage. Il existe deux types de GameActions : les non bloquantes, et les bloquantes. Dès qu‘une GameAction bloquante est lancée, celle-ci va bloquer les actions suivantes qui seront mises en queue, et seront exécutées dès que l‘action sera terminée. Si l‘action bloquante est annulée, toutes les actions suivantes seront elles aussi annulées.

Système de combat

Le combat est une grosse partie du jeu. Dès qu‘un combat est lancé, il n‘est plus possible d‘avoir d‘autres intéractions.

Création du combat

Pour créer un combat, il faut au moins 2 équipes, deux combatants, une carte et un type de combat. Le type de combat définie les actions de fin de combat, ainsi que la configuration du combat (annulable ? Temps de placement ? Gains de fin de combat ?). La création passe par un builder, sachant qu‘il y a un builder par type de combat.

Une fois le combat créé, on enregistre les différents modules permettant de le compléter (ajoute des effets, des events…). Il est ensuite ajouté à la carte, puis initialisé en passant par les différentes états du combat.

États du combat

Le combat est successivement dans différents états :

- Null pendant la création du combat : cet état ne fait rien
- Initialise : premier vrai état du combat, celui-ci initialise les données du combat, et passe à l‘état suivant
- Placement : c‘est l‘état pendant lequel l‘ont peut se placer et rejoindre le combat
- Active : le combat à commencé, on peut lancer les sorts, se déplacer, etc…
- Finish : le combat est erminé, plus aucunes actions n‘est possible. Les récompenses sont données, les stats mises à jour, et les combatants rejoingnent la carte en mode exploration

Tour et actions de combat

Dès que le tour commence, le combatant peut effectuer des actions. Ce sont des GameAction, et elles sont au nombre de 3 : Déplacement, Cast de sort et Corps à corps. Dès que le tour commence, les points de mouvement et d‘action sont initialisés, et les actions de début de tour sont déclenchées. Les effets des actions ne sont appliqués qu‘à la fin de cette action. Et la fin de tour n‘est déclanchée qu‘une fois l‘action en cours est terminée.

Les actions passent par 3 phases :

- Validation de l‘action : vérifie si l‘action est possible en fonction du contexte
- Début de l‘action : l‘action à démarrée, et va donc quoi qu‘il arrive utililser des PA / PM. Le début de l‘action permet de savoir si l‘action à réussie, si c‘est critique, ou un echec, puis l‘animation démarrera
- Fin de l‘action : c‘est là que les effets sont appliqués : le combatant est déplacé en cas de déplacement (avec déclanchement des pièges), ou les effets de sorts sont appliqués.

Effets de sorts et buffs

L‘application des effets de sorts suit le schéma suivant :

- La zone d‘effet est résolue : la liste des cases touchés est générée
- En fonction de cette liste, ainsi que des cibles du sort, la liste des combatant touchés est générée
- Avec ces deux paramètres, en plus des paramètres de cast (sort, effet, case cible, caster), un effect scope est créé
- Une effect handle est résolu en fonction de l‘effect id
- Si la durée de l‘effet est différente de 0, un buff est appliqué
- Sinon l‘effet est appliqué directement par l‘handler

L‘effect handler doit parcourrir la liste des cibles, et appliquer l‘effet. En cas de buff, il doit aussi enregistrer les liste des différents effets du buff.

Une buff permet d‘enregistrer différents hooks qui sont appellés à différents moment du combat. L‘effect handler doit donc définir l‘action à effectuer pour chaque hooks. Par exemple en cas de poison, l‘effet doit écouter le début du tour, pour appliquer le poison, et si la cible meurt, son tour doit être terminé.

Pour créer un nouvel effet de sort il faut :

- Créer l‘effect handler
- Enregistrer l‘effect handler dans un module
  - Si c‘est une effet simple sans dépendences, l‘ajouter dans CommonEffectsModule
  - Si différents events ou dépendences sont nécessaire, créer un nouveau module
- Tester unitairement l‘effet
- Tester le module (si créé)
- Tester fonctionnellement l‘effet (dans fight.castable.effect.FunctionalTest)

Tests

Les tests sont une partie très importante de la qualité de l‘émulateur. Il permettent d‘avoir un regard différent sur le code, et permet de juger son utilisation. C‘est pourquoi **tout code ajouté ou modifié doit être testé**. Pour siplifier la création des tests un petit framework a été mis en place.

Test du jeu

Pour tester le jeu, la classe GameBaseCase a été créée. Celle-ci configure les différents composants du jeu, crée une base de donnée temporaire, et mock la partie réseau. Elle ajoute aussi quelques méthodes permettant de simplifier la création d‘objets du domaine courants, tels que des personnages.

Pour faire le test, il faut initialiser les données dans le setUp (qu‘il faudra bien surcharger), instancier l‘élément à tester, puis tester les différentes méthodes de l‘objet. Lors du tearDown, la base de données et l‘application sont complètement détruites. A moins que l‘objet a tester ait des effets de bords, il n‘y a généralement pas besoin de faire des actions dans le tearDown.

Test de combat

Pour tester les combats, la classe FightBaseCase a été créée. Elle hérite de GameBaseCase, en ajoutant différentes méthode pour créer facilement un combat.

Gestion des données

Pour gérer les données, la classe GameDataSet permet de créer les différentes données en base. Il est ainsi possible de créer une donnée en donnant une instance de l‘entitée à créer. D‘autres méthodes permettent quant-à elles de pousser des données déjà créées, et ainsi les réutiliser dans différents tests (par exemple des sorts ou des maps). Cette classe contient les méthodes suivantes :

- declare : permettant de lier une entité à un dépôt La déclaration doit être faite **obligatoire** avant d‘utiliser l‘entité
- use : pour initialiser le dépôt et créer la table
- push : pour ajouter une entitée en base. Il est possible de lui associer un identifiant pour la récupérer ultérieurement
- get : pour récupérer l‘entitée poussé avec un identifiant
- refresh : recharge l‘entitée depuis la base de données