Les bases

Table des matières

Table des figures

I – Fonctionnalités générales du framework

Tous les modules du framework se reposent sur des fonctionnalités de base grâce à l'héritage. Le fonctionnement de *Godot Mega Assets* est basé sur le langage orienté objet (POO). Tous les modules héritent d'une classe de base s'appelant **MegaAssets**. La classe **MegaAssets** possède une multitude de fonctions très utiles et interressantes dont le développeur pourra s'en servir pour aller plus vite dans ces programmations. Elle est la classe mère de tous les modules et représente le pilier centrale du framework. Le développeur peut également créé des classes, toutes dérivées de la classe **MegaAssets**.

II – Les types de module

Godot Mega Assets répartit les modules en deux types :

+ Les modules indestructibles ;
+ Les modules destructibles.

En effet, les modules indestructibles sont ceux qui ne se détruisent pas lorsqu'on change de scène au cours de l'exécution du jeu. Ces types de module ne peuvent en aucun cas être instanciés plus d'une fois, car leur fonctionnement est assez générale et globale. Prenons l'exemple du système de gestion des différentes langues du jeu. Le développeur dans le développement de son jeu a prévu (03) langues possibles dans son jeu : Le Français, l'Anglais et l'Espagnol.
L'utilisateur de son produit choisi la langue dont-il comprend. Ainsi, partout où l'on irra dans le jeu : les menu, les sous-titres etc..., seront mise à jour en fonction de la langue choisi par l'utilisateur. Pour avoir un tel résultat, il faut que le gestionnaire de langues soit présent dans toutes les scènes du jeu pour pouvoir mettre à jour les éléments concernés et ceci de façon automatique. D'où la néccessité de modules indestructibles. Avec notre analyse nous pouvons affirmer que le gestionnaire de langues est donc un module indestructible.
Les modules destructibles sont ceux qui accompagnent la scène où ils ont été définit. Ils seront donc détruit lorsqu'on changera de scène. Leur champs d'activité est limité à la scène dans laquelle ils se trouvent. On peut les instanciés autant de fois que l'on souhaite.

III – Les modules sauvegardables

Dans les types de modules que nous avons évoquer plus haut, certains sont sauvegardables et d'autres pas. Vue la nature et le fonctionnement de certains modules, il serait interressant de permettre aux développeurs de pouvoir les sauvegardés, puis charger leurs données lorsqu'ils en auront bésoin. Cela permettra à ces derniers de ne plus enregistré les données un à un, même si cette méthode reste possible. Ils leur suffira juste d'appeler une fonctionnalité prédéfinie dans le module pour cela. Cependant, les modules sauvegardables possèdent en plus des fonctionnalités de base, celles qui permetteront aux développeurs de pouvoir gérer facilement les données de ces derniers.

IV – Les propriétés de base d'un module

Tous les modules de ce système possèdent des propriétés de bases. Cependant, la présence de certaines propriétés sont spécifiques à la nature et au fonctionnement du module en question.
+ bool Enabled = true : Contrôle l'état de fonctionnement d'un module (ON/OFF).

+ bool AutoCompile = true : Contrôle la vérification des valeurs des différentes entrées d'un module. Cette option est uniquement disponible en mode édition.

+ int DataManager = 0 : Quel gestionnaire de données voulez- vous choisir ? C'est en fonction de ce dernier que les données du module en question seront chargées et/ ou sauvegardées. Cette option n'est disponible que sur les modules sauvegardables. Les valeurs possibles sont :

-> **MegaAssets.GameDataManager.NONE ou 0 :** Ne sélectionné aucun gestionnaire de données.
-> **MegaAssets.GameDataManager.GAME_SAVES ou 1 :** La sauvegarde et le chargement des données s'éffectueront en utilisant le module *SaveLoadFx*.
-> **MegaAssets.GameDataManager.GAME_CONFIGS ou 2 :** La sauvegarde et le chargement des données s'éffectueront en utilisant le module *SettingsFx*.

+ String Checkpoint : Sur quel point de sauvegarde, les données du module seront sauvegardées et/ou chargées ? Notez que la précision de ce dernier n'est pas obligatoire. Dans ce cas, le point de sauvegarde actif sur le module *SaveLoadFx* sera pris pour cible. **N'utilisez ce champ que si le gestionnaire de données choisi, cible le module *SaveLoadFx***. Cette option n'est disponible que sur les modules sauvegardables.

+ String Section : Sur quelle section de configuration, les données du module seront sauvegardées et/ou chargées ? Notez que la précision de ce dernier est fortement obligatoire. **N'utilisez ce champ que si le gestionnaire de données choisi, cible le module *SettingsFx***. Par défaut, un nom de section de configuration est automatiquement généré par le module. Vous pouvez la changée en ce que vous voulez. Cette option n'est disponible que sur les modules sauvegardables.

+ String GlobalKey : Définit l'identifiant global à utilisé lors des sauvegardes et des chargements de données. Par défaut, une clé est générée par le module. Vous avez la possibilité de la changé en ce que vous voulez. Le rôle de cet attribut est de permettre la mise à jour du gestionnaire de données avec les données de plusieurs instances du même module de façon indépendante. **Notez que le remplissage de ce champ est fortement obligatoire**. Cette option n'est disponible que sur les modules sauvegardables.

+ bool SaveData = false : Contrôle la sauvegarde des données au sein d'un module. En d'autres termes, Voulez-vous sauvegarder automatiquement les données du module à chaque fois que l'on sauvegardera le gestionnaire des données du jeu ? Cette option n'est disponible que sur les modules sauvegardables.

+ bool **LoadData** = false **:** Contrôle le chargement des données au sein d'un module. En d'autres termes, Voulez-vous charger automatiquement les données au démarrage du jeu ? Notez qu'a chaque fois que l'on chargera un fichier de sauvegarde ou que l'on changera de point de sauvegarde, le module se mettra automatiquement à jour. Cette option n'est disponible que sur les modules sauvegardables.

+ int **ActivityZone** = 2 **:** Enumération conditionant l'environement d'exécution d'un module. Cette option restreint le champ d'exécution d'un module. Les valeurs possibles sont :

-> **MegaAssets.ActivityArea.EDITOR_ONLY ou 0** : Le module s'exécutera uniquement dans l'éditeur.

-> **MegaAssets.ActivityArea.RUNTIME_ONLY ou 1** : Le module s'exécutera uniquement lorsque le jeu sera en cours d'exécution.

-> **MegaAssets.ActivityArea.BOTH ou 2** : Le module s'exécutera dans l'éditeur ainsi que dans le jeu.

+ int **EventsScope** = 1 **:** Enumération contrôlant la portée des événements d'un module. Les valeurs possibles sont :

-> **MegaAssets.WornEvents.NONE ou 0** : Bloque l'émission d'événements.

-> **MegaAssets.WornEvents.CHILDREN_ONLY ou 1** : Les événements seront accessible uniquement aux enfants du noeud portant le module en question.

-> **MegaAssets.WornEvents.PARENTS_ONLY ou 2** : Les événements seront accessible uniquement aux pères du noeud portant le module en question.

-> **MegaAssets.WornEvents.ALL ou 3** : Les événements seront accessible aux pères ainsi qu'aux enfants du noeud portant le module en question.

Notez que tout événement écouté sans passé par une connexion au préalable sera victime de la portée des événements. Cette option n'est disponible que sur les modules destructibles.

+ bool **Multiplayer** = false **:** Souhaitez-vous que le module en question prend en charge le système de multijoueur fournit par ce framework ? Notez que pour que cela fonctionne, il fraudra que vous utilisez le module charger de gérer un jeu configuré en mode multijoueur.

+ Array **EventsBindings :** Tableau de dictionnaires gérant les liaisons internes et externes des événements d'un module à une ou plusieurs méthodes et propriétés données. Pour utiliser cette option, le développeur doit suivre le concepte clé/valeur. Cependant, les dictionnaires supportent certains mots clés :

» String **trigger** : Contient le nom de l'événement du module en question. Ce déclencheur représente la condition de déclenchement de la future action qui sera donner par le développeur.

» float **delay** = 0.0 : Contient le temps mort avant le démarrage de l'exécution des actions données par le développeur.

» **Array actions** : Tableau de dictionnaires gérant les différentes configurations liées à une propriété ou à une méthode donnée. Notez que vous avez la possibilité de mettre directement un dictionaire. Les dictionnaires prises en charge par cette clé, supportent les clés suivantes :

- **String | NodePath source = '.'** : Contient l'addresse de la méthode ou de la propriété à ciblée. La présence de cette clé n'est pas obligatoire. Dans ce cas, le module considéra que la méthode ou la propriété référée se trouve au sein de lui-même.

- **String action** : Contient le nom de la méthode ou de la propriété à ciblée. L'utilisation de cette clé est obligatoire.

- **Variant value** : Cette clé, est à utilisée uniquement lorsque l'action à éffectuée est sur une propriété. Elle contient la valeur à affectée à la propriété en question. La présence de cette clé n'est pas obligatoire. Dans ce cas, le module renvoyera la valeur de la propriété spécifiée.

- **Variant params = Array ([])** : Contient les valeurs des différents paramètres de la méthode ciblée. Cette clé, est à utilisée uniquement lorsque l'action à éffectuée est sur une méthode. Le remplissage de ce tableau, doit respecté l'ordre d'alignement des paramètres de la méthode en question. La présence de cette clé n'est pas obligatoire. Dans ce cas, le module considéra que la méthode spécifiée ne possède aucun paramètre(s).

Il est également possible à la place d'une valeur fixe : que cela soit au niveau d'une propriété ou des paramètres d'une méthode, de référer la valeur d'une autre propriété ou d'un résultat renvoyé par une méthode de façon récursive. Pour pouvoir faire cela, il faut utilisé une chaîne de caractères dans laquelle nous auront un caractère spéciale : le `?` placé devant le nom de la propriété ou de la méthode dont la valeur sera utilisée comme valeur à attribuée à la dite propriété ou paramètre de la méthode en question. Lorsque l'on souhaite agir sur une méthode au cours d'un déclenchement, il faut mettre à la fin, les caractères `()` afin de faire la distinction entre une propriété et une méthode.

+ bool Simulate = false : Booléen qui une fois activé, donne un apperçut du fonctionnement du module en question. En d'autres termes, cette option accomplit la tâche principale d'un module en fonction des configurations éffectuées à son niveau. Cette option n'est pas présente dans tous les cas. Cela dépend de la nature et du fonctionnement du module en question. Le champ d'activité de cette propriété est uniquement sur le moteur Godot.

+ bool ResetValues = true : Contrôle la rénitialisation des valeurs des différentes entrées d'un module. Le champ d'activité de cette propriété est uniquement sur le moteur Godot.

V – Les méthodes de base d'un module

Tous les modules de ce framework possèdent des méthodes de bases. Cependant, la présence de certaines méthodes sont spécifiques à la nature et au fonctionnement du module en question.

+ static void set_var (pname, value, object, delay = 0.0) : Utilisée comme modificateur, cette méthode modifie la valeur de n'importe quelle champ d'un module grâce à son nom. L'influence

de cette méthode s'étend également vers d'autres scripts.
» **String pname** : Contient le nom du champ à modifié.
» **Variant value** : Contient la nouvelle valeur du champ.
» **Object object** : Contient l'instance de l'objet possédant la propriété à modifiée.
» **float delay** : Quel est le temps mort avant la mise à jour de la valeur du champ ciblé ? Notez que si vous donnez la référence d'un objet qui n'est pas l'instance d'un noeud, vous serez dispenser d'utiliser ce paramètre.

+ **void set_prop (pname, value, wait = false, delay = 0.0) :** Utilisée comme modificateur, cette méthode modifie la valeur de n'importe quelle champ d'un module grâce à son nom.
Cependant, le champ d'activité de cette méthode est limité au module sur lequel elle couvre. En d'autres termes, vous ne pourrez qu'agir que sur les propriétés définient au sein de la propriété spéciale *__prop__*, contrairement à la méthode *set_var ()* ayant un plus vaste champ d'activité.
» **String pname** : Contient le nom du champ à modifié.
» **Variant value** : Contient la nouvelle valeur du champ.
» **bool wait** : Voulez-vous attendre que l'événement *_enter_tree ()* soit appelé avant l'initialisation et l'exécution des différentes configurations éffectuées au niveau des propriétés du module ? N'activez ce paramètre que dans les méthodes suivantes : *_init, _set, _get, _get_property_list, _notification* ou *_get_configuration_warning*.
» **float delay** : Quel est le temps mort avant la mise à jour de la valeur du champ ciblé ?

+ **static Variant get_var (pname, object, dropdown = false) :** Utilisée comme accèsseur, cette méthode renvoie la valeur de n'importe quelle champ d'un module grâce à son nom. L'influence de cette méthode s'étend également vers d'autres scripts.
» **String pname** : Contient le nom du champ à modifié.
» **Node object** : Contient l'instance de l'objet possédant la propriété à récupérée.
» **bool dropdown** : Voulez-vous prendre en charge les chaînes contenu dans les listes déroulantes ? Dans ce cas, à la place d'un index de position, vous aurez la valeur réelle actuellement sélectionée dans la liste déroulante en question.

+ **Variant get_prop (pname, dropdown = false) :** Utilisée comme accèsseur, cette méthode renvoie la valeur de n'importe quelle champ d'un module grâce à son nom. Cependant, le champ d'activité de cette méthode est limité au module sur lequel elle couvre. En d'autres termes, vous ne pourrez que récupérer la valeur des propriétés définient au sein de la propriété spéciale *__prop__*, contrairement à la méthode *get_var ()* ayant un plus vaste champ d'activité.
» **String pname** : Contient le nom du champ à modifié.
» **bool dropdown** : Voulez-vous prendre en charge les chaînes contenu dans les listes déroulantes ? Dans ce cas, à la place d'un index de position, vous aurez la valeur réelle actuellement sélectionée dans la liste déroulante en question.

+ **void restart (delay = 0.0) :** Redémarre un module. Faites très attention au cours des redémarrages des modules. Cela peut s'avérer problématique dans certains cas.
  » **float delay :** Quel est le temps mort avant le redémarrage du module ?

+ **void simulate (delay = 0.0) :** Donne un apperçut du fonctionnement d'un module. En d'autres termes, elle accomplit la tâche principale d'un module en fonction des configurations éffectuées à son niveau. Cette fonction n'est pas présente dans tous les cas. Cela dépend de la nature et du fonctionnement du module en question.
  » **float delay :** Quel est le temps mort avant la simulation ?

+ **void reset_values (delay = 0.0) :** Rénitialise les valeurs des différentes entrées d'un module.
  » **float delay :** Quel est le temps mort avant la rénitialisation des valeurs ?

+ **static String get_compatibility () :** Renvoie la version de l'éditeur en adéquation avec *Godot Mega Assets*.

+ **static String get_version () :** Renvoie la version d'un module.

+ **static String get_author_name () :** Renvoie le nom de l'auteur du framework *Godot Mega Assets*.

+ **static String get_supported_dimensions () :** Renvoie la dimension en adéquation avec un module.

+ **static String get_supported_platforms () :** Renvoie le(s) nom(s) de(s) platforme(s) en adéquation avec un module.

+ **static String get_used_lisence () :** Renvoie la license qu'utilise *Godot Mega Assets*.

+ **static String get_source () :** Renvoie le lien où l'on peut trouvé le framework *Godot Mega Assets* sur le web.

+ **static bool is_saveable () :** Détermine si l'on peut sauvegarder directement les données d'un module. En d'autres termes, le module en question est-il sauvegardable ?

+ **static String get_category_name () :** Renvoie le nom de la catégorie dont appartient un module.

+ **static String get_origin_name () :** Renvoie les origines d'un module.

+ **bool is_saved () :** Détermine si les données aux sein d'un module ont été sauvegardées dans le gestionnaire des données du jeu. Elle n'est disponible que sur les modules sauvegardables.

+ **void update_data (delay = 0.0) :** Met à jour le gestionnaire des données du jeu. Cette fonction n'est disponible que sur les modules sauvegardables.
  » **float delay** : Quel est le temps mort avant la mise à jour du gestionnaire de données ?

+ **void save_data (delay = 0.0) :** Sauvegarde les données d'un module en utilisant le système de gestion des données du jeu. Notez qu'il est déconseillé d'utiliser cette méthode lorsqu'on veut éffectué plusieurs sauvegardes à la fois. Cette fonction n'est disponible que sur les modules sauvegardables.
  » **float delay** : Quel est le temps mort avant la sauvegarde des données ?

+ **void load_data (delay = 0.0) :** Charge les données d'un module en utilisant le système de gestion des données du jeu. Cette fonction n'est disponible que sur les modules sauvegardables.
  » **float delay** : Quel est le temps mort avant le chargement des données ?

+ **void remove_data (delay = 0.0) :** Détruit tous les données liées à un module du gestionnaire de données. Attention ! il n'y aura pas de retour arrière après la destruction de ces dernières. Cette fonction n'est disponible que sur les modules sauvegardables.
  » **float delay** : Quel est le temps mort avant la suppression des données ?

+ **bool is_dont_destroy_mode () :** Détermine si un module est de base indestructible dans le jeu. En d'autres termes, le module en question a t-il été créé pour rester hors d'atteinte des changements de scènes ?

+ **bool is_unlock () :** Détermine si un module est actif ou pas. Notez que la valeur renvoyée par cette méthode prend également en charge la zone d'activité du module en question.

+ **void set_container (name, id, value, operation = 1, delay = 0.0) :** Effectue les trois opérations (Modifier, Ajouter et Supprimer) uniquement sur les tableaux et dictionaires.
  » **String name** : Quel est le nom du conteneur ciblé ?
  » **Variant id** : Quel identifiant du conteneur voulez-vous atteindre ? Notez que si le conteneur est un tableau, l'identifiant doit être un entier.
  » **Variant value** : Quelle valeur attribuée à l'identifiant du conteneur ciblé ? Notez que l'assignation de la valeur n'est pas typée.
  » **int operation** : Quelle est l'opération à éffectuée sur le conteneur ciblé ? Les valeurs possibles sont :
    -> **MegaAssets.ContainerOperation.NONE ou 0** : Ne rien faire.
    -> **MegaAssets.ContainerOperation.SET ou 1** : Modifier la valeur d'un identifant dans un conteneur.

-> MegaAssets.ContainerOperation.ADD **ou** 2 : Ajouter une nouvelle valeur à un conteneur. Si le conteneur est un tableau et son identifiant n'est pas définit ou invalide, l'élément sera ajouter à la fin de ce dernier. Dans le cas d'un dictionaire, la clé ciblée est créé lorsqu'elle n'existe pas dans le dictionaire ou mise à jour dans le cas contraire.

-> MegaAssets.ContainerOperation.REMOVE **ou** 3 : Supprimer un identifiant d'un conteneur. Si l'identifiant donné est hors des limites du conteneur (tableau ou dictionaire), on assistera à un nétoyage complet de ce dernier.

» **float delay** : Quel est le temps mort avant la mise à jour du conteneur ciblé ?

+ **void set_containers (configs, delay = 0.0) :** Effectue les trois opérations (Modifier, Ajouter et Supprimer) uniquement sur les tableaux et les dictionaires.

» **Dictionary | Array configs** : Contient les différentes configurations sur la manière dont chaque conteneur du module est géré. Le(s) dictionaire(s) de cette méthode supportent les clés suivantes :

- **String name** : Quel est le nom du conteneur ciblé ?

- **Variant id** : Quel identifiant du conteneur voulez- vous atteindre ? Notez que si le conteneur est une liste, l'identifiant doit être un entier.

- **Variant value** : Quelle valeur attribuée à l'identifiant du conteneur ciblé ? Notez que l'assignation de la valeur n'est pas typée.

- **float timeout = 0.0** : Devons-nous patientez sur le délay donné avant d'exécuter les configurations données ?

- **int operation = 0.0** : Quelle est l'opération à éffectuée sur le conteneur ciblé ? Les valeurs possibles sont :
  -> MegaAssets.ContainerOperation.NONE **ou** 0 : Ne rien faire.
  -> MegaAssets.ContainerOperation.SET **ou** 1 : Modifier la valeur d'un identifant dans un conteneur.
  -> MegaAssets.ContainerOperation.ADD **ou** 2 : Ajouter une nouvelle valeur à un conteneur. Si le conteneur est un tableau et son identifiant n'est pas définit ou invalide, l'élément sera ajouter à la fin de ce dernier. Dans le cas d'un dictionaire, la clé ciblée est créé lorsqu'elle n'existe pas dans le dictionaire ou mise à jour dans le cas contraire.
  -> MegaAssets.ContainerOperation.REMOVE **ou** 3 : Supprimer un identifiant d'un conteneur. Si l'identifiant donné est hors des limites du conteneur (tableau ou dictionaire), on assistera à un nétoyage complet de ce dernier.

» **float delay** : Quel est le temps mort avant la mise à jour du conteneur ciblé ?

VI – Les événements de base d'un module

Tous les modules du framework possèdent des événements de bases. Cependant, la présence de certains événements sont spécifiques à la nature et au fonctionnement du module en question.
Les événements, au cours de leur déclenchement peuvent soit renvoyé quelque chose, soit rien. Dans la mesure où ils retournent plusieurs donnée(s), celle(s)-ci sont misent dans un dictionnaire appelé *data*. Les données renvoyées sont spécifiques à l'événement en question.

NB : Si le module ne renvoie qu'une seule donnée, aucun dictionnaire n'est solicité pour envoyer cette dernière. *Cela signifie que le développeur ne doit pas utilisé un dictionnaire et récupéré directement la valeur renvoyée lorsque l'événement en question ne renvoie qu'une seule donnée à son déclenchement*. Petite précision, Les événements écoutés directement sans passé par une connexion au préalable, doivent être utilisés en mettant devant leur nom le préfix : *_on_* suivit du nom de l'événement en question. C'est très important de comprendre cela lorsqu'on veut écouté un événement de façon directe. **Avant de continuer, notez que la référence du noeud n'est pas envoyée lorsque le module est de nature indestructible.**

Code : GDScript

```
# Call on any module values changed.
func _on_values_changed (data):
    # TODO something here...
    pass;
```

+ **values_changed (data) :** Signal déclenché lorsque n'importe quel champ d'un module change de valeur. Cet événement renvoie un dictionaire contenant les clés suivantes :
  » **Node node** : Contient le noeud où cet signal a été émit.
  » **String name** : Contient le nom de la propriété ayant changé de valeur.
  » **Variant value** : Contient la nouvelle valeur de la propriété en question.

+ **enabled (node) :** Signal déclenché lorsqu'on active un module. Notez que cet événement s'appel même si le module est désactivé.
  » **Node node** : Contient le noeud où cet signal a été émit.

+ **disabled (node) :** Signal déclenché lorsqu'on désactive un module. Notez que cet événement s'appel même si le module est désactivé.
  » **Node node** : Contient le noeud où cet signal a été émit.

+ **start (node) :** Signal déclenché après l'initialisation d'un module.
  » **Node node** : Contient le noeud où cet signal a été émit.

+ **children_changed (node) :** Signal déclenché lorsqu'on supprime ou qu'on ajoute un ou plusieurs enfants à un module.
  » **Node node :** Contient le noeud où cet signal a été émit.

+ **parent_changed (node) :** Signal déclenché lorsqu'on change de parent à un module.
  » **Node node :** Contient le noeud où cet signal a été émit.

+ **before_update_data (node) :** Signal déclenché avant la mise à jour du gestionnaire des différentes données du jeu. Cet événement n'est disponible que sur les modules sauvegardables.
  » **Node node :** Contient le noeud où cet signal a été émit.

+ **after_update_data (node) :** Signal déclenché après la mise à jour du gestionnaire des différentes données du jeu. Cet événement n'est disponible que sur les modules sauvegardables.
  » **Node node :** Contient le noeud où cet signal a été émit.

+ **before_save_data (node) :** Signal déclenché avant la sauvegarde des données d'un module. Cet événement n'est disponible que sur les modules sauvegardables.
  » **Node node :** Contient le noeud où cet signal a été émit.

+ **after_save_data (node) :** Signal déclenché après la sauvegarde des données d'un module. Cet événement n'est disponible que sur les modules sauvegardables.
  » **Node node :** Contient le noeud où cet signal a été émit.

+ **before_load_data (node) :** Signal déclenché avant le chargement des données d'un module. Cet événement n'est disponible que sur les modules sauvegardables.
  » **Node node :** Contient le noeud où cet signal a été émit.

+ **after_load_data (node) :** Signal déclenché après le chargement des données d'un module. Cet événement n'est disponible que sur les modules sauvegardables.
  » **Node node :** Contient le noeud où cet signal a été émit.

+ **before_destroy_data (node) :** Signal déclenché avant la destruction des données d'un module. Cet événement n'est disponible que sur les modules sauvegardables.
  » **Node node :** Contient le noeud où cet signal a été émit.

+ **after_destroy_data (node) :** Signal déclenché après la destruction des données d'un module. Cet événement n'est disponible que sur les modules sauvegardables.
  » **Node node :** Contient le noeud où cet signal a été émit.

Tout ce que nous venons d'évoquer peut être mis dans un schéma pour avoir une vue d'ensemble sur toutes les différentes fonctionnalités de bases communes à tous les modules du framework.

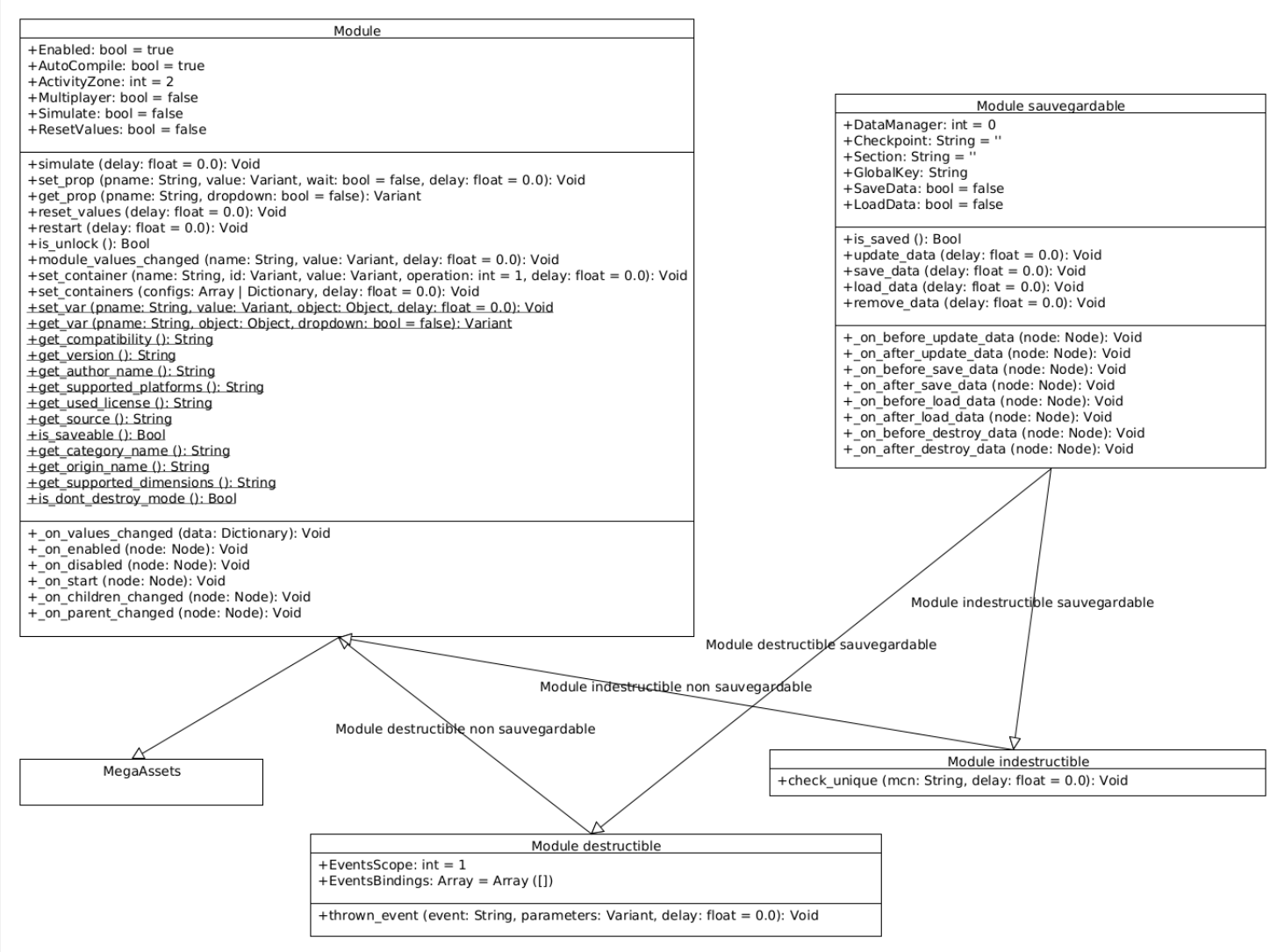

FIGURE 1 – Diagramme récapitulatif des fonctionnalités de base

VII – Structure du code d'un module

Tous les modules possèdent la même structure de code à savoirs :

+ Tout en haut avant tout code se trouve des informations à propos du module en question. Bien évidement, certaines de ces informations sont récupérable via du code à travers certaines fonctions offertes par le module.
+ Les attributs ou variables de classe.
+ Les signals ou événements.
+ Les variables particulières.
+ La gestion des entrées.
+ La logique et tâches principaux.
+ Les fonctionnalités disponibles.

» **Les attributs ou variables de classe :** Partie comportant toutes les données néccessaires à la bonne marche du module. Ces variables ne sont pas privées. Ainsi, le développeur a donc la possibilité de pouvoir y accédées.

» **Les signals ou événements :** Partie regroupant tous les événements dont dispose le module.

» **Les variables particulières :** Partie comportant des données clées du module. Ces variables sont privées donc inaccessible aux développeurs.

» **La gestion des entrées :** Partie regroupant toutes les fonctions de contrôle des valeurs des champs du module. Cette partie joue un rôle important dans le fonctionnement du module. C'est elle qui est responsable des exigences du module. Son objectif est d'aider le développeur à bien configuré le module afin d'éviter les erreurs inutiles. Les méthodes de cette partie sont toutes privées.

» **Logique et tâches principaux :** Cette partie représente le pilier centrale du module. Elle traite les différentes données issues des configurations du *GameMaster* afin de coordonner le comportement et le fonctionnement du module. C'est le coeur du module et toutes les méthodes de cette partie sont privées.

» **Les fonctionnalités disponibles :** Cette partie comporte toutes les méthodes offerte par le module pour son utilisation. Toutes les fonctions de cette partie sont publiques.

VIII – Création d'un module

Cette section est uniquement dédiée aux développeurs qui souhaite créé leur propre module en utilisant *Godot Mega Assets*. Comme vous pouvez le constater, les modules de ce framework suivent tous, une structure bien définie, nous permettant ainsi d'énumérer les types de base suivants : Destructible, Indestructible, Saveable, Recordable et Module.

» **Module :** C'est la classe de base de tous les modules. Elle hérite directement de la classe **MegaAssets**.

» **Destructible :** C'est la classe de base de tous les modules de nature destructible. Elle hérite de la classe Module.

» **Indestructible :** C'est la classe de base de tous les modules de nature indestructible. Elle hérite de la classe Module.

» **Saveable :** C'est la classe de base de tous les modules destructibles et sauvegardables. Elle hérite de la classe Destructible.

» **Recordable :** C'est la classe de base de tous les modules indestuctibles et sauvegardables. Elle hérite de la classe Indestructible.

Tout ce que nous venons d'évoquer peut être mis dans un schéma pour avoir une vue d'ensemble sur les différentes relations entre les classes de base.

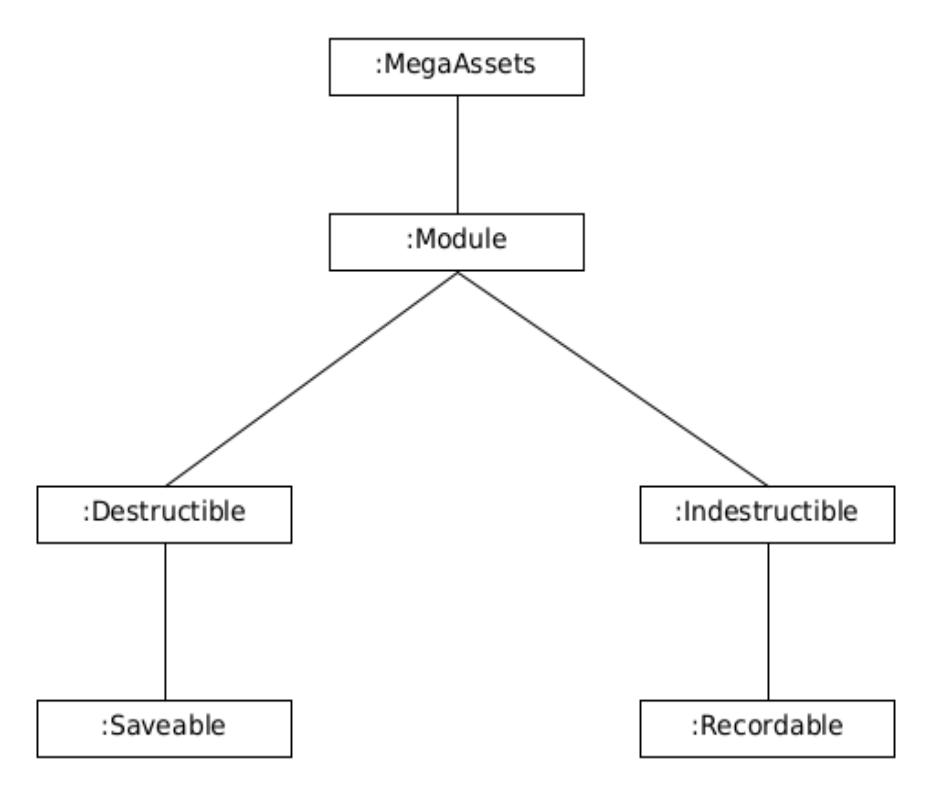

FIGURE 2 – Diagramme récapitulatif des relations entre les classes de base

Choisissez celle que vous voulez en fonction du comportement et du fonctionnement de votre future module. En ce qui concerne la gestion des différentes propriétés au sein du module, des méthodes ont été prévues pour cela.

+ **void thrown_event (event, parameters, delay = 0.0) :** Lève un événement donné. Elle prend également en charge la portée de l'événement ainsi que sa liaison externe vers d'autres exécutions. Notez qu'elle n'utilise pas la fonction *emit_signal ()* dans l'émission d'événement et n'est définie que dans la classe Module.
  » **String event** : Quel est le nom de l'événement à invoqué ?
  » **Variant parameters** : L'événement en question possède t-il des paramètres à renvoyés à ses écouteurs ?
  » **float delay** : Quel est le temps mort avant le déclenchement de l'événement en question ?

+ **void module_values_changed (name, value, delay = 0.0) :** Lève l'événement nommé *values_changed* avec des informations permettant au écouteurs de s'informer du changement de valeur d'une propriété au sein d'un module. Notez qu'elle se sert de la fonction *thrown_event ()* pour accomplir son objectif et n'est définie que dans la classe Module.
  » **String name** : Quel est le nom de la propriété dont la valeur à subit de(s) changement(s) ?
  » **Variant value** : Quelle est la nouvelle valeur de la propriété référée ?
  » **float delay** : Quel est le temps mort avant le déclenchement de l'événement en question ?

+ **void check_unique (mcn, delay = 0.0) :** Le type du module de nature indestructible a t-il plusieurs instances de lui même dans l'arbre de la scène définit ? Si c'est le cas, une erreur se lèvera pour notifier à l'utilisateur de l'unicité des modules indestructibles. Notez que cette méthode n'est définie que dans la classe Indestructible.
  » **String mcn** : Contient le nom de la classe du module dont l'instance ne peut être suppérieur à 1.
  » **float delay** : Quel est le temps mort avant la vérification de l'unicité du module en question ?

+ **void bind_prop (...)** : Ajoute une propriété dans la liste des propriétés d'un script. N'appelez cette méthode que dans la fonction virtuelle : *_init ()*. Pour avoir plus d'informations à ce sujet, consulter la documentation sur la classe **MegaAssets**.

+ **void listen_notifications (...)** : Ecoute les notifications de l'éditeur pour pouvoir ensuite déclenché les configurations éffectuées à propos de la clé *notification* sur les variables d'un script. Pour avoir plus d'informations à ce sujet, consulter la documentation sur la classe **MegaAssets**.

+ **void reset_props_value (...)** : Rénitialise la valeur d'un ou de plusieurs propriétés d'un script. Pour avoir plus d'informations à ce sujet, consulter la documentation sur la classe **MegaAssets**.

+ **Array get_properties (...)** : Renvoie la liste de toutes les propriétés définient au sein du dictionaire *__props__*. Pour avoir plus d'informations à ce sujet, consulter la documentation sur la classe **MegaAssets**.

+ **void destroy_props (...)** : Détruit un ou plusieurs propriété(s) associée(s) à un script. Pour avoir plus d'informations à ce sujet, consulter la documentation sur la classe **MegaAssets**.

+ **void override_prop (...)** : Redéfinit une propriété donnée. Pour avoir plus d'informations à ce sujet, consulter la documentation sur la classe **MegaAssets**.

IX – Template : Création d'un module destructible

Code : GDScript

```
# Dependencies.
tool class_name YourModuleType extends Destructible;

# Contains all module properties.
func _create_module_properties () -> void:
    # Declare all module variables here with "bind_prop ()" method.
    pass;

# This method is called on game initialization and before all nodes instanciation.
func _init (): self._create_module_properties ();

# What is the type of your module ?
func get_class () -> String: return "YourModuleType";

# What is the current version of your module ?
static func get_version () -> String: return "1.0.0";

# What is the supported dimensions of your module ? (Optional).
static func get_supported_dimensions () -> String: return "2D || 3D";

# What is the category of your module ? (Optional).
static func get_category_name () -> String: return "YourModuleCategoryName";

# What is the supported platforms of your module ? (Optional).
static func get_supported_platforms () -> String: return "ANDROID || MACOSX ||
    WINDOWS || LINUX";

# Do you want to allow on others users to restart your module ? (Optional).
func restart (_delay: float = 0.0) -> void: if self.is_unlock (): pass;

# Do you want to give an overview of the main operation of your module ? Called when you
    pressed on "Simulate" module property. (Optional if you destroy "Simulate" property with
    .destroy_props () method).
func simulate (_delay: float = 0.0) -> void: if self.is_unlock (): pass;
```

X – Template : Création d'un module indestructible

Code : GDScript

```
# Dependencies.
tool class_name YourModuleType extends Indestructible;

# Contains all module properties.
func _create_module_properties () -> void:
    # Declare all module variables here with "bind_prop ()" method.
    pass;

# This method is called on game initialization and before all nodes instanciation.
func _init (): self._create_module_properties ();

# Called before ready method run.
func _enter_tree (): self.check_unique (self.get_class ());

# What is the type of your module ?
func get_class () -> String: return "YourModuleType";

# What is the current version of your module ?
static func get_version () -> String: return "1.0.0";

# What is the supported dimensions of your module ? (Optional).
static func get_supported_dimensions () -> String: return "2D || 3D";

# What is the category of your module ? (Optional).
static func get_category_name () -> String: return "YourModuleCategoryName";

# What is the supported platforms of your module ? (Optional).
static func get_supported_platforms () -> String: return "ANDROID || MACOSX ||
    WINDOWS || LINUX";

# Do you want to allow on others users to restart your module ? (Optional).
func restart (_delay: float = 0.0) -> void: if self.is_unlock (): pass;

# Do you want to give an overview of the main operation of your module ? Called when you
    pressed on "Simulate" module property. (Optional if you destroy "Simulate" property with
    .destroy_props () method).
func simulate (_delay: float = 0.0) -> void: if self.is_unlock (): pass;
```