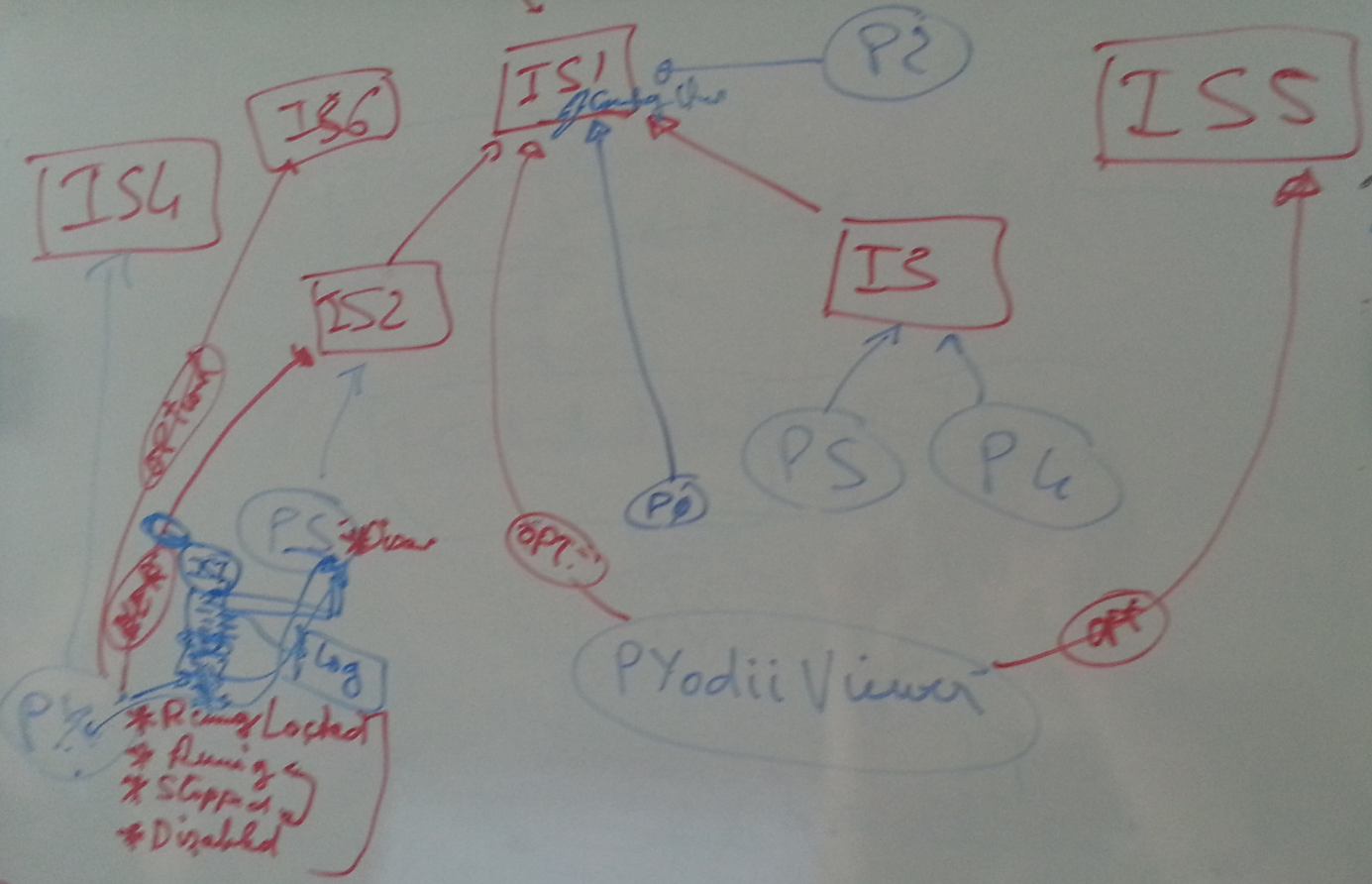
# Yodii



IS = IService

P = Plugin

Un service représente toujours une interface.

Un plugin implémente 0 ou 1 service.

Ainsi comme on peut le voir sur le graph ci-dessus un plugin ne peut implémenter qu’un service au maximum. Mais un service peut spécialiser un autre service.

Exemple :

public class IKeyboardHandler : IYodiiService { … }

public class IKeyboardHandlerAdvance : IKeyboardHandler { … }

On remarque aussi dans l’exemple que tous les services implémentent « IYodiiService ». Cette interface va principalement nous servir de tag pour reconnaître un service.

Un service possède aussi un « RunningStatus » possédant les états :

* RunningLocked
* Running
* Stopped
* Disabled

Ainsi un plugin implémentant le service concerné possède lui aussi un « RunningStatus » qui est logiquement le même que celui du service.

Les états **Running** et **Stopped** n’imposent aucune contrainte par rapport au changement de cet état : le système assurera que l’on pourra switch de l’état **Running** à **Stopped** et inversement sans vérification. De même, on peut switch de ces deux états vers l’état **Disabled** ou **RunningLocked** sans vérification, par contre l’inverse n’est pas possible.

Les états **Disabled** et **RunningLocked** ne sont pas liés au plugin en lui-même, mais à un élément extérieur qui force ces états.

Un plugin dépendant d’un autre plugin va changer l’état **Running** du plugin en **RunningLocked**.

Si une contrainte définie par le développeur ne peut être honorée, un message sera émis pour expliquer les raisons à cela.

Exemple :

//IPlugin.RunningStatus == Stopped

IPlugin.Start(); //IPlugin.RunningStatus == Running

IPlugin.Stop(); //IPlugin.RunningStatus == Stopped

IPlugin.RunningStatus = RunningStatus.Disabled;

IPlugin.Start(); //impossible car le plugin est explicitement disabled

IPlugin.RunningStatus = RunningStatus.RunningLocked;

IPlugin.Stop(); //impossible car le plugin est explicitement RunningLocked

Le plugin implémente l’interface IPlugin qui possède les méthodes :

* Setup()
* Start()
* Stop()
* Teardown()

Ainsi, les différents plugins peuvent être gérés dynamiquement au cours de l’exécution d’un programme.

/// <summary>

/// This interface defines the minimal properties and behavior of a plugin.

/// </summary>

public interface IPlugin

{

/// <summary>

/// This method initializes the plugin: own resources must be acquired and running conditions should be tested.

/// No interaction with other plugins must occur (interactions must be in <see cref="Start"/>).

/// </summary>

/// <param name="info">Enables the implementation to give detailed information in case of error.</param>

/// <returns>True on success. When returning false, <see cref="IPluginSetupInfo"/> should be used to return detailed explanations.</returns>

bool Setup( IPluginSetupInfo info );

/// <summary>

/// This method must start the plugin: it is called only if <see cref="Setup"/> returned true.

/// Implementations can interact with other components (such as subscribing to their events).

/// </summary>

void Start();

/// <summary>

/// This method uninitializes the plugin (it is called after <see cref="Stop"/>).

/// Implementations MUST NOT interact with any other external components: only internal resources should be freed.

/// </summary>

void Teardown();

/// <summary>

/// This method is called by the host when the plugin must not be running anymore.

/// Implementations can interact with other components (such as unsubscribing to their events).

/// <see cref="Teardown"/> will be called to finalize the stop.

/// </summary>

void Stop();

}

Remarque : Deux plugins implémentant un même service ne peuvent être existant simultanément à un même instant T.

Si un plugin dépend d’un autre service il le représentera de la manière suivante :

[requires( RunningRequirement.Optional )]

IService<T> Service { get; set; }

L’attribut décrit ce dont a besoin le plugin pour qu’il puisse fonctionner.

Le « requires » prend en paramètre un Enum :

/// <summary>

/// Describes how a service or a plugin is required.

/// A requirement is a gradation between <see cref="Optional"/> and <see cref="MustExistAndRun"/>.

/// </summary>

[Flags]

public enum RunningRequirement

{

/// <summary>

/// The service or plugin is optional: it can be unavailable.

/// </summary>

Optional = 0,

/// <summary>

/// If it is available the service or plugin should be started.

/// </summary>

OptionalTryStart = 1,

/// <summary>

/// The service or plugin must be available (but it can be stopped).

/// </summary>

MustExist = 2,

/// <summary>

/// The service or plugin must be available and, if possible, should be started.

/// </summary>

MustExistTryStart = 2+1,

/// <summary>

/// The service or plugin must be available and must run.

/// </summary>

MustExistAndRun = 2+4

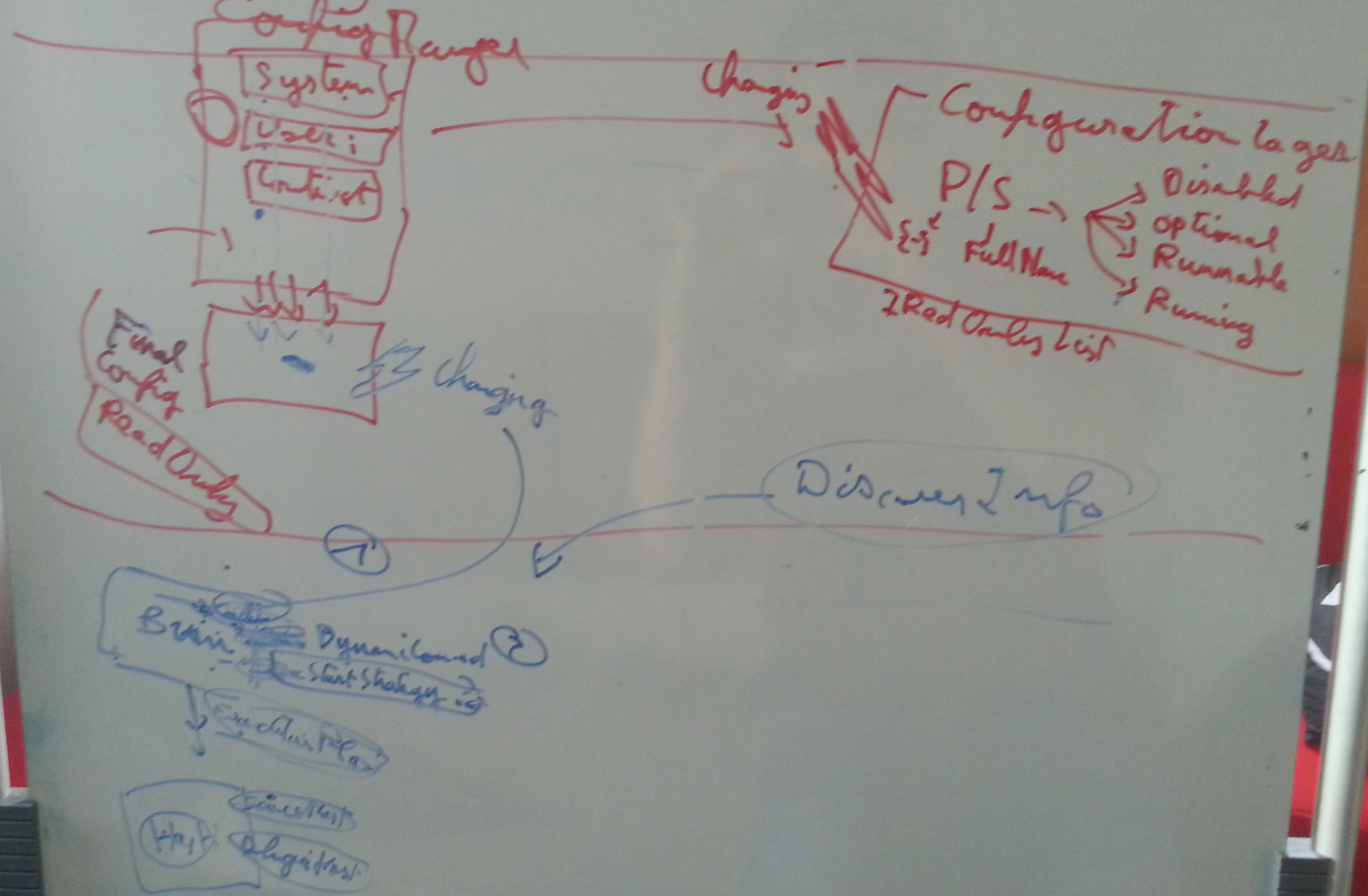
}

« MustExist » deviendra par la suite « Runnable », et représente un service qui est disponible et peut être démarré selon besoin.

Cela représente la dépendance d’un plugin par rapport à au service.

Un « IService<T> » est en réalité un proxy qui ré implémente dynamiquement le service pour fournir seulement les méthodes nécessaires. Ainsi, on peut aisément rajouter des fonctions supplémentaires non prévues par le service, comme un IActivityLogger spécifique.

## ConfigurationManager



Le système est semblable au système de services Windows à quelques points près.

Le **ConfigurationManager** possèdera un « bag » de **ConfigurationLayer** qui lui sera fournis au fur et à mesure de l’exécution du programme.

Un **ConfigurationLayer** représente un ensemble de règles pour un ou des services ou plugin.

Cet ensemble sera regroupé dans un **FinalConfigurationLayer** après validation de la possibilité d’ajouter chaque nouvelle règle.

Le « bag » devra obligatoirement contenir un **ConfigurationLayer système** qui constituera une base.

Ces règles applicables sont représentées de telle manière :

* **Disable** : le service ne peut être lancé, au runtime le **RunningStatus ==** **Disabled**
* **Optional** : le service peut être dans n’importe quel état et même de ne pas exister au runtime
* **Runnable** : le service doit exister au runtime et le **RunningStatus != Disabled**
* **Running** : le service doit exister au runtime et le **RunningStatus == RunningLocked**

La règle **Optional** peut être modifiée au cours de l’exécution, vers n’importe quelle autre règle. **Runnable** peut être modifiée que vers Running.

**Disable** et **Running** ne peuvent pas être modifiées une fois appliqué.

Par défaut, chaque plugin et service non présent dans le **FinalConfigurationLayer** est considéré avec la règle **Optional**.

Un service est identifié par son FullName (Namespace+Class/Interface name) et un plugin par un GUID.

Brouillon :

public interface IConfigurationManager

{

bool Add( IConfigurationLayer configurationLayer );

void Remove( IConfigurationLayer configurationLayer );

ConfigStatus( GUID plugin, ConfigStatus status );

ConfigStatus( String ServiceFullName, ConfigStatus status );

}

Le **ConfigurationManager** n’a pas de taille limite.

Grâce aux fonctions **SetConfigStatus** et **Add** des règles pourront être ajoutées à la volée ; par contre, à chaque modification, le système devra faire valider la demande en lançant un Event (ConfigurationChanging). En changeant une propriété de l’EventArgs relié à l’event, un nouveau layer de configuration ou une modification ont être refusés, si ceux-ci entre en conflit avec les règles déjà établies.

## DynamicCommand

Il s’agit d’un système d’historique des demandes de démarrage et d’extinction de plugin par rapport à une action utilisateur.

L’ensemble des commands n’a pas de taille limite ; il est limité implicitement par le nombre de plugins et de services présents.

Pour assurer ce fonctionnement, un nettoyage est effectué lorsqu’une command devient obsolète.

## PlanCalculatorStrategy

C’est un moyen de définir une stratégie d’exécution (tout Running ou tout Stopped) qui lorsqu’un plugin est présent mais représente une « dépendance » optionnelle pour un autre plugin.

## Final

Après l’ensemble du traitement, lors du démarrage ou lors d’une modification :

Brain throws => PlanCalculationResult

Engine gets <= PlanCalculationResult