

Domain-Driven Design Technique :

Les Nouveautés de C# (Static Abstract
Members, Implicit Operators) et le
Pattern Factory



Lien du Meeting
<https://bit.ly/4092ksH>



Samedi 11 Janvier 2025
à 10H00 GMT



Daniel WOAGOU



CODE CRAFT CHRONICLES

2025

PARTIE 1 : **DOMAIN-DRIVEN DESIGN &** **CONCEPTS CLÉS**



@DANIEL WOAGOU



1. INTRODUCTION

Définition

LE DDD EST UNE APPROCHE DE CONCEPTION LOGICIELLE QUI PLACE LE DOMAINE MÉTIER AU CŒUR DU DÉVELOPPEMENT, EN COLLABORATION ÉTROITE AVEC LES EXPERTS MÉTIER.



@DANIEL WOAGOU



1. INTRODUCTION

Pourquoi ?

- GÉRER LA COMPLEXITÉ MÉTIER.
- ASSURER UNE COHÉRENCE ENTRE LE CODE ET LES BESOINS MÉTIER.
- RENDRE LE CODE PLUS LISIBLE ET MAINTENABLE.



1. INTRODUCTION

Conditions?

- PROJETS AVEC DES RÈGLES MÉTIER COMPLEXES.
- SYSTÈMES NÉCESSITANT DES MODÈLES ÉVOLUTIFS
- UNE FORTE COLLABORATION MÉTIER-TECHNIQUE.



@DANIEL WOAGOU



- VALUE OBJECTS

Un Value Object est un élément fondamental du Domain-Driven Design (DDD) qui représente un concept ou une valeur métier définie uniquement par son contenu (ses valeurs) et non par une identité unique. Contrairement à une entité, un Value Object ne possède pas d'identifiant distinct et ne change pas d'état une fois créé.

Les Value Objects sont utilisés pour modéliser des concepts métiers immuables, encapsulant des règles de validation et des comportements spécifiques liés à ces valeurs. Ils permettent d'exprimer de manière claire et précise les concepts métier complexes tout en rendant le code plus robuste et maintenable.



• VALUE OBJECTS

- Pas d'identité propre: deux Value Objects sont considérés comme égaux si leurs valeurs sont identiques, peu importe où ils sont instanciés.
- Immutabilité (sécurité) : ils ne changent pas après leur création. Toute modification implique la création d'un nouvel objet.
- Comparaison par valeur: doivent être comparés uniquement sur la base de leurs contenus
- Encapsulation métier : règles métier et des comportements liés à leurs valeurs
- Validation lors de la création
- Focus sur la simplicité et la réutilisation





@DANIEL WOAGOU



• ENTITIES

Une Entity est un objet du domaine qui est identifié de manière unique par un attribut, souvent appelé ID. Ce qui caractérise une entité, c'est que son identité persiste indépendamment des changements apportés à ses autres propriétés.

Caractéristiques des Entities :

1. Identité unique : Chaque entité possède une identité unique qui permet de la distinguer des autres. L'identité ne change jamais, même si d'autres propriétés de l'entité changent.
2. Cycle de vie : Les entités ont un cycle de vie qui inclut leur création, leur mise à jour et éventuellement leur suppression.
 - Le suivi de leur état dans le temps est important.
3. Égalité basée sur l'identité : deux entités sont considérées comme égales si elles partagent la même identité, et non pas parce que leurs propriétés sont identiques.



1. CONCEPTS CLÉS DU DDD

• ENTITIES



@DANIEL WOAGOU



1. CONCEPTS CLÉS DU DDD

• ENTITIES



@DANIEL WOAGOU



• AGGREGATES

Un agrégat DDD est un groupe d'objets de domaine fortement liés par des règles métiers et qui peuvent être traités comme une seule unité.

- L'aggregate garantit l'intégrité des invariants métiers : ces règles qui doivent toujours être respectées.
- Il possède une root entity ou aggregate root, qui agit comme point d'accès principal pour interagir avec l'aggregate.

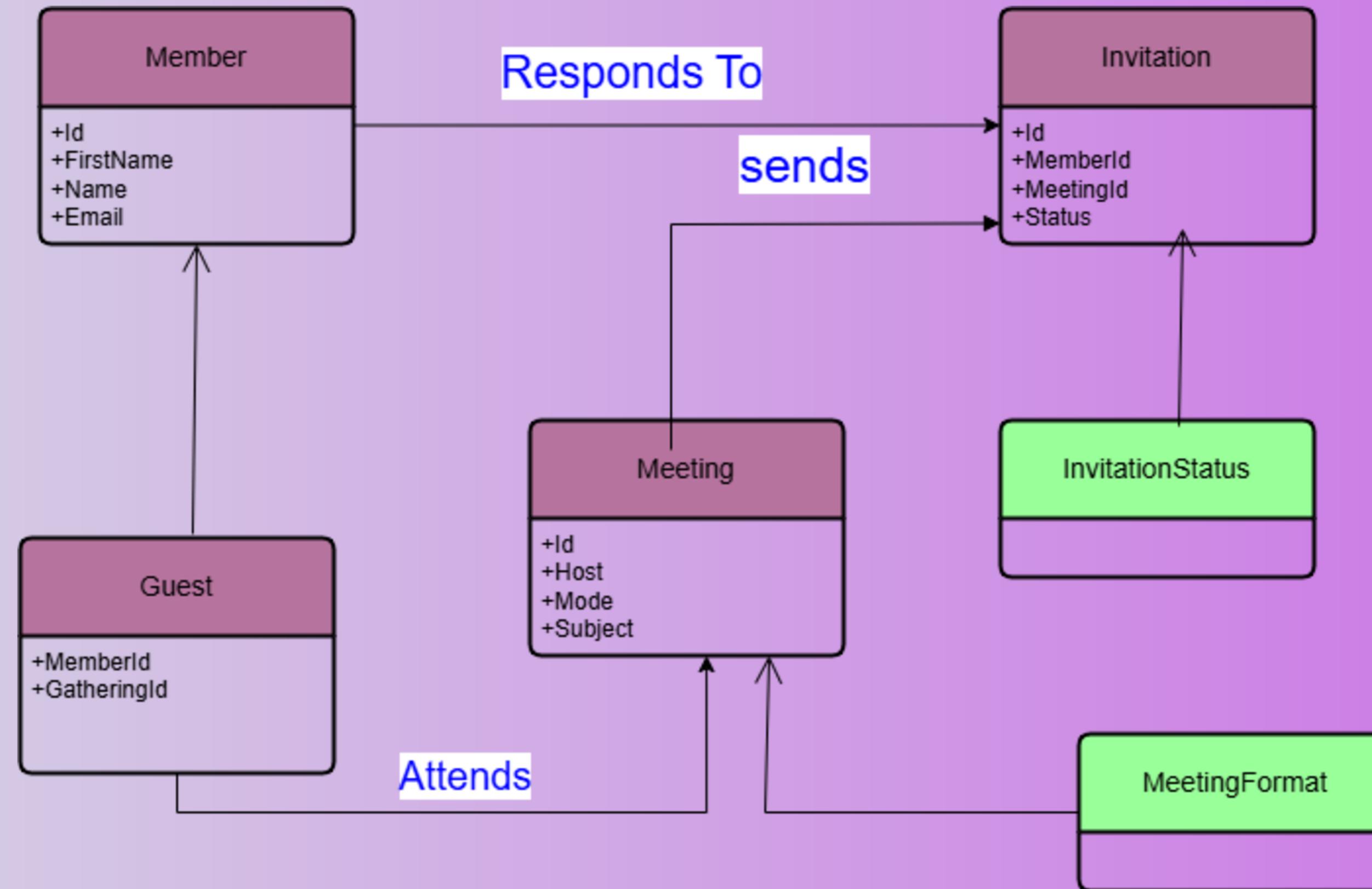


1. CONCEPTS CLÉS DU DDD

• AGGREGATES: LES CONCEPTS

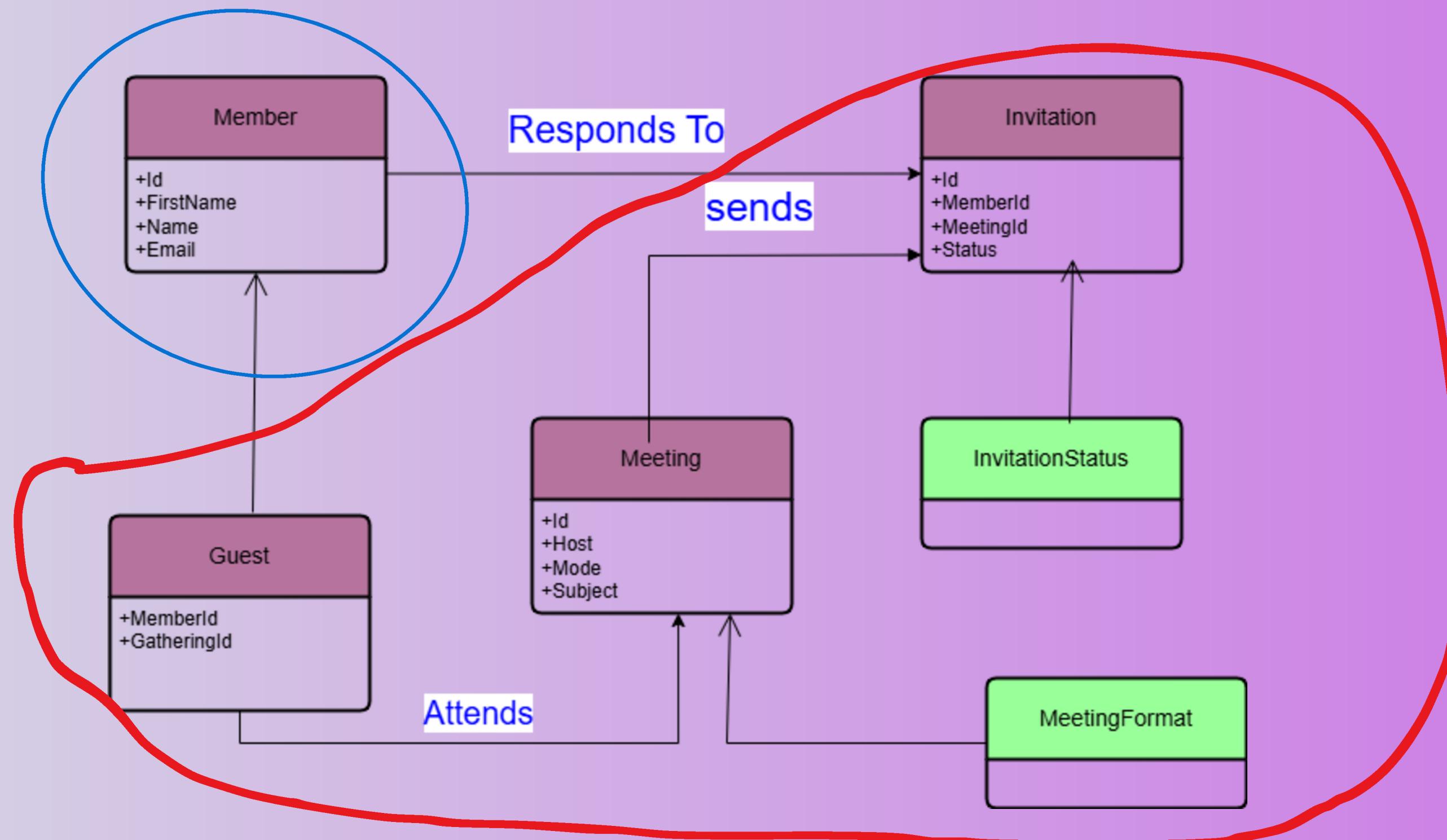
- Aggregate Root :
 - L'élément principal de l'aggregate.
 - C'est à travers cette racine que toutes les interactions avec l'aggregate se font.
 - Elle est la seule entité exposée en dehors de l'aggregate.
- Bounded Context : Un aggregate appartient à un bounded context, qui définit les limites d'un domaine métier.
- Encapsulation : Tout ce qui est dans un aggregate est protégé. Les modifications doivent se faire via les méthodes exposées par l'aggregate root.
- Invariant : Les invariants sont des règles métiers strictes qui doivent être respectées en permanence à l'intérieur de l'aggregate.





@DANIEL WOAGOU





@DANIEL WOAGOU



1. CONCEPTS CLÉS DU DDD

• SERVICES, FACTORIES, REPOS

- Domain Services : Des objets métier contenant des logiques complexes qui ne peuvent pas être correctement associées à une seule Entity ou Value Object.
Exemple : Calcul de disponibilité de place pour un meeting ou validation d'un planning.
- Factories : Utilisées pour créer des Entities ou Aggregates complexes.
 - Encapsulent les règles métier liées à leur création.
- Repositories :
 - Fournissent une interface d'accès au stockage pour les Aggregates.
- Exemple : Trouver toutes les meetings organisé pour une date donnée ou par un membre.



@DANIEL WOAGOU



PARTIE 2 :
TRANSITION VERS LES
NOUVEAUTÉS C# DANS LE
DDD



@DANIEL WOAGOU



2. LES NOUVEAUTÉS C#

- STATIC ABSTRACT MEMBERS DANS LES INTERFACES (C# 11)

- Exemple : Capacité (Capacity) avec Static Abstract Members

Imaginons que tu as un VO représentant une Capacité, comme dans un contexte de gestion de ressources ou de stockage.

- Avant C# 11 :

Tu devais définir des méthodes statiques et les réimplémenter dans chaque classe. Cela augmentait la duplication de code.

- Après C# 11 avec Static Abstract Members :

Tu peux définir une méthode statique abstraite dans une interface et l'implémenter directement dans chaque VO, tout en profitant de l'auto-résolution des types.



2. LES NOUVEAUTÉS C#

- STATIC ABSTRACT MEMBERS DANS LES INTERFACES (C# 11)



```
// Définition de l'interface avec une méthode statique abstraite
public interface IEntity<TEntity, TId, TDto>
{
    TId Id { get; set; }

    static abstract TEntity Create(TDto dto); // Méthode statique abstraite
}
```



@DANIEL WOAGOU



2. LES NOUVEAUTÉS C#

• STATIC ABSTRACT MEMBERS DANS LES INTERFACES (C# 11)



```
// Définition de l'interface avec une méthode statique abstraite
// Le DTO pour transférer les données
public record UserData(string Name, int Age);

// La classe User avec un constructeur privé
public class User : IEntity<User, int, UserData>
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }

    // Constructeur privé pour restreindre la création directe de l'instance
    private User() { }

    // Implémentation de la méthode statique pour créer un User
    public static User Create(UserData dto)
    {
        // Utilisation du constructeur privé pour créer une instance de User
        return new User { Name = dto.Name, Age = dto.Age };
    }
}
```



2. LES NOUVEAUTÉS C#

- STATIC ABSTRACT MEMBERS DANS LES INTERFACES (C# 11)



```
Console.WriteLine("Hello, World!");  
  
// Cr ation d'un User en utilisant le DTO UserData  
UserData userData = new UserData("John", 30);  
  
// Cr ation de l'entit  User via la m thode statique Create  
User user = User.Create(userData);
```



@DANIEL WOAGOU



2. LES NOUVEAUTÉS C#

- OPÉRATEURS IMPLICITES

Les opérateurs implicites en C# permettent de définir des conversions automatiques entre types, sans avoir besoin de faire appel explicitement à une méthode de conversion ou un constructeur. Une conversion implicite se produit automatiquement lorsqu'un type peut être converti en un autre type de manière sûre (sans perte d'information). Cela rend le code plus propre et plus facile à lire, car il n'est pas nécessaire d'utiliser des appels explicites pour ces conversions.



```
public static implicit operator TypeC(TypeA value)
{
    // Code de conversion
}
```



2. LES NOUVEAUTÉS C#

- OPÉRATEURS IMPLICITES

Les opérateurs implicites en C# permettent de définir des conversions automatiques entre types, sans avoir besoin de faire appel explicitement à une méthode de conversion ou un constructeur. Une conversion implicite se produit automatiquement lorsqu'un type peut être converti en un autre type de manière sûre (sans perte d'information). Cela rend le code plus propre et plus facile à lire, car il n'est pas nécessaire d'utiliser des appels explicites pour ces conversions.



```
public static implicit operator TypeC(TypeA value)
{
    // Code de conversion
}
```



2. LES NOUVEAUTÉS C#

- OPÉRATEURS IMPLICITES



```
public class User : IEntity<User, int, UserData>
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }

    // Implémentation de la méthode statique pour créer un User
    public static User Create(UserData dto) => new User { Name = dto.Name, Age = dto.Age };

    // Définir un opérateur implicite pour convertir UserData en User
    public static implicit operator User(UserData dto) => new User { Name = dto.Name, Age = dto.Age };

    // Définir un opérateur implicite pour convertir User en UserData (si nécessaire)
    public static implicit operator UserData(User user) => new UserData(user.Name, user.Age);
}
```



2. LES NOUVEAUTÉS C#

- AGGREGATES: LES CONCEPTS

- Aggregate Root :
 - L'élément principal de l'aggregate.
 - C'est à travers cette racine que toutes les interactions avec l'aggregate se font.
 - Elle est la seule entité exposée en dehors de l'aggregate.
- Bounded Context : Un aggregate appartient à un bounded context, qui définit les limites d'un domaine métier.
- Encapsulation : Tout ce qui est dans un aggregate est protégé. Les modifications doivent se faire via les méthodes exposées par l'aggregate root.
- Invariant : Les invariants sont des règles métiers strictes qui doivent être respectées en permanence à l'intérieur de l'aggregate.



**MERCI
POUR VOTRE
ATTENTION**



@DANIEL WOAGOU

