



MODULE PROJECT MANAGEMENT

# Spécification des Services Métier

SERVICES\_SPEC.md

---

HR\_ONIAN · Spécification Technique · Février 2026

# Spécifications des Services Métier - Module GAC

**Version:** 1.0

**Date:** 01/02/2026

**Projet:** HR\_ONIAN

**Module:** Gestion des Achats & Commandes

## Vue d'ensemble

Les services métier encapsulent toute la logique métier du module GAC. Ils suivent le **Service Layer Pattern** pour séparer la logique métier des vues et permettre la réutilisation du code.

### Principes de conception

- **Single Responsibility** : Chaque service a une responsabilité unique
- **Transactionnalité** : Utilisation de `@transaction.atomic` pour garantir la cohérence
- **Gestion d'erreurs** : Exceptions personnalisées avec messages explicites
- **Logging** : Traçabilité de toutes les opérations importantes
- **Notifications** : Déclenchement automatique via NotificationService
- **Permissions** : Vérification des droits avant chaque opération sensible

## 1. DemandeService

### 1.1 Responsabilités

- Création et modification des demandes d'achat
- Gestion du workflow de validation (N1, N2)
- Calcul des totaux et vérification budgétaire
- Conversion en bon de commande
- Annulation et refus de demandes

### 1.2 Méthodes

**1.2.1** `creer_demande_brouillon(demandeur, objet, justification, departement=None, projet=None, budget=None)`

**Description** : Crée une demande d'achat en brouillon

**Paramètres :**

- `demandeur` (ZY00) : L'employé qui crée la demande
- `objet` (str) : L'objet de la demande
- `justification` (str) : La justification métier
- `departement` (DPMT, optionnel) : Le département concerné
- `projet` (Project, optionnel) : Le projet lié si applicable
- `budget` (GACBudget, optionnel) : L'enveloppe budgétaire à impacter

**Retour** : Instance de `GACDemandeAchat`

**Processus :**

1. Génération automatique du numéro : `DA-YYYY-NNNN`
2. Création de l'instance avec statut `BROUILLON`
3. Association au demandeur
4. Création de l'historique initial
5. Log de l'opération

**Exceptions :**

- `ValidationError` si les données sont invalides

**Code exemple :**

```
@transaction.atomic
def creer_demande_brouillon(demandeur, objet, justification, departement=None, projet=None, budget=None):
    """Crée une demande d'achat en brouillon."""

    # Générer le numéro
    annee = datetime.now().year
    dernier_numero = GACDemandeAchat.objects.filter(
        numero__startswith=f'DA-{annee}-'
    ).count()
    numero = f'DA-{annee}-{str(dernier_numero + 1).zfill(4)}'

    # Créer la demande
    demande = GACDemandeAchat.objects.create(
        numero=numero,
        demandeur=demandeur,
        objet=objet,
        justification=justification,
        departement=departement,
        projet=projet,
        budget=budget,
        statut='BROUILLON',
        date_creation=timezone.now(),
        cree_par=demandeur
    )

    # Créer l'historique
    HistoriqueService.ajouter_entree(
        objet=demande,
        utilisateur=demandeur,
        action='CREATION',
        description=f"Création de la demande {numero} en brouillon"
    )

    logger.info(f"Demande {numero} créée en brouillon par {demandeur}")

    return demande
```

**1.2.2** ajouter\_ligne(demande, article, quantite, prix\_unitaire, commentaire=None)**Description :** Ajoute une ligne à une demande d'achat**Paramètres :**

- `demande` (GACDemandeAchat) : La demande concernée
- `article` (GACArticle) : L'article à commander
- `quantite` (Decimal) : La quantité demandée
- `prix_unitaire` (Decimal) : Le prix unitaire estimé
- `commentaire` (str, optionnel) : Commentaire sur la ligne

**Retour :** Instance de `GACLigneDemandeAchat`**Processus :**

1. Vérifier que la demande est en brouillon ou en cours de modification
2. Créer la ligne avec calcul automatique du montant
3. Recalculer les totaux de la demande
4. Vérifier la disponibilité budgétaire si budget associé
5. Log de l'opération

**Exceptions :**

- `ValidationError` si la demande n'est pas modifiable
- `BudgetInsuffisantError` si le budget est dépassé

**Code exemple :**

```

@transaction.atomic
def ajouter_ligne(demande, article, quantite, prix_unitaire, commentaire=None):
    """Ajoute une ligne à une demande d'achat."""

    # Vérifier que la demande est modifiable
    if demande.statut not in ['BROUILLON', 'REFUSEE']:
        raise ValidationError("La demande n'est pas modifiable")

    # Créer la ligne
    ligne = GACLigneDemandeAchat.objects.create(
        demande_achat=demande,
        article=article,
        quantite=quantite,
        prix_unitaire=prix_unitaire,
        commentaire=commentaire
    )

    # Recalculer les totaux
    demande.calculer_totaux()

    # Vérifier le budget si défini
    if demande.budget:
        BudgetService.verifier_disponibilite(
            budget=demande.budget,
            montant=demande.montant_total_ttc
        )

    logger.info(f"Ligne ajoutée à {demande.numero}: {article.reference} x {quantite}")

    return ligne

```

### 1.2.3 soumettre\_demande(demande, utilisateur)

**Description :** Soumet une demande pour validation

**Paramètres :**

- `demande` (GACDemandeAchat) : La demande à soumettre
- `utilisateur` (ZY00) : L'utilisateur qui soumet

**Retour :** `demande` mise à jour

**Processus :**

1. Vérifier que la demande a au moins une ligne
2. Vérifier que le total est > 0
3. Déterminer le validateur N1 (manager du demandeur)
4. Passer le statut à `SOUMISE`
5. Enregistrer la date de soumission
6. Créer notification pour le validateur N1
7. Envoyer email au validateur N1
8. Créer l'historique
9. Log de l'opération

**Exceptions :**

- `ValidationError` si la demande est vide ou déjà soumise
- `ValidationError` si pas de validateur N1 trouvé

**Code exemple :**

```

@transaction.atomic
def soumettre_demande(demande, utilisateur):
    """Soumet une demande d'achat pour validation."""

    # Vérifications
    if demande.statut != 'BROUILLON':
        raise ValidationError("Seules les demandes en brouillon peuvent être soumises")

    if not demande.lignes.exists():
        raise ValidationError("La demande doit contenir au moins une ligne")

    if demande.montant_total_ttc <= 0:
        raise ValidationError("Le montant total doit être supérieur à 0")

    # Déterminer le validateur N1 (manager direct)
    validateur_n1 = demande.demandeur.responsable_hierarchique
    if not validateur_n1:
        raise ValidationError("Aucun validateur N1 (manager) trouvé pour cet employé")

    # Mettre à jour la demande
    demande.statut = 'SOUMISE'
    demande.date_soumission = timezone.now()
    demande.validateur_n1 = validateur_n1
    demande.save()

    # Créer notification
    NotificationService.notifier_validation_n1(demande)

    # Historique
    HistoriqueService.ajouter_entree(
        objet=demande,
        utilisateur=utilisateur,
        action='SOUMISSION',
        description=f"Demande soumise pour validation N1 à {validateur_n1}"
    )

    logger.info(f"Demande {demande.numero} soumise par {utilisateur}")

    return demande

```

#### 1.2.4 valider\_n1(demande, validateur, commentaire=None)

**Description :** Valide une demande au niveau N1 (manager)

**Paramètres :**

- `demande` (GACDemandeAchat) : La demande à valider
- `validateur` (ZY00) : Le validateur N1
- `commentaire` (str, optionnel) : Commentaire de validation

**Retour :** `demande` mise à jour

**Processus :**

1. Vérifier que le statut est `SOUMISE`
2. Vérifier que l'utilisateur est bien le validateur N1
3. Déterminer le validateur N2 selon les règles métier
4. Passer le statut à `VALIDEE_N1`
5. Enregistrer date et commentaire
6. Si montant < seuil validation N2 → passer directement à `VALIDEE_N2`
7. Sinon → créer notification pour validateur N2
8. Créer l'historique

**Exceptions :**

- `PermissionDenied` si l'utilisateur n'est pas le validateur N1
- `ValidationError` si le statut n'est pas correct

**Code exemple :**

```

@transaction.atomic
def valider_n1(demande, validateur, commentaire=None):
    """Valide une demande au niveau N1 (manager)."""

    # Vérifications
    if demande.statut != 'SOUmise':
        raise ValidationError("La demande doit être au statut SOUMISE")

    if demande.validateur_n1 != validateur:
        raise PermissionDenied("Vous n'êtes pas le validateur N1 de cette demande")

    # Mettre à jour la demande
    demande.statut = 'VALIDEE_N1'
    demande.date_validation_n1 = timezone.now()
    demande.commentaire_validation_n1 = commentaire
    demande.save()

    # Déterminer si validation N2 nécessaire
    SEUIL_VALIDATION_N2 = getattr(settings, 'GAC_SEUIL_VALIDATION_N2', 5000)

    if demande.montant_total_ttc < SEUIL_VALIDATION_N2:
        # Validation directe N2
        return valider_n2_auto(demande, validateur, "Validation automatique (montant < seuil)")
    else:
        # Déterminer validateur N2
        demande.validateur_n2 = determiner_validateur_n2(demande)
        demande.save()

        # Notifier validateur N2
        NotificationService.notifier_validation_n2(demande)

    # Historique
    HistoriqueService.ajouter_entree(
        objet=demande,
        utilisateur=validateur,
        action='VALIDATION_N1',
        description=f"Validation N1 par {validateur}. Commentaire: {commentaire or 'Aucun'}"
    )

    logger.info(f"Demande {demande.numero} validée N1 par {validateur}")

    return demande

```

### 1.2.5 valider\_n2(demande, validateur, commentaire=None)

**Description :** Valide une demande au niveau N2 (direction/achats)

**Paramètres :**

- `demande` (GACDemandeAchat) : La demande à valider
- `validateur` (ZY00) : Le validateur N2
- `commentaire` (str, optionnel) : Commentaire de validation

**Retour :** `demande` mise à jour

**Processus :**

1. Vérifier que le statut est `VALIDEE_N1`
2. Vérifier que l'utilisateur est bien le validateur N2 ou a le rôle ACHETEUR
3. Passer le statut à `VALIDEE_N2`
4. Enregistrer date et commentaire
5. Si budget défini → engager le montant
6. Notifier le demandeur de la validation
7. Notifier les acheteurs pour création BC
8. Créer l'historique

**Exceptions :**

- `PermissionDenied` si l'utilisateur n'a pas les droits
- `ValidationError` si le statut n'est pas correct
- `BudgetInsuffisantError` si engagement impossible

**Code exemple :**

```

@transaction.atomic
def valider_n2(demande, validateur, commentaire=None):
    """Valide une demande au niveau N2 (direction/achats)."""

    # Vérifications
    if demande.statut != 'VALIDEE_N1':
        raise ValidationError("La demande doit être au statut VALIDEE_N1")

    # Vérifier permissions
    if not (demande.validateur_n2 == validateur or
            validateur.has_role('ACHETEUR') or
            validateur.has_role('VALIDATEUR_N2')):
        raise PermissionDenied("Vous n'avez pas les droits de validation N2")

    # Mettre à jour la demande
    demande.statut = 'VALIDEE_N2'
    demande.date_validation_n2 = timezone.now()
    demande.commentaire_validation_n2 = commentaire
    demande.validateur_n2 = validateur
    demande.save()

    # Engager le budget si défini
    if demande.budget:
        BudgetService.engager_montant(
            budget=demande.budget,
            montant=demande.montant_total_ttc,
            reference=f"DA {demande.numero}"
        )

    # Notifications
    NotificationService.notifier_demande_validee(demande)

    # Historique
    HistoriqueService.ajouter_entree(
        objet=demande,
        utilisateur=validateur,
        action='VALIDATION_N2',
        description=f"Validation N2 par {validateur}. Commentaire: {commentaire or 'Aucun'}"
    )

    logger.info(f"Demande {demande.numero} validée N2 par {validateur}")

    return demande

```

### 1.2.6 `refuser_demande(demande, validateur, motif_refus)`

**Description :** Refuse une demande d'achat

**Paramètres :**

- `demande` (`GACDemandeAchat`) : La demande à refuser
- `validateur` (`ZY00`) : Le validateur qui refuse
- `motif_refus` (`str`) : Le motif du refus

**Retour :** `demande` mise à jour

**Processus :**

1. Vérifier que le validateur a les droits
2. Passer le statut à `REFUSEE`
3. Enregistrer le motif et la date
4. Libérer le budget engagé si applicable
5. Notifier le demandeur
6. Créer l'historique

**Code exemple :**

```

@transaction.atomic
def refuser_demande(demande, validateur, motif_refus):
    """Refuse une demande d'achat."""

    # Vérifications des permissions
    if demande.statut == 'SOUMISE' and demande.validateur_n1 != validateur:
        raise PermissionDenied("Vous n'êtes pas le validateur N1")

    if demande.statut == 'VALIDEE_N1' and demande.validateur_n2 != validateur:
        if not validateur.has_role('ACHETEUR'):
            raise PermissionDenied("Vous n'êtes pas le validateur N2")

    # Libérer le budget si engagé
    if demande.budget and demande.statut in ['VALIDEE_N1', 'VALIDEE_N2']:
        BudgetService.liberer_montant(
            budget=demande.budget,
            montant=demande.montant_total_ttc,
            reference=f"Refus DA {demande.numero}"
        )

    # Mettre à jour la demande
    demande.statut = 'REFUSEE'
    demande.motif_refus = motif_refus
    demande.date_refus = timezone.now()
    demande.save()

    # Notifier le demandeur
    NotificationService.notifier_demande_refusee(demande, motif_refus)

    # Historique
    HistoriqueService.ajouter_entree(
        objet=demande,
        utilisateur=validateur,
        action='REFUS',
        description=f"Demande refusée par {validateur}. Motif: {motif_refus}"
    )

    logger.info(f"Demande {demande.numero} refusée par {validateur}")

    return demande

```

### 1.2.7 annuler\_demande(demande, utilisateur, motif\_annulation)

**Description :** Annule une demande d'achat

**Paramètres :**

- `demande` (GACDemandeAchat) : La demande à annuler
- `utilisateur` (ZY00) : L'utilisateur qui annule
- `motif_annulation` (str) : Le motif de l'annulation

**Retour :** `demande` mise à jour

**Processus :**

1. Vérifier que l'utilisateur est le demandeur ou un admin
2. Vérifier que la demande n'est pas déjà convertie en BC
3. Passer le statut à `ANNULEE`
4. Libérer le budget engagé si applicable
5. Notifier les validateurs concernés
6. Créer l'historique

### 1.2.8 convertir\_en\_bon\_commande(demande, utilisateur, fournisseur, date\_livraison\_souhaitee=None)

**Description :** Convertit une demande validée en bon de commande

**Paramètres :**

- `demande` (GACDemandeAchat) : La demande à convertir
- `utilisateur` (ZY00) : L'utilisateur acheteur
- `fournisseur` (GACFournisseur) : Le fournisseur sélectionné
- `date_livraison_souhaitee` (date, optionnel) : Date de livraison souhaitée

**Retour :** Instance de `GACBonCommande` créé

**Processus :**

1. Vérifier que la demande est au statut `VALIDEE_N2`
2. Vérifier que l'utilisateur a le rôle ACHETEUR
3. Créer le bon de commande avec génération du numéro
4. Copier toutes les lignes de la demande
5. Calculer les totaux du BC
6. Passer le statut de la demande à `CONVERTIE_BC`
7. Lier le BC à la demande
8. Si budget → transférer engagement vers BC
9. Notifier le demandeur
10. Créer l'historique

**Exceptions :**

- `PermissionDenied` si pas le rôle ACHETEUR
- `ValidationError` si demande pas validée N2

**Code exemple :**

```
@transaction.atomic
def convertir_en_bon_commande(demande, utilisateur, fournisseur, date_livraison_souhaitee=None):
    """Convertit une demande validée en bon de commande."""

    # Vérifications
    if demande.statut != 'VALIDEE_N2':
        raise ValidationError("La demande doit être validée N2")

    if not utilisateur.has_role('ACHETEUR'):
        raise PermissionDenied("Seuls les acheteurs peuvent créer des BCs")

    # Créer le BC via BonCommandeService
    bc = BonCommandeService.creer_bon_commande(
        demande_achat=demande,
        fournisseur=fournisseur,
        acheteur=utilisateur,
        date_livraison_souhaitee=date_livraison_souhaitee
    )

    # Mettre à jour la demande
    demande.statut = 'CONVERTIE_BC'
    demande.bon_commande = bc
    demande.save()

    # Notifier
    NotificationService.notifier_bc_cree_depuis_demande(demande, bc)

    # Historique
    HistoriqueService.ajouter_entree(
        objet=demande,
        utilisateur=utilisateur,
        action='CONVERSION_BC',
        description=f"Convertie en BC {bc.numero}"
    )

    logger.info(f"Demande {demande.numero} convertie en BC {bc.numero}")

    return bc
```

**1.2.9** `get_demandes_en_attente_validation(utilisateur)`

**Description :** Récupère les demandes en attente de validation pour un utilisateur

**Paramètres :**

- `utilisateur` (ZY00) : L'utilisateur valideur

**Retour :** QuerySet de `GACDemandeAchat`

**Processus :**

1. Si valideur N1 → demandes SOUMISE où valideur\_n1 = utilisateur
2. Si valideur N2 ou ACHETEUR → demandes VALIDEE\_N1 où valideur\_n2 = utilisateur ou rôle ACHETEUR
3. Trier par priorité puis date de soumission

**Code exemple :**

```
def get_demandes_en_attente_validation(utilisateur):
    """Récupère les demandes en attente de validation pour un utilisateur."""

    from django.db.models import Q

    demandes = GACDemandeAchat.objects.none()

    # Demandes en attente validation N1
    demandes_n1 = GACDemandeAchat.objects.filter(
        statut='SOUMISE',
        valideur_n1=utilisateur
    )

    # Demandes en attente validation N2
    demandes_n2 = GACDemandeAchat.objects.filter(
        statut='VALIDEE_N1'
    ).filter(
        Q(valideur_n2=utilisateur) |
        Q(valideur_n2__isnull=True) # Si pas encore assigné
    )

    # Filtrer selon les rôles
    if utilisateur.has_role('ACHETEUR') or utilisateur.has_role('VALIDATEUR_N2'):
        demandes = demandes_n1 | demandes_n2
    else:
        demandes = demandes_n1

    # Trier par priorité et date
    return demandes.order_by('-priorite', 'date_soumission')
```

### 1.2.10 `get_statistiques_demandes(date_debut=None, date_fin=None)`

**Description :** Récupère les statistiques sur les demandes d'achat

**Paramètres :**

- `date_debut` (date, optionnel) : Date de début de la période
- `date_fin` (date, optionnel) : Date de fin de la période

**Retour :** Dictionnaire avec les statistiques

**Exemple de retour :**

```
{
    'total_demandes': 150,
    'par_statut': {
        'BROUILLON': 10,
        'SOUMISE': 15,
        'VALIDEE_N1': 8,
        'VALIDEE_N2': 20,
        'CONVERTIE_BC': 85,
        'REFUSEE': 10,
        'ANNULEE': 2
    },
    'montant_total': Decimal('125000.00'),
    'montant_moyen': Decimal('833.33'),
    'delai_validation_moyen_jours': 3.5,
    'taux_refus': 6.67,
    'top_demandeurs': [
        {'demandeur': 'AKAM Golda', 'nombre': 12},
        ...
    ]
}
```

## 2. BonCommandeService

### 2.1 Responsabilités

- Création et gestion des bons de commande
- Génération de PDF
- Envoi au fournisseur
- Suivi des réceptions

- Gestion du cycle de vie du BC

## 2.2 Méthodes

**2.2.1** `creer_bon_commande(demande_achat, fournisseur, acheteur, date_livraison_souhaitee=None, conditions_paiement=None)`

**Description** : Crée un bon de commande

**Paramètres** :

- `demande_achat` (GACDemandeAchat, optionnel) : Demande d'origine
- `fournisseur` (GACFournisseur) : Le fournisseur
- `acheteur` (ZY00) : L'acheteur qui crée le BC
- `date_livraison_souhaitee` (date, optionnel) : Date de livraison souhaitée
- `conditions_paiement` (str, optionnel) : Conditions de paiement spécifiques

**Retour** : Instance de `GACBonCommande`

**Processus** :

1. Générer le numéro : `BC-YYYY-NNNN`
2. Créer le BC avec statut `BROUILLON`
3. Si `demande_achat` → copier les lignes et les infos
4. Sinon → BC vierge
5. Récupérer conditions de paiement du fournisseur si non spécifiées
6. Créer l'historique
7. Log de l'opération

**Code exemple** :

```

@transaction.atomic
def creer_bon_commande(demande_achat=None, fournisseur=None, acheteur=None,
                       date_livraison_souhaitee=None, conditions_paieement=None):
    """Crée un bon de commande."""

    # Générer le numéro
    annee = datetime.now().year
    dernier_numero = GACBonCommande.objects.filter(
        numero__startswith=f'BC-{annee}-'
    ).count()
    numero = f'BC-{annee}-{str(dernier_numero + 1).zfill(4)}'

    # Conditions de paiement
    if not conditions_paieement and fournisseur:
        conditions_paieement = fournisseur.conditions_paieement

    # Créer le BC
    bc = GACBonCommande.objects.create(
        numero=numero,
        demande_achat=demande_achat,
        fournisseur=fournisseur,
        acheteur=acheteur,
        statut='BROUILLON',
        date_creation=timezone.now(),
        date_livraison_souhaitee=date_livraison_souhaitee,
        conditions_paieement=conditions_paieement,
        cree_par=acheteur
    )

    # Si création depuis demande, copier les lignes
    if demande_achat:
        for ligne_da in demande_achat.lignes.all():
            GACLigneBonCommande.objects.create(
                bon_commande=bc,
                article=ligne_da.article,
                quantite_commandee=ligne_da.quantite,
                prix_unitaire=ligne_da.prix_unitaire,
                commentaire=ligne_da.commentaire
            )

    # Calculer les totaux
    bc.calculer_totaux()

    # Historique
    HistoriqueService.ajouter_entree(
        objet=bc,
        utilisateur=acheteur,
        action='CREATION',
        description=f"Création du BC {numero}" +
            (f" depuis DA {demande_achat.numero}" if demande_achat else "")
    )

    logger.info(f"BC {numero} créé par {acheteur}")

    return bc

```

### 2.2.2 `emettre_bon_commande(bc, utilisateur)`

**Description :** Émet un bon de commande (finalise et verrouille)

**Paramètres :**

- `bc` (GACBonCommande) : Le bon de commande
- `utilisateur` (ZY00) : L'utilisateur acheteur

**Retour :** `bc` mis à jour

**Processus :**

1. Vérifier que le BC est en BROUILLON
2. Vérifier qu'il a au moins une ligne
3. Générer le PDF du BC
4. Passer au statut `EMIS`
5. Enregistrer la date d'émission
6. Si budget lié → transférer de "engagé" à "commandé"
7. Créer l'historique

**Code exemple :**

```

@transaction.atomic
def emettre_bon_commande(bc, utilisateur):
    """Émet un bon de commande."""

    # Vérifications
    if bc.statut != 'BROUILLON':
        raise ValidationError("Le BC doit être en brouillon")

    if not bc.lignes.exists():
        raise ValidationError("Le BC doit contenir au moins une ligne")

    # Générer le PDF
    pdf_content = generer_pdf_bon_commande(bc)

    # Sauvegarder le PDF
    from django.core.files.base import ContentFile
    bc.fichier_pdf.save(
        f'BC_{bc.numero}.pdf',
        ContentFile(pdf_content),
        save=False
    )

    # Mettre à jour le BC
    bc.statut = 'EMIS'
    bc.date_emission = timezone.now()
    bc.save()

    # Mettre à jour le budget
    if bc.demande_achat and bc.demande_achat.budget:
        BudgetService.commander_montant(
            budget=bc.demande_achat.budget,
            montant=bc.montant_total_ttc,
            reference=f"BC {bc.numero}"
        )

    # Historique
    HistoriqueService.ajouter_entree(
        objet=bc,
        utilisateur=utilisateur,
        action='EMISSION',
        description=f"BC {bc.numero} émis"
    )

    logger.info(f"BC {bc.numero} émis par {utilisateur}")

    return bc

```

**2.2.3** `envoyer_au_fournisseur(bc, utilisateur, email_destinataire=None)`**Description :** Envoie le BC au fournisseur par email**Paramètres :**

- `bc` (GACBonCommande) : Le bon de commande
- `utilisateur` (ZY00) : L'utilisateur acheteur
- `email_destinataire` (str, optionnel) : Email du fournisseur (sinon utilise celui du fournisseur)

**Retour :** `bc` mis à jour**Processus :**

1. Vérifier que le BC est EMIS
2. Vérifier qu'un PDF existe
3. Déterminer l'email du destinataire
4. Construire l'email avec le PDF en pièce jointe
5. Envoyer l'email
6. Passer au statut `ENVOYE`
7. Enregistrer date et email d'envoi
8. Créer l'historique

**Code exemple :**

```

@transaction.atomic
def envoyer_au_fournisseur(bc, utilisateur, email_destinataire=None):
    """Envoie le BC au fournisseur par email."""

    # Vérifications
    if bc.statut != 'EMIS':
        raise ValidationError("Le BC doit être émis avant envoi")

    if not bc.fichier_pdf:
        raise ValidationError("Aucun PDF généré")

    # Email destinataire
    if not email_destinataire:
        email_destinataire = bc.fournisseur.email

    if not email_destinataire:
        raise ValidationError("Aucun email de destinataire")

    # Construire l'email
    from django.core.mail import EmailMessage

    email = EmailMessage(
        subject=f"Bon de commande {bc.numero}",
        body=f"""\
Bonjour,

Veuillez trouver ci-joint notre bon de commande {bc.numero}.

Date de livraison souhaitée : {bc.date_livraison_souhaitee or 'À définir'}

Cordialement,
{utilisateur.get_full_name()}
Service Achats
""",
        from_email=settings.DEFAULT_FROM_EMAIL,
        to=[email_destinataire],
        reply_to=[utilisateur.email] if utilisateur.email else None
    )

    # Attacher le PDF
    email.attach_file(bc.fichier_pdf.path)

    # Envoyer
    email.send()

    # Mettre à jour le BC
    bc.statut = 'ENVOYE'
    bc.date_envoi = timezone.now()
    bc.email_envoi = email_destinataire
    bc.save()

    # Historique
    HistoriqueService.ajouter_entree(
        objet=bc,
        utilisateur=utilisateur,
        action='ENVOI',
        description=f"BC envoyé à {email_destinataire}"
    )

    logger.info(f"BC {bc.numero} envoyé à {email_destinataire}")

    return bc

```

#### 2.2.4 confirmer\_commande(bc, utilisateur, numero\_confirmation\_fournisseur=None, date\_livraison\_confirmee=None)

**Description :** Enregistre la confirmation du fournisseur

**Paramètres :**

- `bc` (GACBonCommande) : Le bon de commande
- `utilisateur` (ZY00) : L'utilisateur acheteur
- `numero_confirmation_fournisseur` (str, optionnel) : Numéro de confirmation du fournisseur
- `date_livraison_confirmee` (date, optionnel) : Date de livraison confirmée

**Retour :** `bc` mis à jour

**Processus :**

1. Vérifier que le BC est ENVOYE

2. Passer au statut `CONFIRME`
3. Enregistrer les infos de confirmation
4. Créer une alerte si date de livraison confirmée > date souhaitée
5. Créer l'historique

### 2.2.5 `annuler_bon_commande(bc, utilisateur, motif_annulation)`

**Description** : Annule un bon de commande

**Paramètres** :

- `bc` (`GACBonCommande`) : Le bon de commande à annuler
- `utilisateur` (`ZY00`) : L'utilisateur acheteur
- `motif_annulation` (`str`) : Le motif d'annulation

**Retour** : `bc` mis à jour

**Processus** :

1. Vérifier que le BC n'est pas déjà reçu
2. Passer au statut `ANNULE`
3. Libérer le budget
4. Notifier le fournisseur si déjà envoyé
5. Créer l'historique

### 2.2.6 `generer_pdf_bon_commande(bc)`

**Description** : Génère le PDF du bon de commande

**Paramètres** :

- `bc` (`GACBonCommande`) : Le bon de commande

**Retour** : Contenu du PDF (bytes)

**Processus** :

1. Utiliser ReportLab ou WeasyPrint
2. Template HTML → PDF
3. Inclure : logo entreprise, infos fournisseur, lignes, totaux, conditions

**Code exemple** :

```
def generer_pdf_bon_commande(bc):
    """Génère le PDF du bon de commande."""

    from django.template.loader import render_to_string
    from weasyprint import HTML

    # Rendre le template HTML
    html_content = render_to_string('gestion_achats/pdf/bon_commande.html', {
        'bc': bc,
        'entreprise': {
            'nom': settings.COMPANY_NAME,
            'adresse': settings.COMPANY_ADDRESS,
            'siret': settings.COMPANY_SIRET,
            'logo': settings.COMPANY_LOGO_PATH
        }
    })

    # Convertir en PDF
    pdf = HTML(string=html_content).write_pdf()

    return pdf
```

## 3. FournisseurService

### 3.1 Responsabilités

- Gestion du référentiel fournisseurs

- Évaluation des fournisseurs
- Statistiques et reporting
- Gestion des contacts

## 3.2 Méthodes

### 3.2.1 `creer_fournisseur(code, raison_sociale, siret, email, telephone, adresse, ...)`

**Description** : Crée un nouveau fournisseur

**Retour** : Instance de `GACFournisseur`

**Processus** :

1. Vérifier l'unicité du code et du SIRET
2. Valider le format SIRET
3. Créer le fournisseur avec statut `ACTIF`
4. Créer l'historique

### 3.2.2 `evaluer_fournisseur(fournisseur, evaluateur, note_qualite, note_delai, note_prix, commentaire=None)`

**Description** : Enregistre une évaluation de fournisseur

**Paramètres** :

- `fournisseur` (`GACFournisseur`) : Le fournisseur à évaluer
- `evaluateur` (`ZY00`) : L'utilisateur qui évalue
- `note_qualite` (int) : Note sur 5 pour la qualité
- `note_delai` (int) : Note sur 5 pour les délais
- `note_prix` (int) : Note sur 5 pour les prix
- `commentaire` (str, optionnel) : Commentaire

**Retour** : `fournisseur` mis à jour

**Processus** :

1. Créer une entrée d'évaluation (modèle `GACEvaluationFournisseur`)
2. Recalculer la note moyenne du fournisseur
3. Mettre à jour le champ `evaluation_moyenne`
4. Créer l'historique

### 3.2.3 `get_fournisseurs_pour_article(article)`

**Description** : Récupère les fournisseurs pouvant fournir un article

**Paramètres** :

- `article` (`GACArticle`) : L'article recherché

**Retour** : QuerySet de `GACFournisseur` triés par note

### 3.2.4 `get_statistiques_fournisseur(fournisseur, date_debut=None, date_fin=None)`

**Description** : Statistiques sur un fournisseur

**Retour** :

```
{
  'nombre_commandes': 25,
  'montant_total_commandes': Decimal('125000.00'),
  'montant_moyen_commande': Decimal('5000.00'),
  'taux_livraison_temps': 85.5, # %
  'delai_moyen_livraison_jours': 12,
  'evaluation_moyenne': 4.2,
  'nombre_evaluations': 10
}
```

## 4. ReceptionService

### 4.1 Responsabilités

- Enregistrement des réceptions
- Contrôle de conformité (quantité, qualité)
- Gestion des refus et litiges
- Mise à jour du statut des BCs

### 4.2 Méthodes

#### 4.2.1 `creer_reception(bon_commande, receptionnaire, date_reception=None)`

**Description :** Crée une réception de marchandises

**Paramètres :**

- `bon_commande` (GACBonCommande) : Le BC concerné
- `receptionnaire` (ZY00) : La personne qui réceptionne
- `date_reception` (date, optionnel) : Date de réception (défaut: aujourd'hui)

**Retour :** Instance de `GACReception`

**Processus :**

1. Générer le numéro : `REC-YYYY-NNNN`
2. Créer la réception avec statut `BROUILLON`
3. Copier les lignes du BC comme lignes de réception (quantité\_recue = 0)
4. Créer l'historique

**Code exemple :**

```
@transaction.atomic
def creer_reception(bon_commande, receptionnaire, date_reception=None):
    """Crée une réception de marchandises."""

    # Générer le numéro
    annee = datetime.now().year
    dernier_numero = GACReception.objects.filter(
        numero__startswith=f'REC-{annee}-'
    ).count()
    numero = f'REC-{annee}-{str(dernier_numero + 1).zfill(4)}'

    # Créer la réception
    reception = GACReception.objects.create(
        numero=numero,
        bon_commande=bon_commande,
        receptionnaire=receptionnaire,
        date_reception=date_reception or timezone.now().date(),
        statut='BROUILLON',
        cree_par=receptionnaire
    )

    # Copier les lignes du BC
    for ligne_bc in bon_commande.lignes.all():
        GACLigneReception.objects.create(
            reception=reception,
            ligne_bon_commande=ligne_bc,
            quantite_recue=0,
            quantite_acceptee=0,
            quantite_refusee=0
        )

    # Historique
    HistoriqueService.ajouter_entree(
        objet=reception,
        utilisateur=receptionnaire,
        action='CREATION',
        description=f"Création de la réception {numero} pour BC {bon_commande.numero}"
    )

    logger.info(f"Réception {numero} créée par {receptionnaire}")

    return reception
```

#### 4.2.2 enregistrar\_ligne\_reception(ligne\_reception, quantite\_recue, quantite\_acceptee, quantite\_refusee, conforme, commentaire=None)

**Description :** Enregistre les quantités reçues pour une ligne

**Paramètres :**

- `ligne_reception` (GACLigneReception) : La ligne concernée
- `quantite_recue` (Decimal) : Quantité totale reçue
- `quantite_acceptee` (Decimal) : Quantité acceptée
- `quantite_refusee` (Decimal) : Quantité refusée
- `conforme` (bool) : Conformité globale
- `commentaire` (str, optionnel) : Commentaire

**Retour :** `ligne_reception` mise à jour

**Processus :**

1. Vérifier que  $\text{quantite\_acceptee} + \text{quantite\_refusee} = \text{quantite\_recue}$
2. Mettre à jour la ligne
3. Mettre à jour les quantités reçues sur la ligne BC correspondante

**Code exemple :**

```
@transaction.atomic
def enregistrar_ligne_reception(ligne_reception, quantite_recue, quantite_acceptee,
                               quantite_refusee, conforme, commentaire=None):
    """Enregistre les quantités reçues pour une ligne."""

    # Vérifications
    if quantite_acceptee + quantite_refusee != quantite_recue:
        raise ValidationError("La somme acceptée + refusée doit égaler la quantité reçue")

    # Mettre à jour la ligne de réception
    ligne_reception.quantite_recue = quantite_recue
    ligne_reception.quantite_acceptee = quantite_acceptee
    ligne_reception.quantite_refusee = quantite_refusee
    ligne_reception.conforme = conforme
    ligne_reception.commentaire_reception = commentaire
    ligne_reception.save()

    # Mettre à jour la ligne BC
    ligne_bc = ligne_reception.ligne_bon_commande
    ligne_bc.quantite_recue = ligne_bc.lignes_reception.aggregate(
        total=Sum('quantite_acceptee')
    )['total'] or 0
    ligne_bc.save()

    logger.info(f"Ligne réception {ligne_reception.id} enregistrée: {quantite_acceptee} acceptées, {quantite_refusee} refusées")

    return ligne_reception
```

#### 4.2.3 valider\_reception(reception, utilisateur)

**Description :** Valide une réception complète

**Paramètres :**

- `reception` (GACReception) : La réception à valider
- `utilisateur` (ZY00) : L'utilisateur qui valide

**Retour :** `reception` mise à jour

**Processus :**

1. Vérifier que toutes les lignes ont des quantités
2. Déterminer la conformité globale
3. Passer au statut `VALIDEE`
4. Mettre à jour le statut du BC
5. Si totalement reçu → BC passe à `RECU_COMPLET`
6. Si partiellement reçu → BC passe à `RECU_PARTIEL`
7. Si budget → consommer le montant reçu

8. Créer l'historique

9. Notifier l'acheteur et le demandeur

**Code exemple :**

```
@transaction.atomic
def valider_reception(reception, utilisateur):
    """Valide une réception complète."""

    # Vérifications
    if reception.statut != 'BROUILLON':
        raise ValidationError("La réception doit être en brouillon")

    lignes = reception.lignes.all()
    if not lignes.exists():
        raise ValidationError("La réception doit contenir au moins une ligne")

    # Vérifier que toutes les lignes sont renseignées
    for ligne in lignes:
        if ligne.quantite_recue is None or ligne.quantite_recue == 0:
            raise ValidationError(f"Toutes les lignes doivent avoir une quantité reçue")

    # Déterminer conformité globale
    conformite_globale = all(ligne.conforme for ligne in lignes)

    # Mettre à jour la réception
    reception.statut = 'VALIDEE'
    reception.date_validation = timezone.now()
    reception.conforme = conformite_globale
    reception.save()

    # Mettre à jour le BC
    bc = reception.bon_commande
    bc.calculer_reception() # Recalcule quantites_recues

    if bc.est_totalelement_recu():
        bc.statut = 'RECU_COMPLET'
        bc.date_reception_complete = timezone.now()
    else:
        bc.statut = 'RECU_PARTIEL'

    bc.save()

    # Consommer le budget
    if bc.demande_achat and bc.demande_achat.budget:
        montant_recu = sum(
            ligne.quantite_acceptee * ligne.ligne_bon_commande.prix_unitaire
            for ligne in lignes
        )

        BudgetService.consommer_montant(
            budget=bc.demande_achat.budget,
            montant=montant_recu,
            reference=f"Réception {reception.numero}"
        )

    # Notifications
    NotificationService.notifier_reception_validee(reception)

    # Historique
    HistoriqueService.ajouter_entree(
        objet=reception,
        utilisateur=utilisateur,
        action='VALIDATION',
        description=f"Réception validée. Conformité: {'Oui' if conformite_globale else 'Non'}"
    )

    logger.info(f"Réception {reception.numero} validée par {utilisateur}")

    return reception
```

## 5. BudgetService

### 5.1 Responsabilités

- Contrôle de la disponibilité budgétaire

- Engagement des montants (lors de validation demande)
- Commande des montants (lors d'émission BC)
- Consommation des montants (lors de réception)
- Génération d'alertes budgétaires
- Reporting budgétaire

## 5.2 Méthodes

### 5.2.1 `verifier_disponibilite(budget, montant)`

**Description :** Vérifie qu'un montant est disponible sur un budget

**Paramètres :**

- `budget` (GACBudget) : L'enveloppe budgétaire
- `montant` (Decimal) : Le montant à vérifier

**Retour :** `True` si disponible, sinon lève une exception

**Exceptions :**

- `BudgetInsuffisantError` si `montant > disponible`

**Code exemple :**

```
def verifier_disponibilite(budget, montant):
    """Vérifie la disponibilité budgétaire."""

    disponible = budget.montant_disponible()

    if montant > disponible:
        raise BudgetInsuffisantError(
            f"Budget insuffisant. Disponible: {disponible} €, Demandé: {montant} €"
        )

    return True
```

### 5.2.2 `engager_montant(budget, montant, reference)`

**Description :** Engage un montant sur un budget (validation demande)

**Paramètres :**

- `budget` (GACBudget) : L'enveloppe budgétaire
- `montant` (Decimal) : Le montant à engager
- `reference` (str) : Référence de l'engagement (ex: "DA DA-2026-0001")

**Retour :** `budget` mis à jour

**Processus :**

1. Vérifier la disponibilité
2. Incrémenter `montant_engage`
3. Sauvegarder
4. Vérifier les seuils d'alerte
5. Créer l'historique

**Code exemple :**

```
@transaction.atomic
def engager_montant(budget, montant, reference):
    """Engage un montant sur un budget."""

    # Vérifier disponibilité
    verifier_disponibilite(budget, montant)

    # Engager
    budget.montant_engage += montant
    budget.save()

    # Vérifier alertes
    _verifier_seuils_alerte(budget)

    # Historique
    HistoriqueService.ajouter_entree(
        objet=budget,
        utilisateur=None,
        action='ENGAGEMENT',
        description=f"Engagement de {montant} € ({reference})"
    )

    logger.info(f"Montant {montant} engagé sur budget {budget.code} ({reference})")

    return budget
```

### 5.2.3 commander\_montant(budget, montant, reference)

**Description** : Passe un montant de "engagé" à "commandé" (émission BC)

**Processus** :

1. Décrémenter `montant_engage`
2. Incrémenter `montant_commande`
3. Sauvegarder

### 5.2.4 consommer\_montant(budget, montant, reference)

**Description** : Passe un montant de "commandé" à "consommé" (réception)

**Processus** :

1. Décrémenter `montant_commande`
2. Incrémenter `montant_consomme`
3. Sauvegarder
4. Vérifier les seuils d'alerte

### 5.2.5 liberer\_montant(budget, montant, reference)

**Description** : Libère un montant engagé ou commandé (annulation)

**Processus** :

1. Si `montant_engage > 0` → décrémenter `montant_engage`
2. Sinon → décrémenter `montant_commande`
3. Sauvegarder

### 5.2.6 \_verifier\_seuils\_alerte(budget)

**Description** : Vérifie si les seuils d'alerte sont atteints

**Processus** :

1. Calculer le taux de consommation
2. Si `taux >= seuil_alerte_1` → créer alerte INFO
3. Si `taux >= seuil_alerte_2` → créer alerte AVERTISSEMENT
4. Si dépassement → créer alerte CRITIQUE

**Code exemple** :

```
def _verifier_seuils_alerte(budget):
    """Vérifie les seuils d'alerte budgétaire."""

    taux = budget.taux_consommation()

    if taux >= budget.seuil_alerte_2 and not budget.alerte_2_envoyee:
        NotificationService.notifier_alerte_budget(
            budget=budget,
            niveau='CRITIQUE',
            message=f"Budget {budget.code}: {taux}% consommé (seuil 2: {budget.seuil_alerte_2}%)"
        )
        budget.alerte_2_envoyee = True
        budget.save()

    elif taux >= budget.seuil_alerte_1 and not budget.alerte_1_envoyee:
        NotificationService.notifier_alerte_budget(
            budget=budget,
            niveau='AVERTISSEMENT',
            message=f"Budget {budget.code}: {taux}% consommé (seuil 1: {budget.seuil_alerte_1}%)"
        )
        budget.alerte_1_envoyee = True
        budget.save()
```

### 5.2.7 `get_synthese_budgets(exercice=None)`

**Description :** Synthèse de tous les budgets

**Retour :**

```
{
    'total_initial': Decimal('500000.00'),
    'total_engage': Decimal('125000.00'),
    'total_commande': Decimal('80000.00'),
    'total_consomme': Decimal('60000.00'),
    'total_disponible': Decimal('375000.00'),
    'taux_consommation_global': 25.0,
    'budgets_en_alerte': [
        {'code': 'BUD-2026-IT', 'taux': 85.5},
        ...
    ]
}
```

## 6. CatalogueService

### 6.1 Responsabilités

- Gestion du catalogue produits
- Gestion des catégories
- Association articles-fournisseurs
- Gestion des prix

### 6.2 Méthodes

#### 6.2.1 `creer_categorie(nom, parent=None, description=None)`

**Description :** Crée une catégorie de produits

**Retour :** Instance de `GACCategorie`

#### 6.2.2 `creer_article(reference, designation, categorie, prix_unitaire, unite, ...)`

**Description :** Crée un article dans le catalogue

**Retour :** Instance de `GACArticle`

**Processus :**

1. Vérifier l'unicité de la référence

2. Créer l'article
3. Créer l'historique

**6.2.3** `associer_fournisseur_article(article, fournisseur, prix_fournisseur, delai_livraison, reference_fournisseur=None)`

**Description** : Associe un fournisseur à un article avec ses conditions

**Retour** : Relation créée

**6.2.4** `rechercher_articles(query, categorie=None, actif_uniquement=True)`

**Description** : Recherche d'articles dans le catalogue

**Paramètres** :

- `query` (str) : Terme de recherche (référence, désignation)
- `categorie` (GACCategorie, optionnel) : Filtrer par catégorie
- `actif_uniquement` (bool) : Ne retourner que les articles actifs

**Retour** : QuerySet de `GACArticle`

## 7. NotificationService

### 7.1 Responsabilités

- Création de notifications in-app
- Envoi d'emails
- Gestion des préférences de notification

### 7.2 Méthodes

**7.2.1** `notifier_validation_n1(demande)`

**Description** : Notifie le validateur N1 qu'une demande est en attente

**Processus** :

1. Créer notification in-app pour validateur\_n1
2. Envoyer email au validateur\_n1
3. Contenu: lien vers la demande, montant, demandeur

**7.2.2** `notifier_validation_n2(demande)`

**Description** : Notifie le validateur N2 qu'une demande est en attente

**7.2.3** `notifier_demande_validee(demande)`

**Description** : Notifie le demandeur que sa demande est validée

**7.2.4** `notifier_demande_refusee(demande, motif)`

**Description** : Notifie le demandeur que sa demande est refusée

**7.2.5** `notifier_bc_cree_depuis_demande(demande, bc)`

**Description** : Notifie le demandeur qu'un BC a été créé

**7.2.6** `notifier_reception_validee(reception)`

**Description** : Notifie l'acheteur et le demandeur de la réception

### 7.2.7 notifier\_alerte\_budget(budget, niveau, message)

**Description** : Notifie les gestionnaires de budget d'une alerte

## 8. HistoriqueService

### 8.1 Méthodes

#### 8.1.1 ajouter\_entree(objet, utilisateur, action, description)

**Description** : Ajoute une entrée dans l'historique

**Paramètres** :

- `objet` : L'objet concerné (GenericForeignKey)
- `utilisateur` (ZY00, optionnel) : L'utilisateur auteur
- `action` (str) : Code de l'action (CREATION, MODIFICATION, VALIDATION, etc.)
- `description` (str) : Description textuelle

**Retour** : Instance de `GACHistorique`

**Code exemple** :

```
def ajouter_entree(objet, utilisateur, action, description):
    """Ajoute une entrée dans l'historique."""

    from django.contrib.contenttypes.models import ContentType

    historique = GACHistorique.objects.create(
        content_type=ContentType.objects.get_for_model(objet),
        object_id=obj.pk,
        utilisateur=utilisateur,
        action=action,
        description=description,
        date_action=timezone.now()
    )

    return historique
```

## 9. Exceptions personnalisées

Créer dans `gestion_achats/exceptions.py` :

```
class GACException(Exception):
    """Exception de base pour le module GAC."""
    pass

class BudgetInsuffisantError(GACException):
    """Exception levée quand le budget est insuffisant."""
    pass

class WorkflowError(GACException):
    """Exception levée pour les erreurs de workflow."""
    pass

class ValidationError(GACException):
    """Exception levée pour les erreurs de validation métier."""
    pass
```

## 10. Helpers et utilitaires

### 10.1 determiner\_validateur\_n2(demande)

**Description** : Détermine le validateur N2 selon les règles métier

**Règles :**

- Si montant > 10 000 € → Direction générale
- Si catégorie IT → Responsable IT + Achats
- Sinon → Responsable achats

**Retour :** Instance de `ZY00`

---

**10.2** `calculer_delai_validation(demande)`

**Description :** Calcule le délai de validation d'une demande

**Retour :** Nombre de jours ouvrés

---

## 11. Logs et traçabilité

Tous les services doivent logger :

- Les opérations importantes (création, validation, refus, etc.)
- Les erreurs et exceptions
- Les alertes budgétaires

Exemple de configuration :

```
import logging
logger = logging.getLogger('gestion_achats')

# Dans chaque méthode importante
logger.info(f"Demande {demande.numero} créée par {utilisateur}")
logger.warning(f"Budget {budget.code} atteint {taux}% de consommation")
logger.error(f"Erreur lors de la génération du PDF: {e}")
```

---

## 12. Tests unitaires

Chaque service doit avoir des tests couvrant :

- Les cas nominaux
- Les cas d'erreur
- Les validations
- Les calculs
- Les workflows

Exemple de structure :

```
# tests/test_demande_service.py
class DemandeServiceTestCase(TestCase):
    def test_creer_demande_brouillon(self):
        """Test création demande en brouillon."""
        ...

    def test_soumettre_demande_sans_ligne(self):
        """Test soumission demande vide (doit échouer)."""
        ...

    def test_workflow_complet_validation(self):
        """Test du workflow complet de validation."""
        ...
```

---

### Fin des spécifications des services