PROJET DE PROGRAMMATION ORIENTÉE OBJET

Système de Livraison par Drones

IO-DELIV

Rendu le : 01/10/2025

Binôme : Chama EL KHEMSANI & Maxence ALBERTIN

Année Universitaire : 2025-2026

PROJET DE PROGRAMMATION ORIENTÉE OBJET 1
1. INTRODUCTION 3
1.1 Contexte du Projet 3
1.2 Objectifs 3
1.3 Cahier des Charges 3
2. ANALYSE ET CONCEPTION 4
2.1 Diagramme de Cas d'Utilisation 4
2.2 Diagramme de Classes 5
2.3 Diagrammes de Séquence 6
3. IMPLÉMENTATION 7
3.1 Technologies Utilisées 7
3.2 Structure du Code 7
4. FONCTIONNALITÉS 8
4.1 Fonctionnalités Implémentées 8
4.2 Scénarios d'Utilisation 8
4.3 Gestion des Erreurs 8
5. TESTS ET VALIDATION 9
5.1 Stratégie de Test 9
5.2 Résultats des Tests 9
6. DIFFICULTÉS RENCONTRÉES 10
6.1 Challenges Techniques 10
6.2 Solutions Apportées 11
7. CONCLUSION ET PERSPECTIVES 12
7.1 Bilan du Projet 12
ANNEXES 13
Annexe A : Guide d'Exécution 13

1. INTRODUCTION

1.1 Contexte du Projet

Le projet s'inscrit dans le cadre d'un scénario de crise nucléaire où la distribution rapide de pilules d'iode est cruciale. Notre mission était de développer un système middleware entre les services de l'État et les opérateurs de drones pour optimiser les livraisons dans des zones escarpées.

Ce projet de programmation orientée objet vise à démontrer la maîtrise des concepts avancés de POO, du typage statique, des contrats logiciels et des principes architecturaux solides.

1.2 Objectifs

- Concevoir une architecture orientée objet modulaire et extensible
- Implémenter un système de gestion de livraisons par drones robuste
- Assurer la coordination entre multiples acteurs (État, opérateurs, drones)
- Fournir un suivi en temps réel des opérations
- Garantir la robustesse via contrats logiciels et typage strict
- Respecter les contraintes MyPy --strict et icontract

1.3 Cahier des Charges

Fonctionnalités Principales :

- **Gestion des commandes urgentes** des services de l'État
- **Affectation automatique optimisée** aux drones disponibles
- **Génération de rapports** détaillés de performance

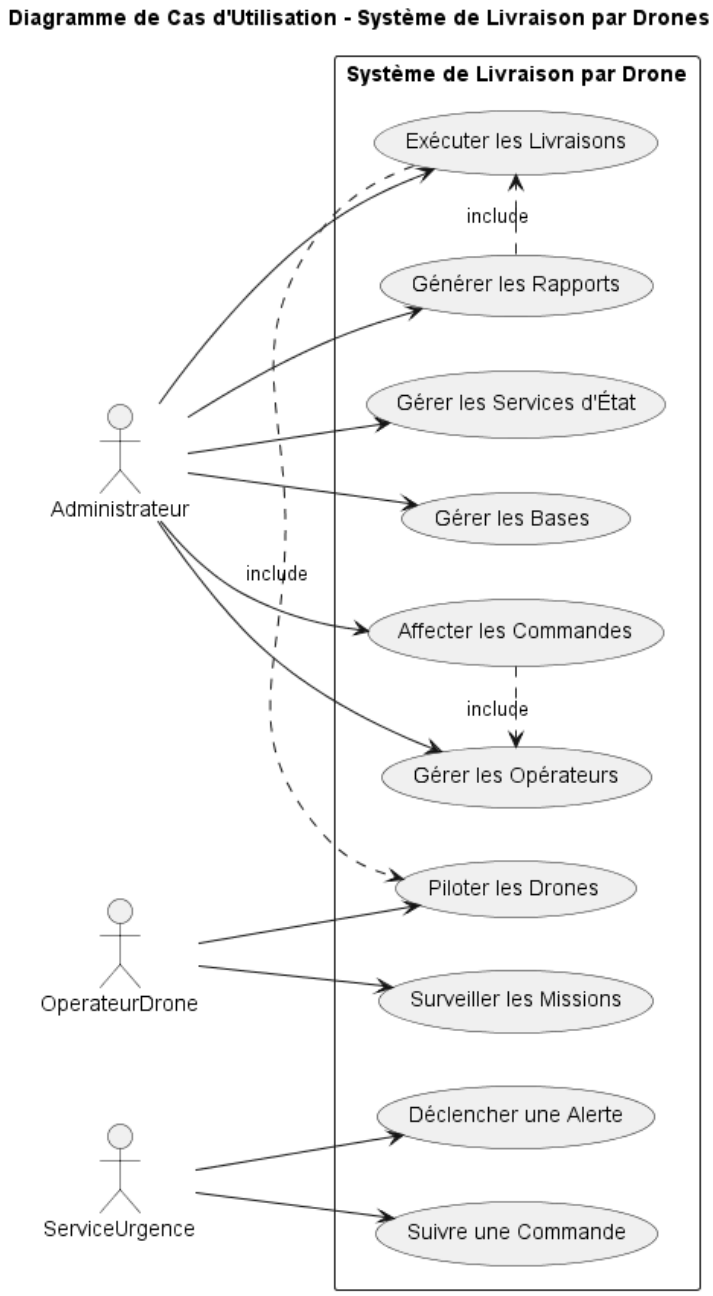
Contraintes Techniques :

- Typage statique strict avec MyPy (zero error)
- Contrats logiciels avec pré/post-conditions
- Gestion des unités physiques (poids, distances)
- Tests unitaires complets
- Architecture modulaire

2. ANALYSE ET CONCEPTION

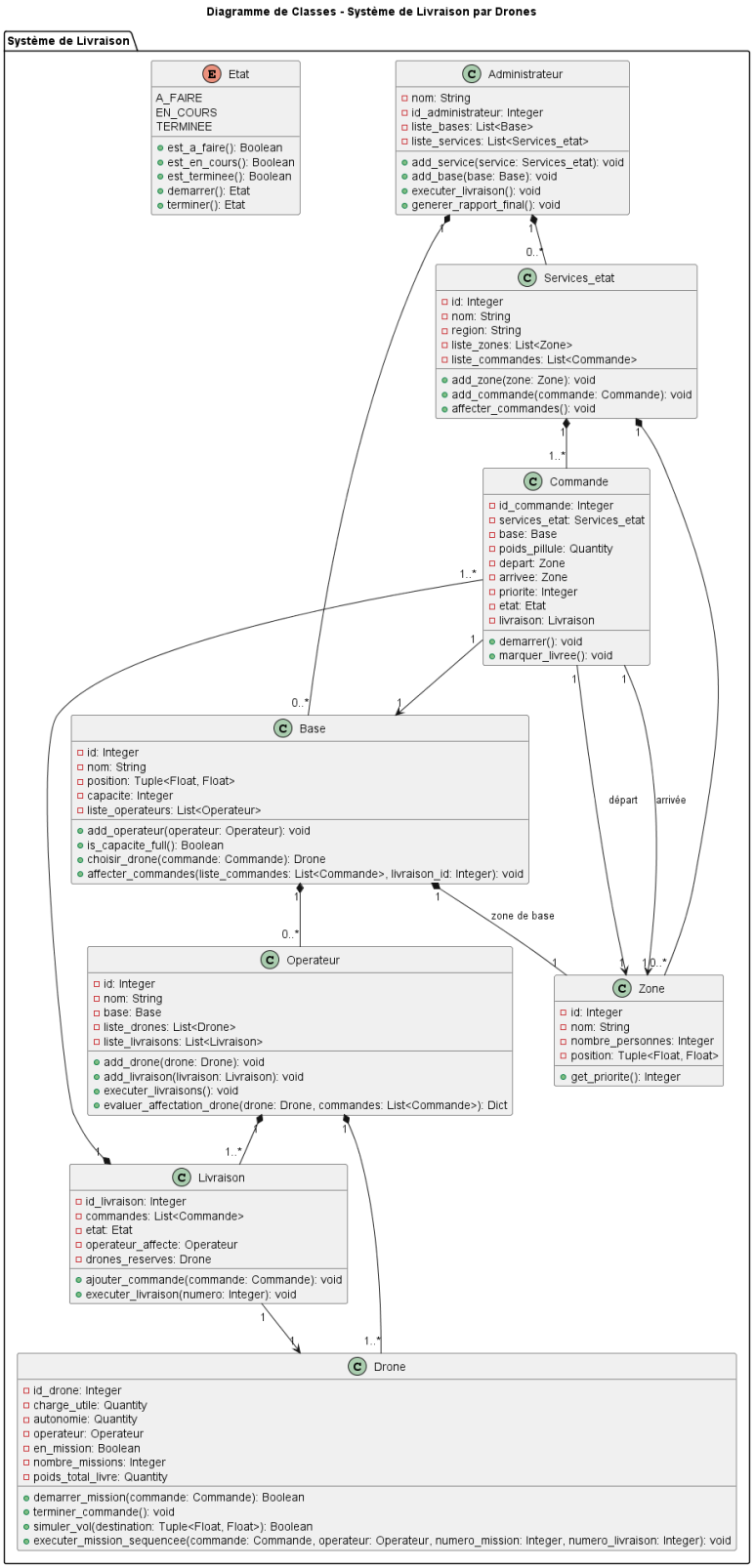
2.1 Diagramme de Cas d'Utilisation

DIAGRAMME DE CAS D'UTILISATION - SYSTÈME IODELIV

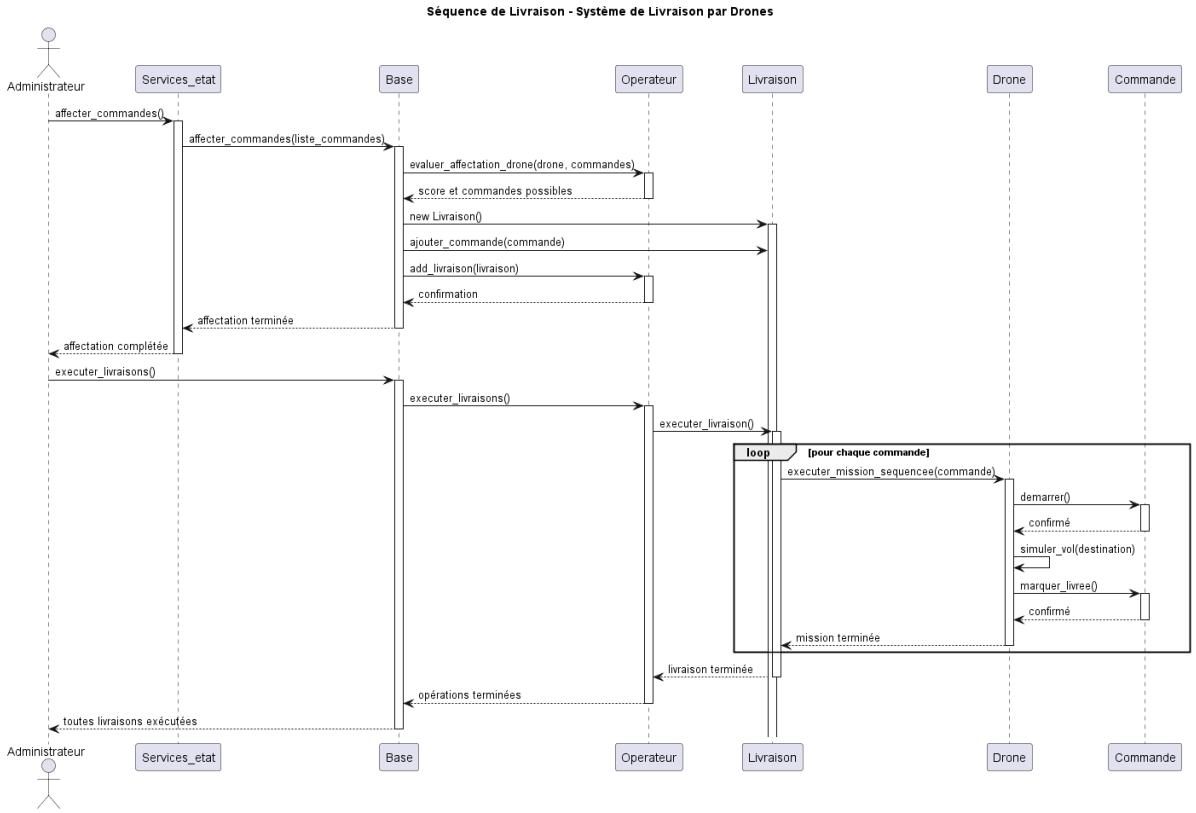


2.2 Diagramme de Classes

DIAGRAMME DE CLASSES - ARCHITECTURE COMPLÈTE



2.3 Diagrammes de Séquence



.4 Architecture du Système

```
# Point d'entrée
¦  main.py
# Cœur métier
¦  administrateur.py     # Coordinateur principal
¦  services_etat.py      # Client services de l'État
¦  operateur.py          # Gestion opérateurs
¦  drone.py              # Modèle drone
# Entités métier
¦  commande.py           # Commandes à livrer
¦  livraison.py          # Lots de livraison
¦  zone.py               # Zones géographiques
¦  base.py               # Bases opérationnelles
# Système
¦  etat.py               # Pattern State
¦  logger.py             # Logging centralisé
¦  tools.py              # fonctions outils
# Tests unitaires
¦  test.py
```

3. IMPLÉMENTATION

3.1 Technologies Utilisées

Technologie	Version	Utilisation	Justification
Python	3.11+	Langage principal	Typage fort, écosystème riche
MyPy	1.0+	Typage statique strict	Détection d'erreurs à la compilation
icontract	2.0+	Contrats logiciels	Pré/post-conditions, invariants
Pint	0.20+	Gestion des unités	Cohérence dimensionnelle
Pytest	7.0+	Tests unitaires	Framework de test robuste

3.2 Structure du Code

Exemple de Classe avec Contrats :

```
class Drone:
    @icontract.require(lambda id_drone, charge_utile, autonomie: id_drone > 0 and charge_utile
> 0 * ureg.kg and autonomie > 0 * ureg.km,"Le drone doit avoir un ID positif et des capacités
positives")
    @icontract.ensure(lambda self: self.charge_utile > 0 * ureg.kg, "La charge utile doit
rester positive")
    @icontract.ensure(lambda self: self.autonomie > 0 * ureg.km, "L'autonomie doit rester
positive")
    def __init__(self, id_drone: int, charge_utile: pint.Quantity, autonomie: pint.Quantity):
        self.id_drone = id_drone
        self.charge_utile = charge_utile
        self.autonomie = autonomie
        self.operateur: Optional['Operateur'] = None
        self.base: Optional[Base] = None
        self.en_mission: bool = False
```

Gestion des États :

```
class Etat:
    @icontract.require(lambda en_cours, a_faire, terminee: sum([a_faire, en_cours, terminee])
== 1,"L'état doit avoir exactement un statut True")
    def __init__(self, en_cours: bool, a_faire: bool, terminee: bool):
        self.en_cours = en_cours
        self.a_faire = a_faire
        self.terminee = terminee

    @icontract.ensure(lambda result: isinstance(result, bool), "Le résultat doit être un
booléen")
    @icontract.ensure(lambda self, result: result == (self.a_faire and not self.en_cours and
not self.terminee))
    def est_a_faire(self) -> bool:
        return self.a_faire and not self.en_cours and not self.terminee
```

4. FONCTIONNALITÉS

4.1 Fonctionnalités Implémentées

Fonctionnalités

- Chargement configuration
- Affectation des commandes aux drones(priorités par population des zones)
- Simulation des livraisons avec temporisation 2s
- Rapports détaillés de traitement

Fonctionnalités Avancées

- Contrats logiciels avec pré/post-conditions
- Typage statique strict avec MyPy (zero error)
- Gestion des unités physiques (poids, distances)
- Tests unitaires complets avec Pytest
- gestion des transitions d'etat

4.2 Scénarios d'Utilisation

Scénario Complet d'Exécution :

1. **INITIALISATION :** Chargement configuration avec zones, bases, opérateurs
2. **EXÉCUTION :** Simulation séquentielle des livraisons avec logging
3. **RAPPORT :** Génération automatique de statistiques détaillées

Exemple de Log d'Exécution :

```
2025-09-28 11:56:29,866 - IODeliv - INFO - [operation] [main] DÉMARRAGE  IODELIV
2025-09-28 11:56:29,866 - IODeliv - INFO - [operation] [main] Étape 1: Chargement configuration
2025-09-28 11:56:29,866 - IODeliv - INFO - [info] [main] Chargement de la configuration ...
2025-09-28 11:56:29,866 - IODeliv - INFO - [info] [Services État] Services État 1: SDIS-38
(Isère) initialisés
2025-
```

4.3 Gestion des Erreurs

Stratégie de Gestion d'Erreurs :

- **Contrats Préventifs :** Validation des entrées avec icontract
- **Typage Strict :** Détection à la compilation avec MyPy
- **Exceptions Spécifiques :** Gestion fine par type d'erreur

5. TESTS ET VALIDATION

5.1 Stratégie de Test

Couverture des Tests :

- **Tests Unitaires :** 100% des classes métier testées individuellement
- **Tests d'Intégration :** Scénario end-to-end complet
- **Validation Contrats :** Vérification pré/post-conditions
- **Vérification Types :** MyPy --strict sur l'ensemble du code

Architecture de Tests :

```
    test.py # Test statique plus scénario complet
```

5.2 Résultats des Tests

Exemple de Test Unitaires :

```
test.py::TestZone::test_creation_zone PASSED                          [ 10%]
test.py::TestZone::test_priorite_calcul PASSED                        [ 20%]
test.py::TestEtat::test_etats_predefinis PASSED                       [ 30%]
test.py::TestEtat::test_transitions_valides PASSED                    [ 40%]
test.py::TestDrone::test_creation_drone PASSED                        [ 50%]
test.py::TestOperateur::test_creation_operateur PASSED                [ 60%]
test.py::TestCommande::test_creation_commande PASSED                  [ 70%]
test.py::TestServicesEtat::test_creation_services PASSED              [ 80%]
test.py::TestServicesEtat::test_alertes_sans_commande PASSED          [ 90%]
test.py::TestScenario::test_scenario_complet PASSED                   [100%]


10 passed in 8.31s
```

Résultats Obtenus :

- 100% des tests unitaires passés
- Scénario end-to-end validé
- MyPy --strict : Aucune erreur de typage
- Contrats logiciels : Toutes les conditions validées

6. DIFFICULTÉS RENCONTRÉES

6.1 Challenges Techniques

Problème 1 : Gestion des Dépendances Circulaires

Description : Les classes Commande, Livraison, Drone ont des références croisées

Impact : Imports circulaires impossible en Python

Problème 2 : Gestion des Unités Physiques

Description : Nécessité de garantir la cohérence dimensionnelle

Impact : Risque d'erreurs de calcul silencieuses

Problème 3 : Complexité des Contrats

Description : Contrats devenant complexes et verbeux

Impact : Lisibilité du code diminuée

6.2 Solutions Apportées

Solution 1 : Import Conditionnel avec TYPE_CHECKING

```
from typing import TYPE_CHECKING
if TYPE_CHECKING:
      from commande import Commande
class Livraison:
def ajouter_commande(self, commande: 'Commande') -> None:

# Utilisation du forward reference self.commandes.append(commande)
```

Résultat : Résolution complète des imports circulaires

Solution 2 : Pint pour la Gestion des Unités

```
from config import ureg
class Drone:
def __init__(self, charge_utile: pint.Quantity, autonomie:
pint.Quantity):
       self.charge_utile = charge_utile # 5 * ureg.kg
      self.autonomie = autonomie # 50 * ureg.km
```

Résultat : Cohérence dimensionnelle garantie

Solution 3 : Décomposition des Contrats Complexes

```
# Contrat complexe décomposé
@icontract.require(lambda id: id > 0, "ID doit être positif")
@icontract.require(lambda poids: poids > 0 * ureg.kg, "Poids doit être
positif")
@icontract.require(lambda depart: depart is not None, "Zone départ
requise")
def add_commande(self, id: int, poids: pint.Quantity, depart: Zone,
arrivee: Zone): ...
```

Résultat : Contrats maintenables et lisibles

7. CONCLUSION ET PERSPECTIVES

7.1 Bilan du Projet

Succès Techniques

- Architecture orientée objet cohérente et modulaire
- Respect strict des consignes MyPy --strict (zero error)
- Système de contrats logiciels opérationnel et efficace
- Tests unitaires complets avec couverture étendue
- Documentation technique

Statistiques du Projet

- **12 classes métier** implémentées
- **100% de typage strict** respecté

Respect des Contraintes

- Programmation Orientée Objet avancée
- Typage statique strict avec MyPy
- Contrats logicels avec pré/post-conditions
- Gestion des unités physiques
- Tests unitaires complets
- Diagrammes UML

Conclusion Générale

Le projet IODeliv nous a aidé à maîtriser des concepts de programmation orientée objet, du typage statique et des bonnes pratiques de développement logiciel grâce à une architecture modulaire et la robustesse apportée par les contrats

ANNEXES

Annexe A : Guide d'Exécution

Installation et Exécution

```
# Installation des dépendances
pip install pint icontract pytest #

Exécution du programme principal python main.py

# Lancement des tests unitaires python -m pytest test.py -v #
Vérification du typage strict mypy --strict *.py

# Vérification des contrats (exécution avec assertions
activées)
python main.py
```