# Relations entre Objets

**Définition** :

Les relations décrivent comment les objets interagissent.

Il existe 4 types de relations:

Association, agrégation, composition et dérivation

L’héritage permet de partager ou étendre des comportements entre classes.

Le polymorphisme fournit une interface commune à différents types d'objets.

# L’association

**Définition:**
Relation simple où deux objets interagissent sans dépendance forte.

Une association représente un lien unissant les instances de classe. Elle repose sur la relation « a-un » ou « utilise-un ».

```python
class IoTDevice:
    def send_data(self):
        return "IoTDevice data"

class MonitoringServer:
    def monitor(self, device):
        print(f"Monitoring: {device.send_data()}")
```

# L’agrégation

**Définition:**
Un objet peut contenir d’autres objets, qui peuvent cependant exister indépendamment.

Une agrégation est une association non symétrique entre deux classes (*l'agrégat* et le *composant*).

```python
class IoTDevice:
    def __init__(self, device_id):
        self.device_id = device_id

class Network:
    def add_device(self, device):
        self.devices.append(device)
```

# La composition

**Définition:**

Une composition est un type particulier d'agrégation dans laquelle la vie des composants est liée à celle de l'agrégat

```python
class SecurityEvent:
    def __init__(self, event_type, description):
        self.event_type = event_type
        self.description = description

class SecurityAlert:
    def __init__(self, alert_id):
        self.alert_id = alert_id
        self.events = []

    def add_event(self, event_type, description):
        self.events.append(SecurityEvent(event_type, description))
```

# La dérivation: l’héritage

**Définition** : L'héritage permet de créer une classe dérivée à partir d'une classe existante.

```
class BaseClass:
    pass # Code de la classe de base

class DerivedClass(BaseClass):
    pass # Code spécifique à la classe dérivée
```

**Avantages :**

- Réutilisation du code
- Ajout ou modification des fonctionnalités

# L’héritage

```python
class EquipementReseau:
    def __init__(self, ip, mac):
        self.ip = ip
        self.mac = mac

    def afficher_infos(self):
        return f"IP: {self.ip}, MAC: {self.mac}"

class Routeur(EquipementReseau):
    def __init__(self, ip, mac, protocole_routage):
        super().__init__(ip, mac)
        self.protocole_routage = protocole_routage

    def afficher_infos(self):
        return f"""{super().afficher_infos()},
        Protocole: {self.protocole_routage}"""
```

# Exemple

```python
equipements = [
    EquipementReseau("192.168.1.1", "AA:BB:CC:DD:EE:FF"),
    Routeur("192.168.1.254", "11:22:33:44:55:66", "OSPF"),
]

for equipement in equipements:
    print(equipement.afficher_infos())
```

# Le polymorphisme

**Définition** :

Le polymorphisme permet d'utiliser une même interface pour des objets de classes différentes.

**Avantages** :

• Simplifie la gestion des objets divers.

• Facilite l'extension et la maintenance.

.

# Le polymorphisme

```python
class Device:
    def communicate(self):
        pass

class IoTDevice(Device):
    def communicate(self):
        return "Communicating via MQTT"

class Sensor(Device):
    def communicate(self):
        return "Sending data via HTTP"

def test_communication(device):
    print(device.communicate())
```