

Multitâche & objets

Modélisation objet des paradigmes multitâches



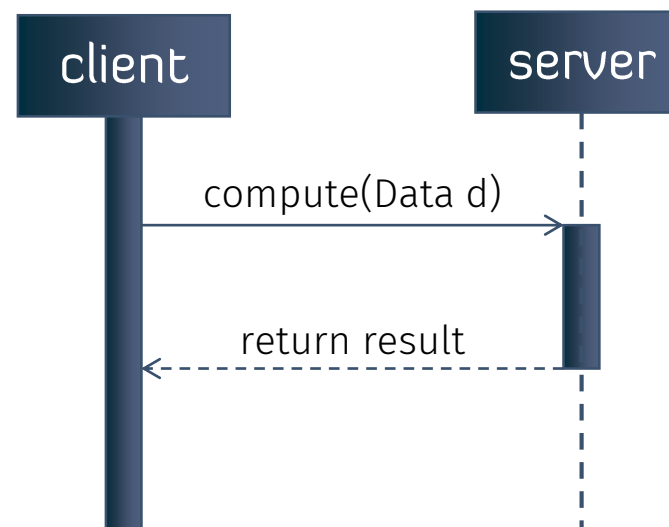
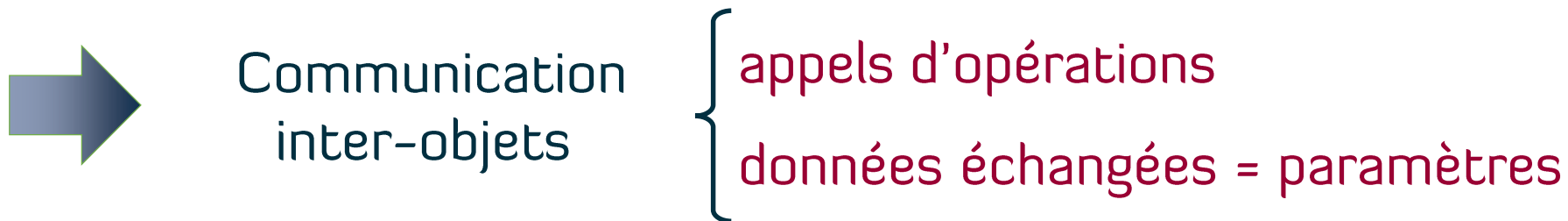
© Shebli Anvar

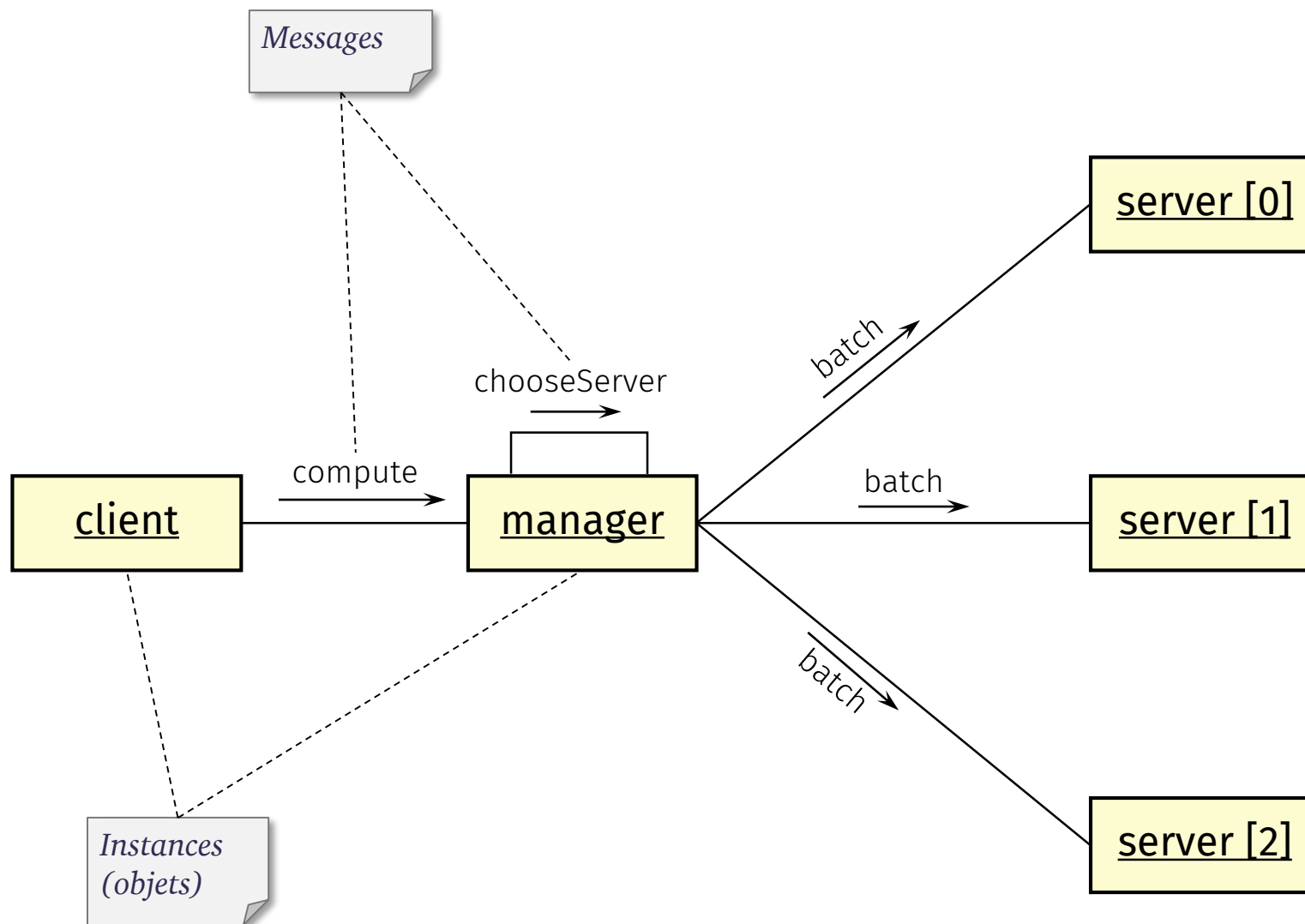
CEA — Institut de recherches sur les lois fondamentales de l'Univers

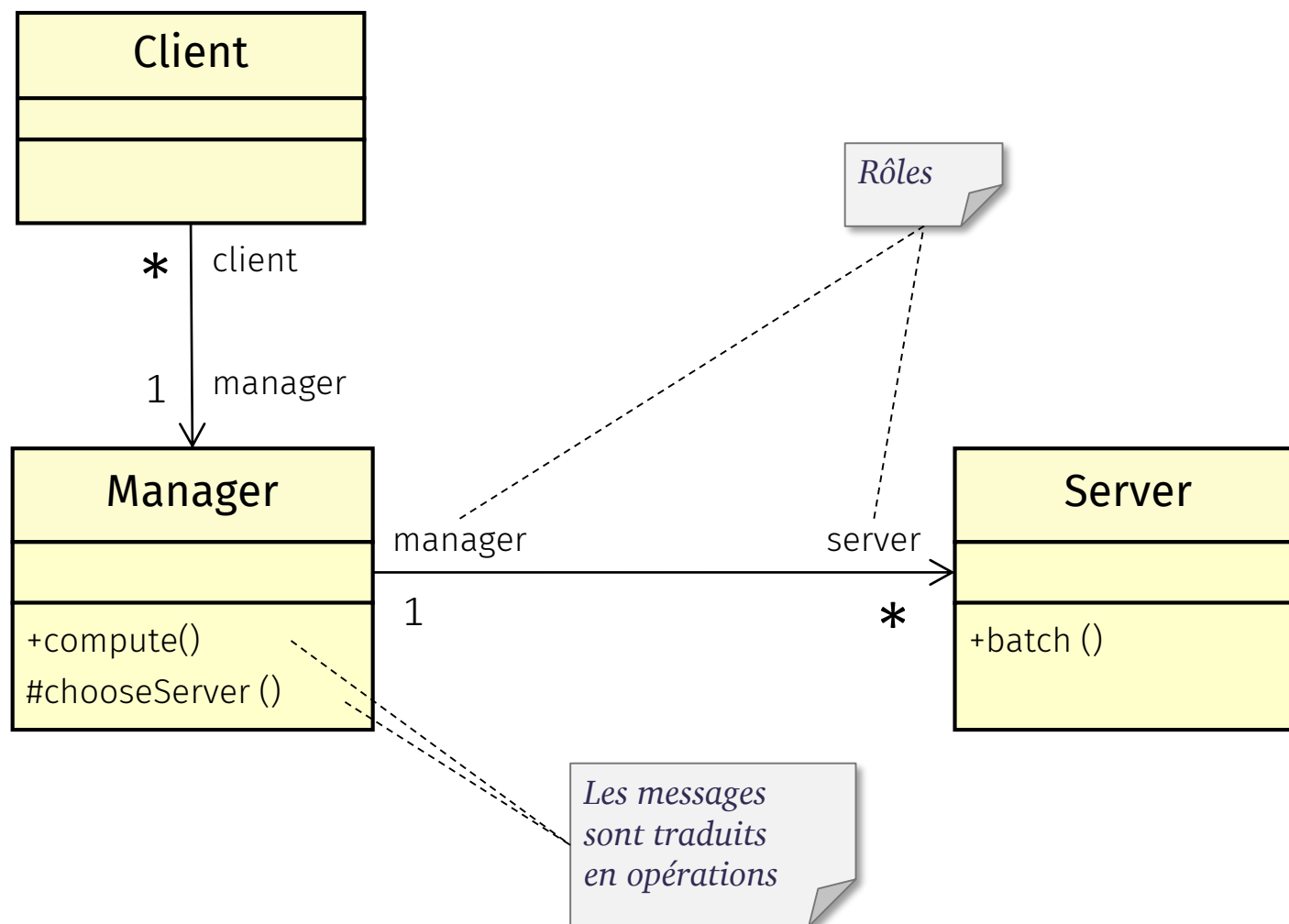
Centre de Saclay — 91191 Gif-sur-Yvette — France

shebli.anvar@cea.fr

- Un objet est une instance de classe
- La classe encapsule sa structure interne
- La classe spécifie une interface à base d'opérations







C++

```

class Client
{
private:
    Manager* manager;
};
    
```

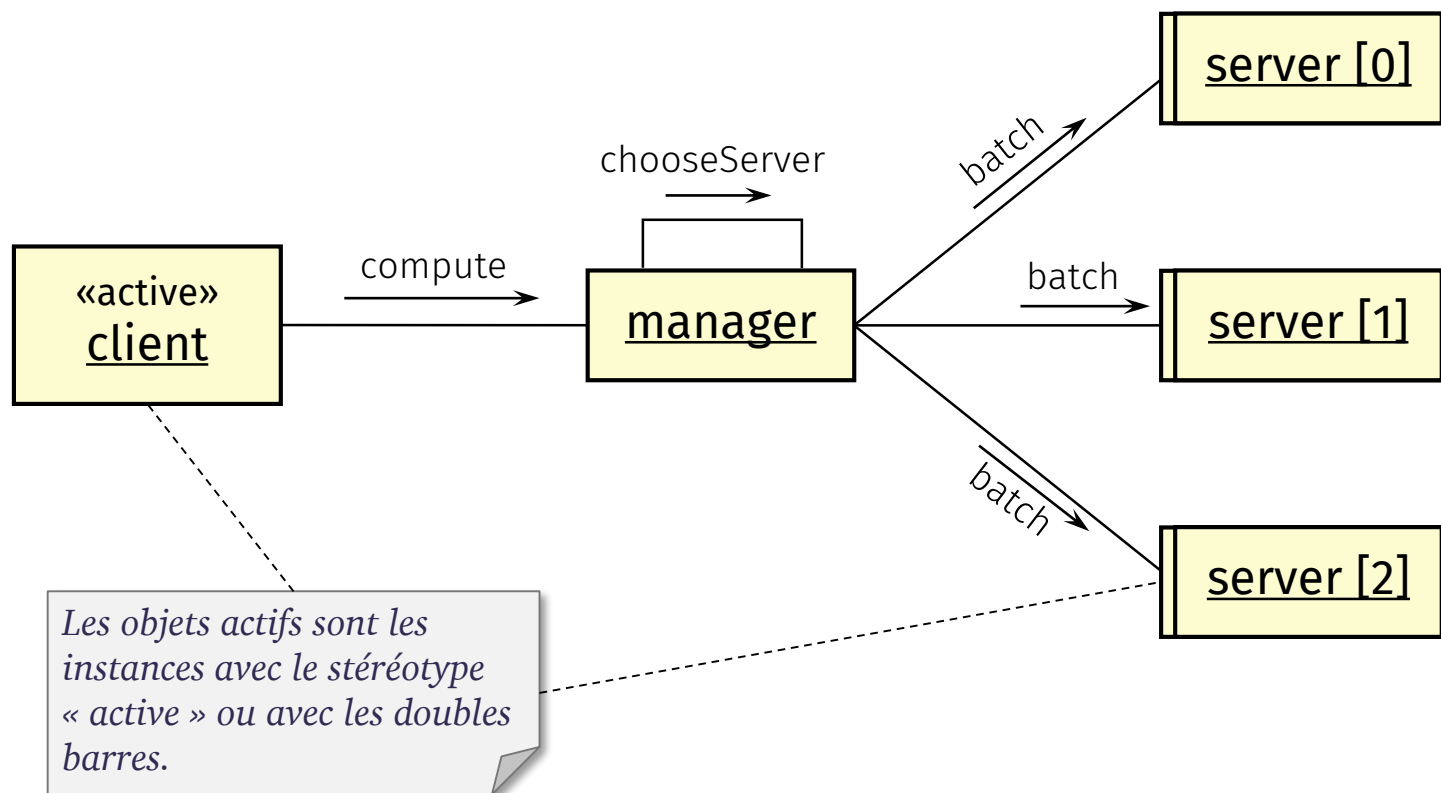
```

class Manager
{
private:
    std::vector<Server*> server;
public:
    double compute(Data input);
protected:
    int chooseServer();
};
    
```

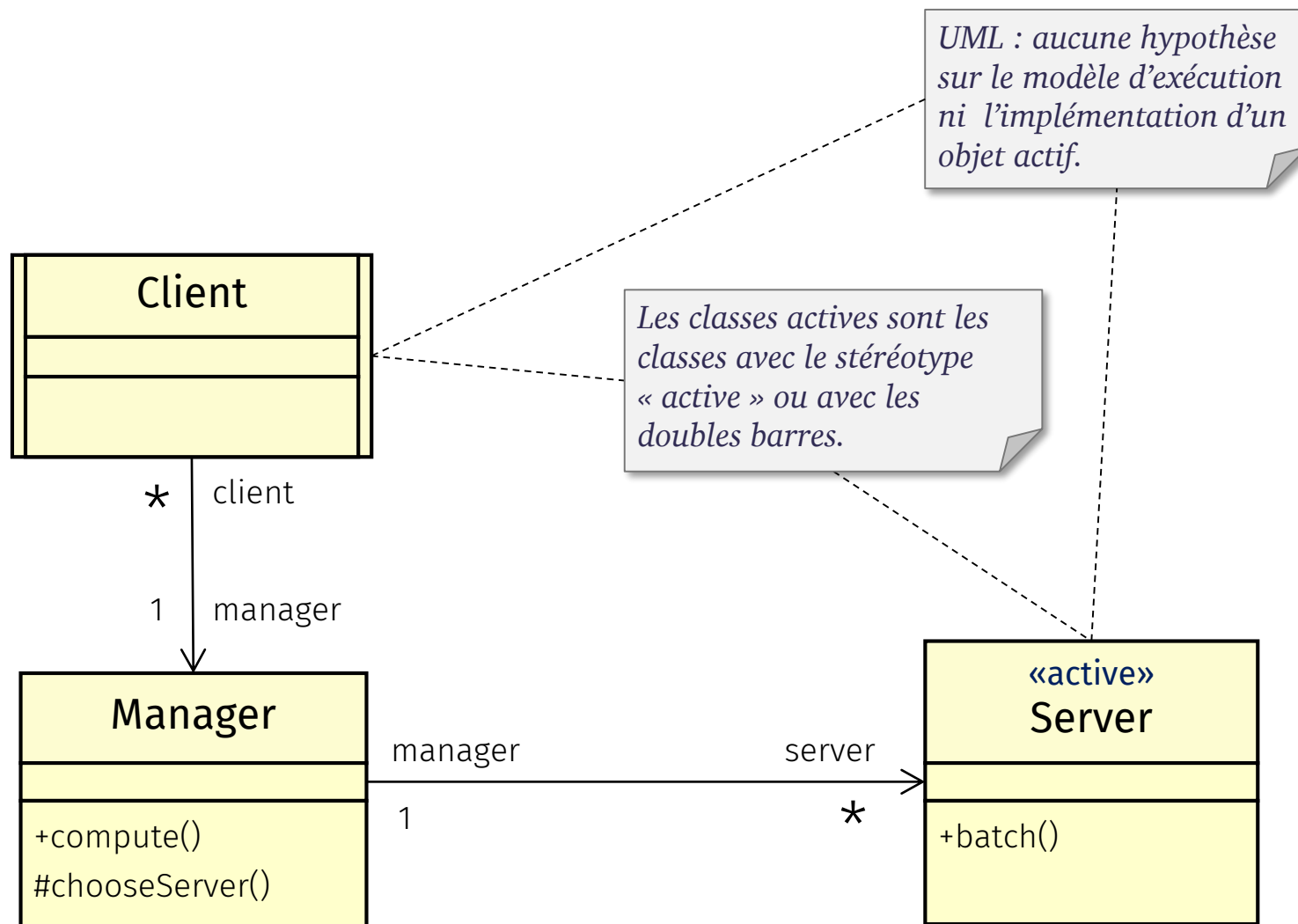
```

class Server
{
public:
    double batch(Data input);
};
    
```

Un « objet actif » est un objet qui encapsule son (ses) propre(s) fil(s) d'exécution



Les objets actifs sont les instances avec le stéréotype « active » ou avec les doubles barres.



■ Tâche (Thread)

- Création, lancement
- Endormissement, suspension
- Arrêt, destruction
- Attente d'arrêt (join)

■ Mutex

- Création, destruction
- Types (simple, récursif...)
- Prise et rendu de jeton
- Rendu automatique

■ Condition

- Association avec Mutex
- Attente et notification
- Timeout

■ Sémaphore

- Binaire, à compte
- Conditions initiales
- Prise et rendu de jetons
- Timeout

■ Communication

- Asynchrone (file d'attente)
- Synchronisation différée
- À distance
- Broadcast

■ Encapsulation

- Objets thread-safe
- Objets actifs

■ Fusion entre objet et tâche

- **Modèle d'exécution**
 - monotâche
 - multitâche (une par opération)
- **Appel d'opération asynchrone**
- **Appel d'opération à distance**

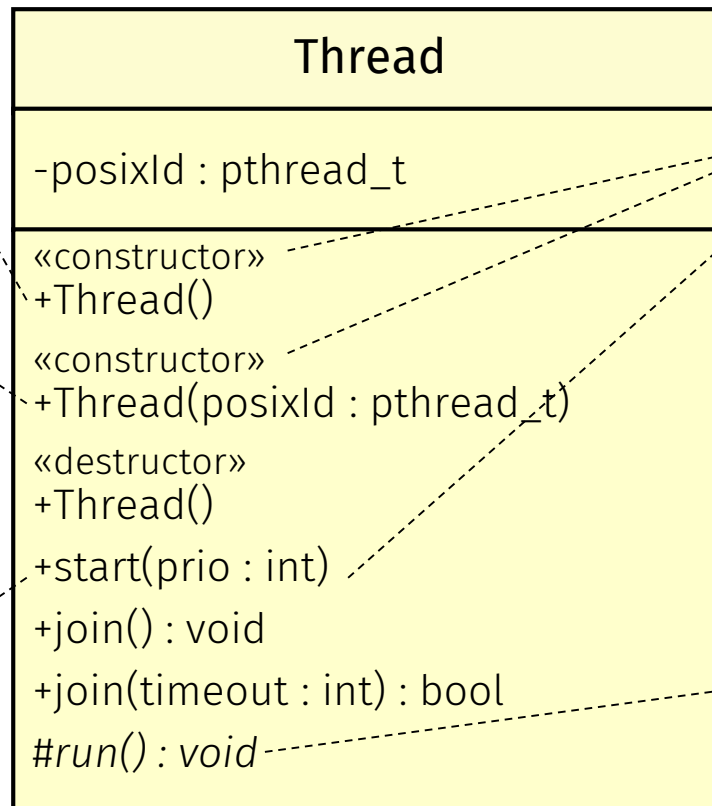
■ En pratique:

- **La classe dérive d'une classe « Thread »**
(ou implémente une interface ad hoc comme l'interface Runnable en Java)
- **Met en œuvre une file d'attente de requêtes d'exécution**

Création d'un objet Thread
non encore démarré.

Création d'un objet Thread
à partir un thread Posix
déjà démarré.

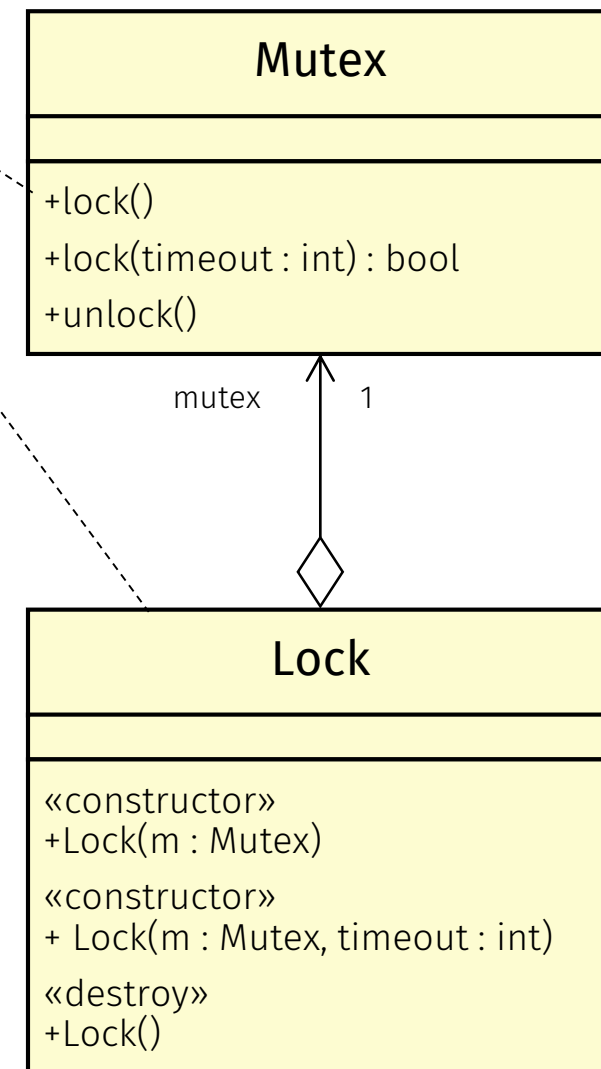
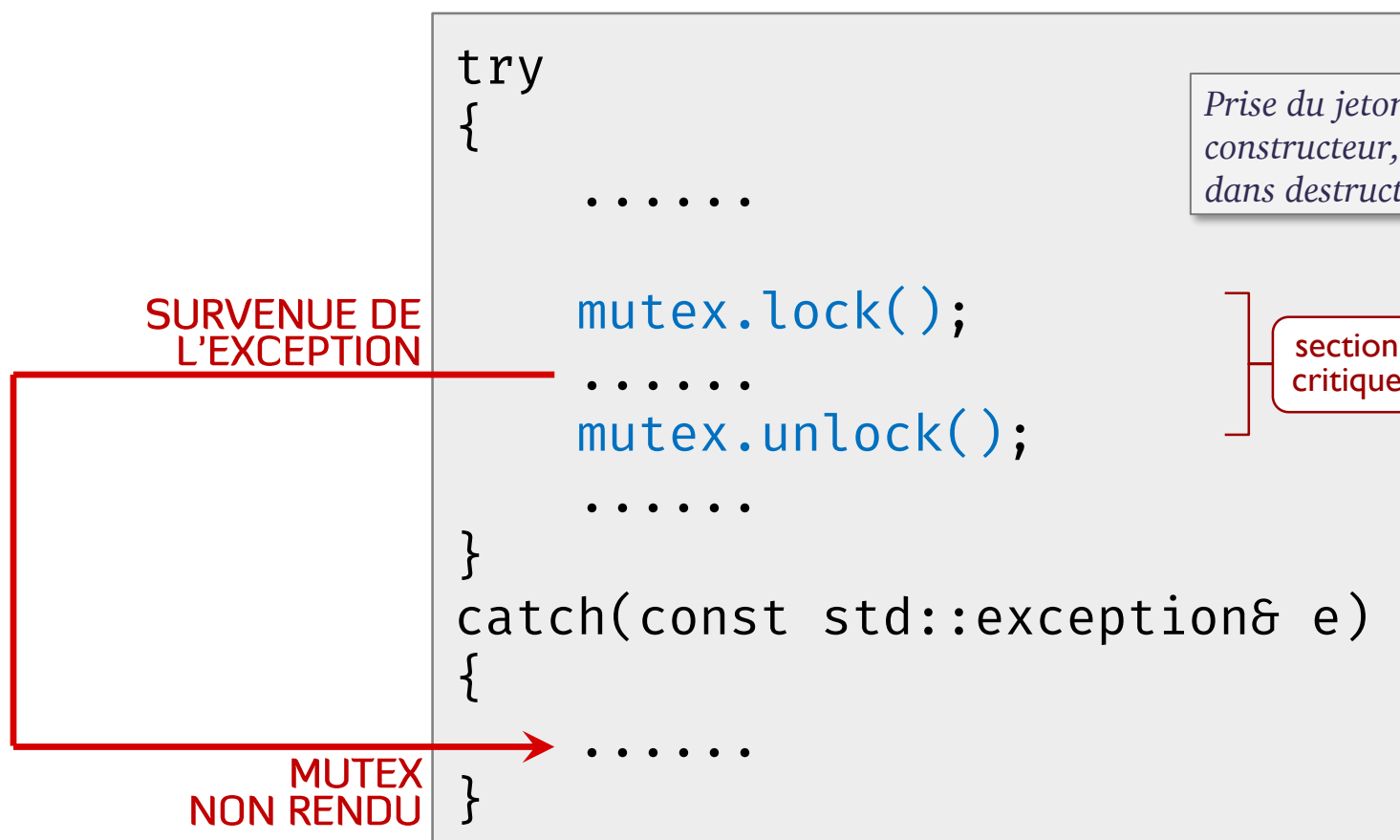
On peut redémarrer un objet
Thread une fois qu'il terminé
son exécution.



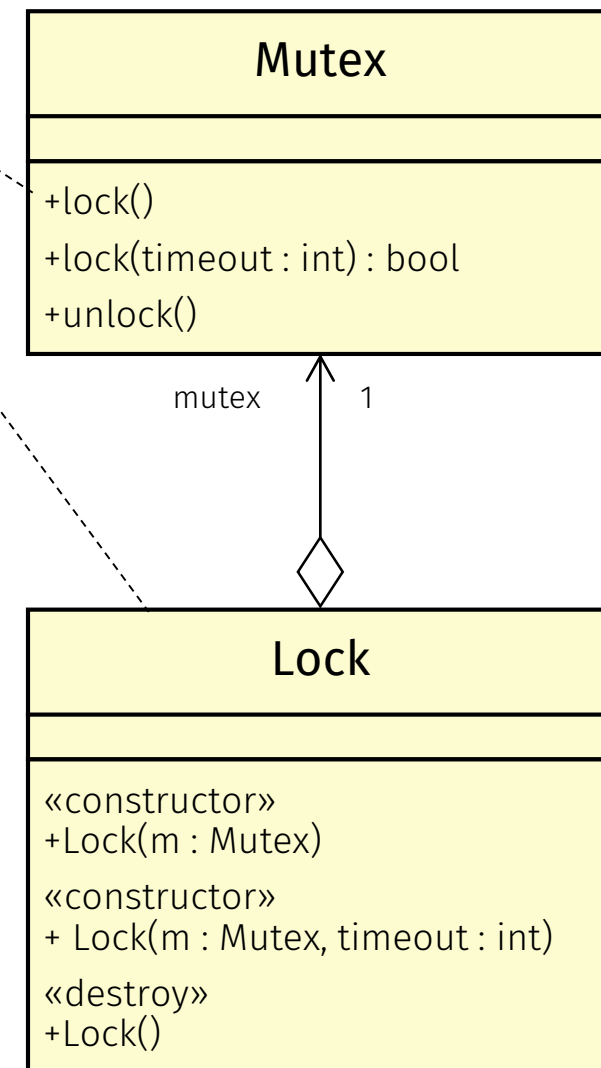
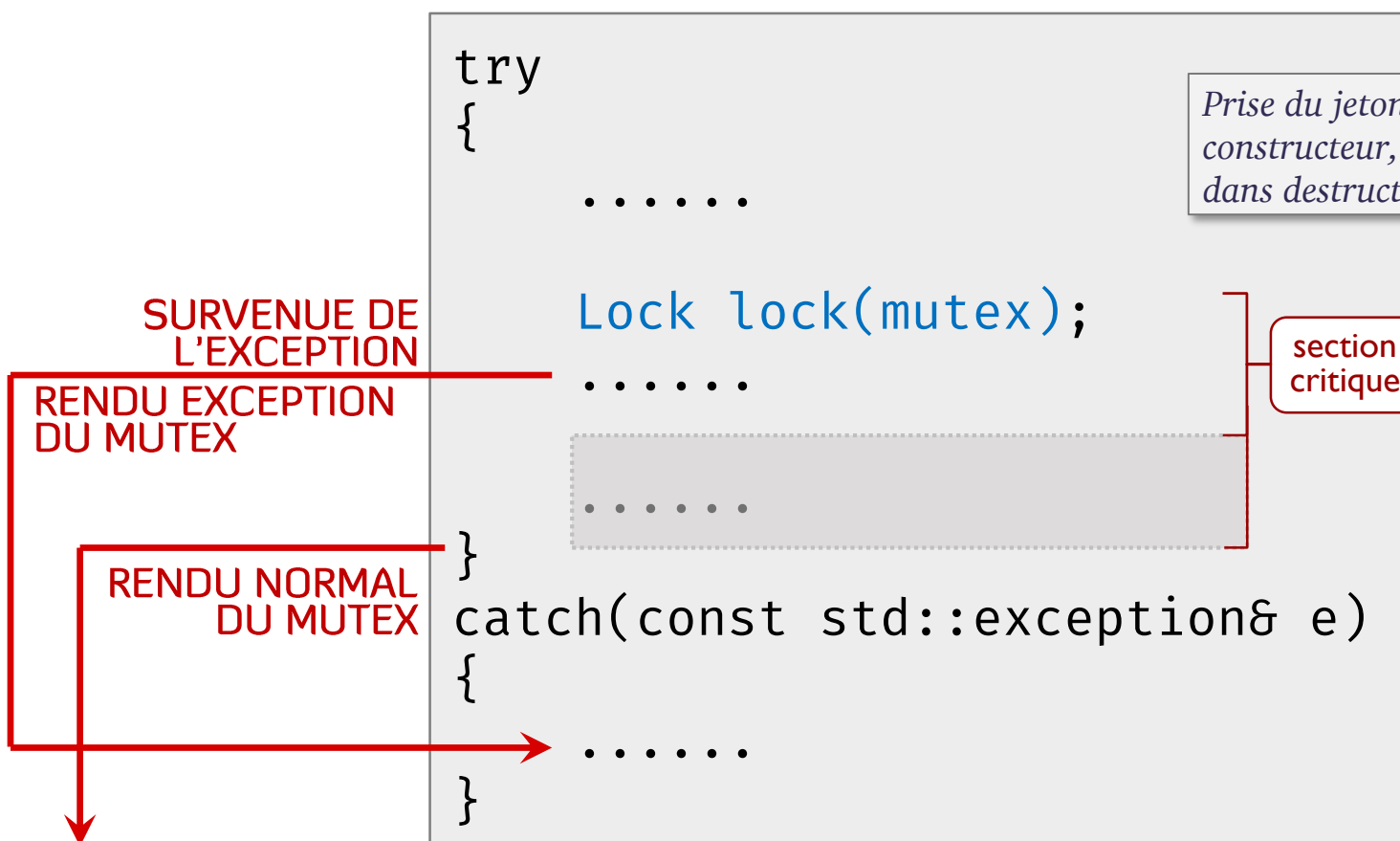
On distingue la création
de l'objet Thread (constructeur) ...
de son démarrage (start).

Questions
Pourquoi la méthode
Thread::run() doit être :
- protected ?
- virtuelle pure ?

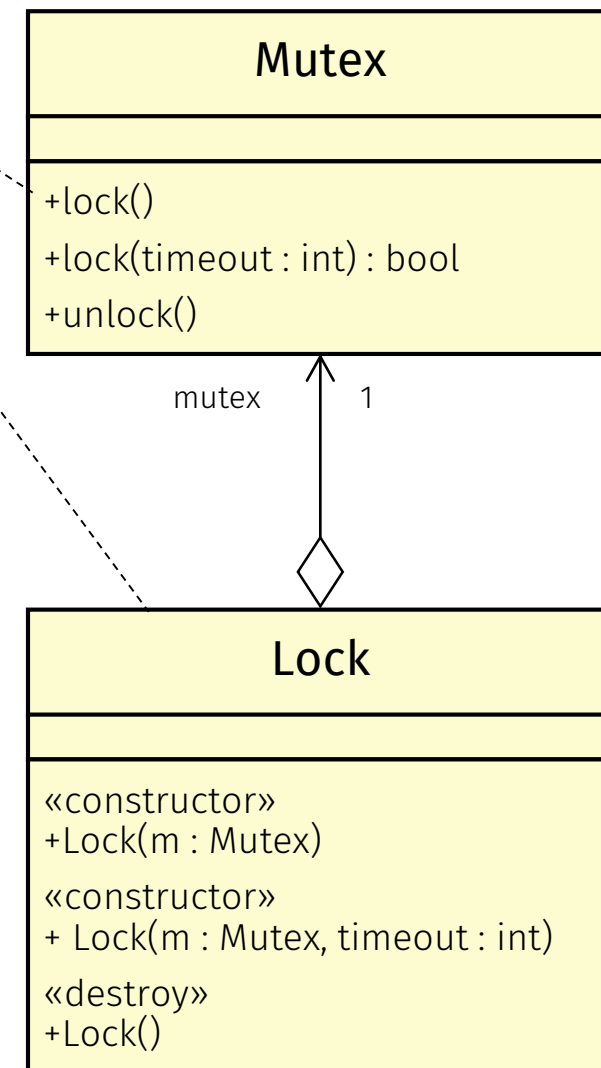
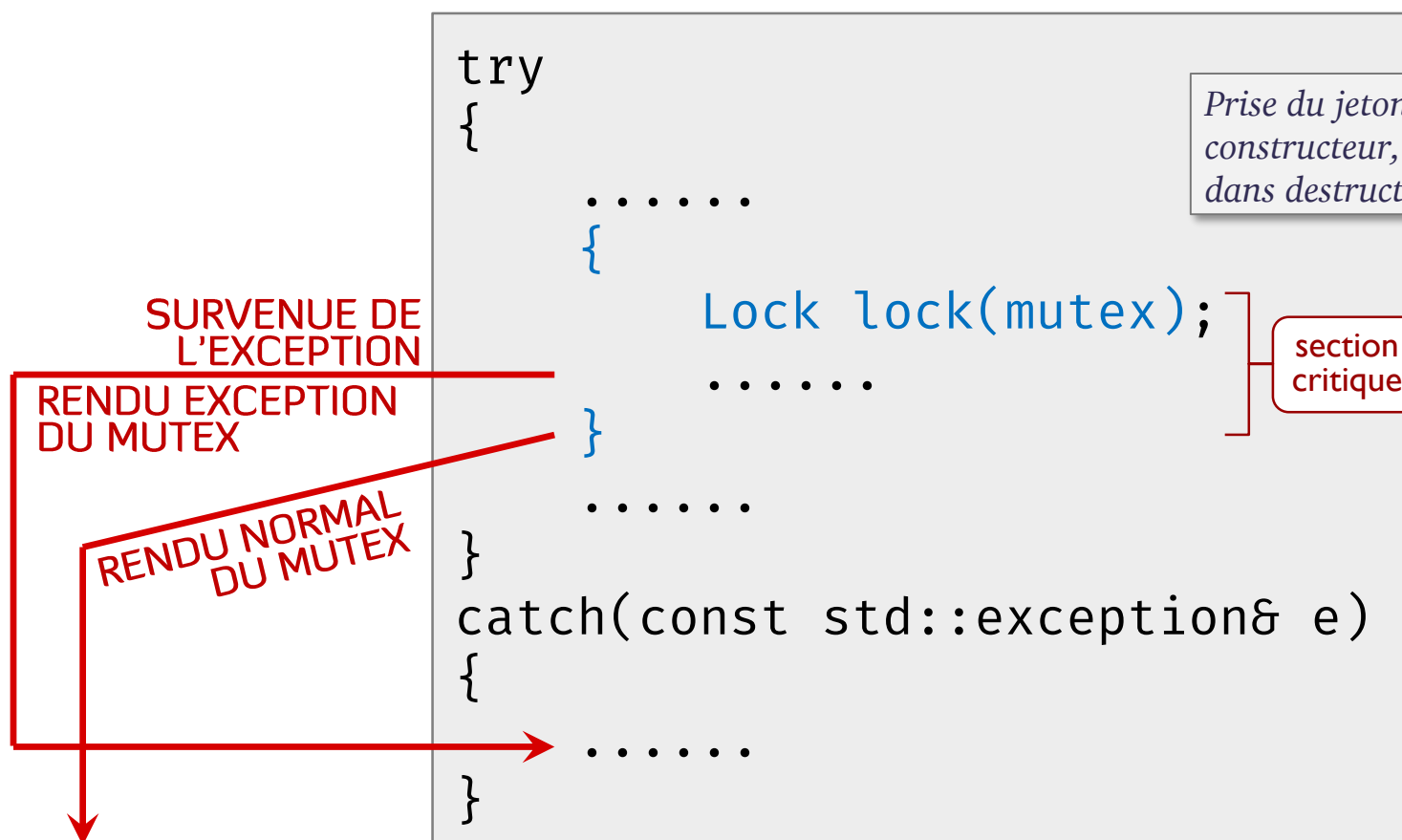
Le problème du rendu de mutex lors des exceptions



Le problème du rendu de mutex lors des exceptions



Le problème du rendu de mutex lors des exceptions



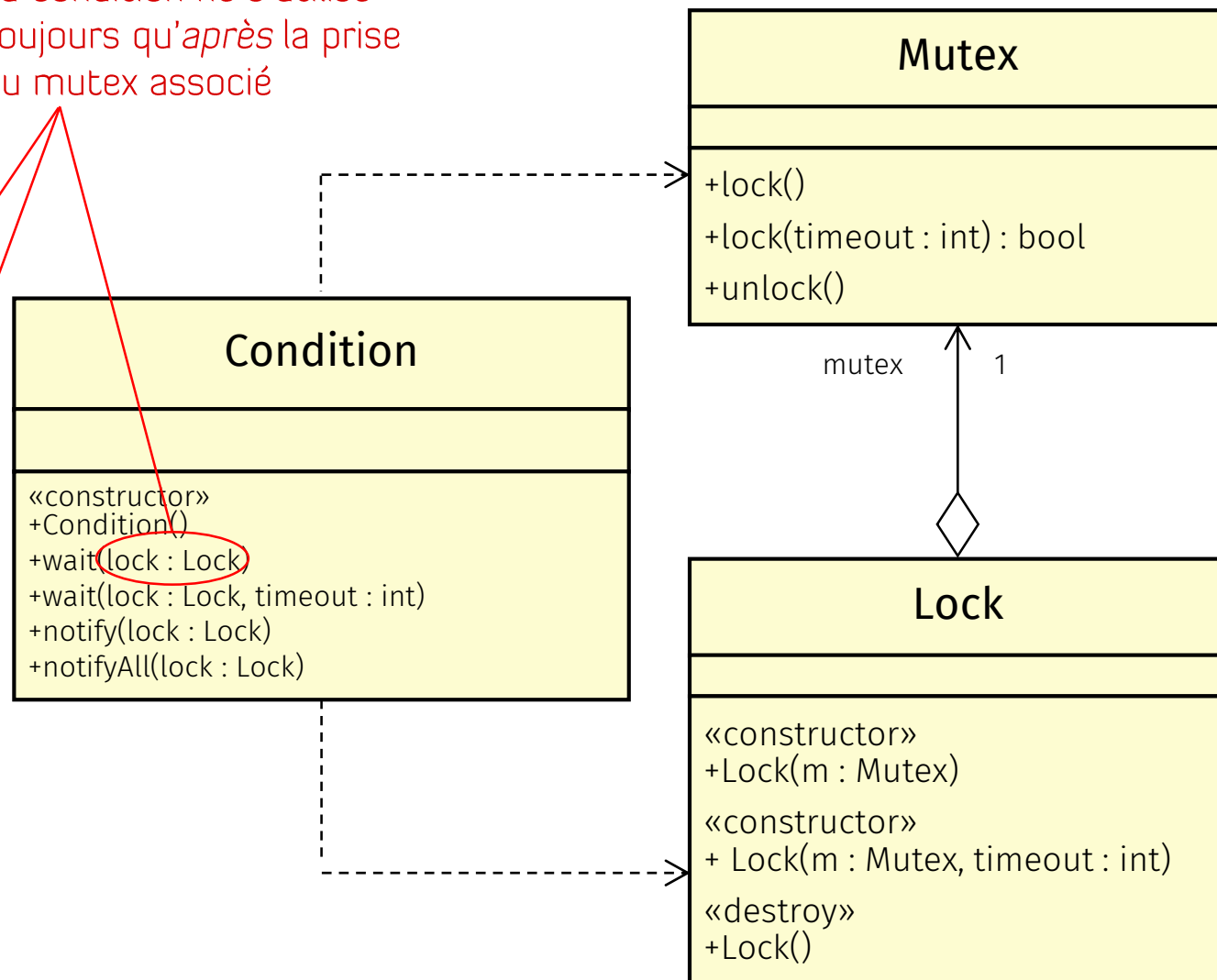
tâche A

```
void waitStop (
    volatile int* pCommand,
    Mutex* mtx,
    Condition* cnd
)
{
    Lock lock(mutex);
    while (*pCommand != STOP)
    {
        cnd->wait(&lock);
    }
}
```

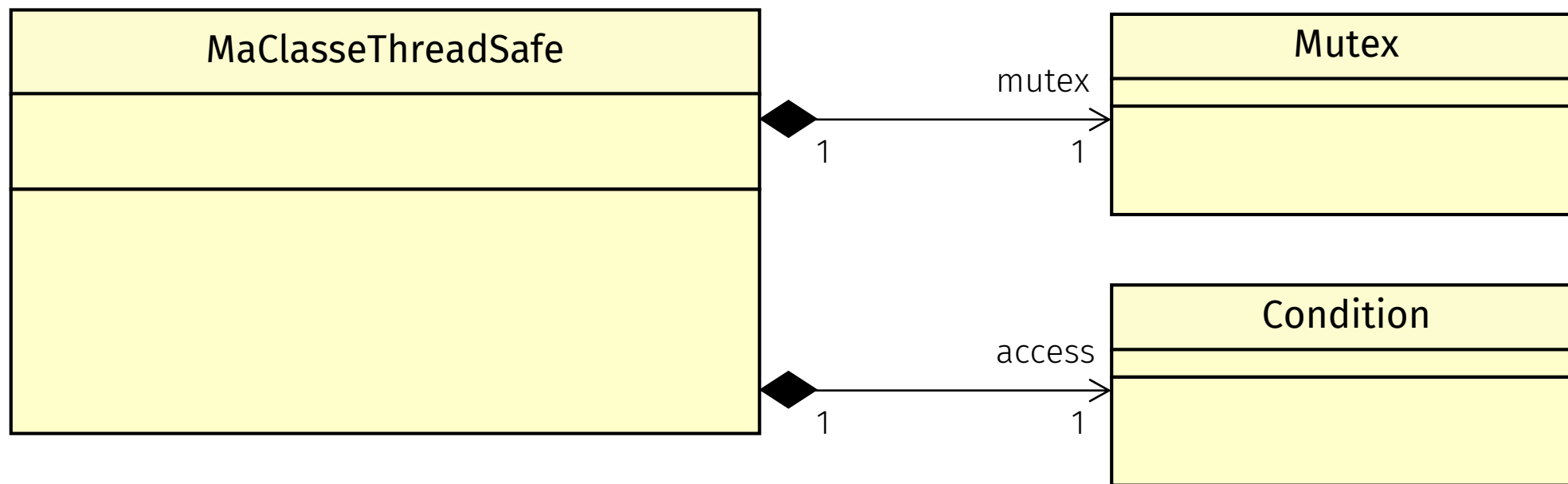
tâche B

```
void doStop (
    volatile int* pCommand,
    Mutex* mtx,
    Condition* cnd
)
{
    Lock lock(mutex);
    *pCommand = STOP;
    cnd->notify(&lock);
}
```

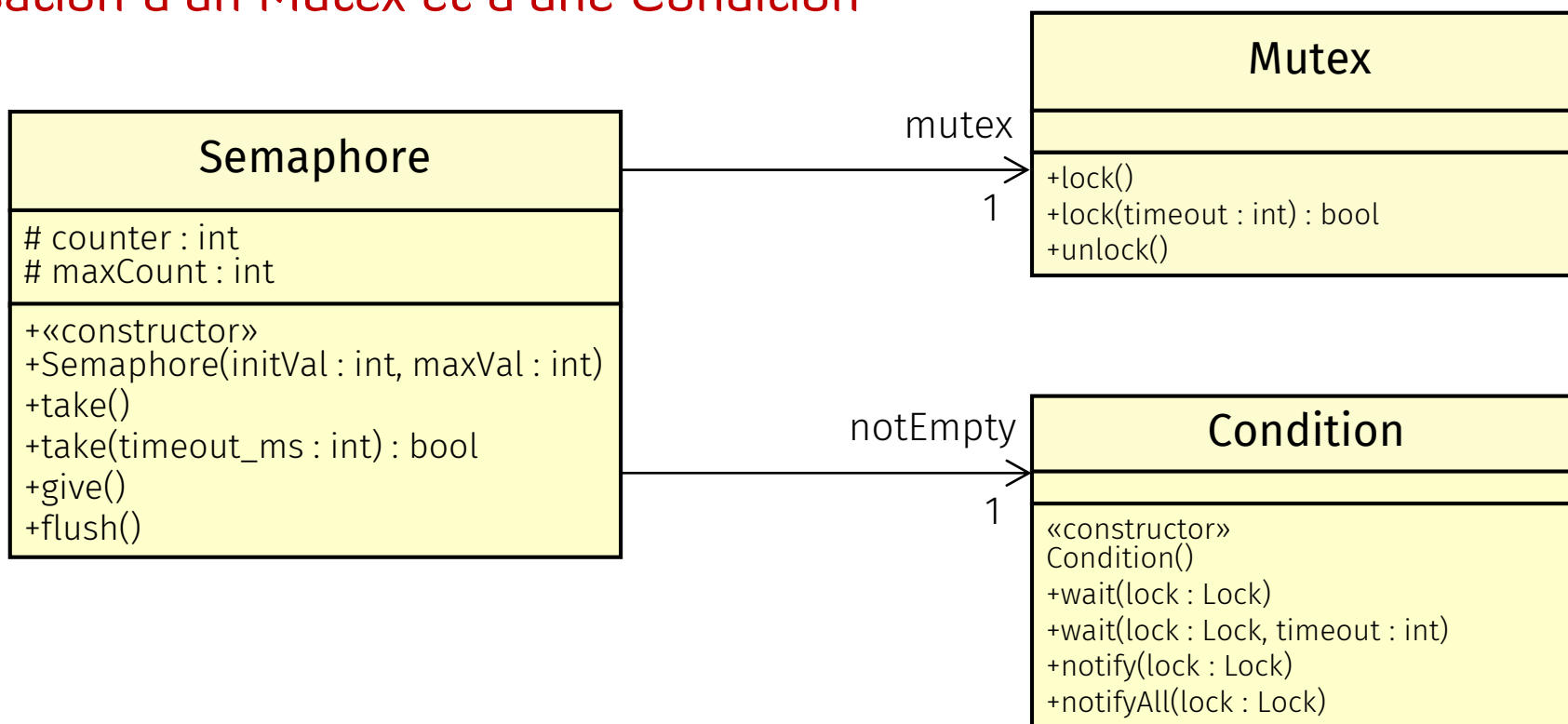
La condition ne s'utilise
toujours qu'après la prise
du mutex associé

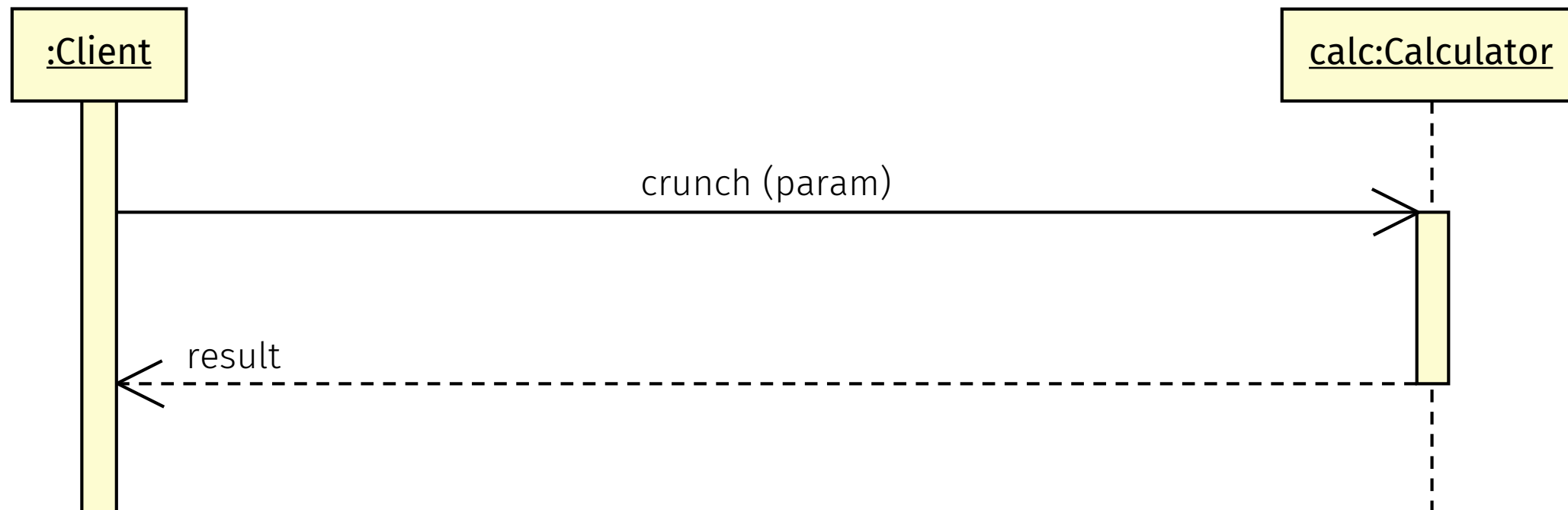


- Toutes les opérations d'une classe thread-safe doivent accéder ou modifier l'état de l'objet en garantissant les appels concurrents.



- Un sémaphore est un compteur de jetons
Lorsqu'il est vide, la demande de jeton bloque la tâche
Déblocage : une autre tâche fournit un jeton
- Mécanisme de blocage déblocage ?
 - Variable partagée : compteur de jetons
 - Utilisation d'un Mutex et d'une Condition

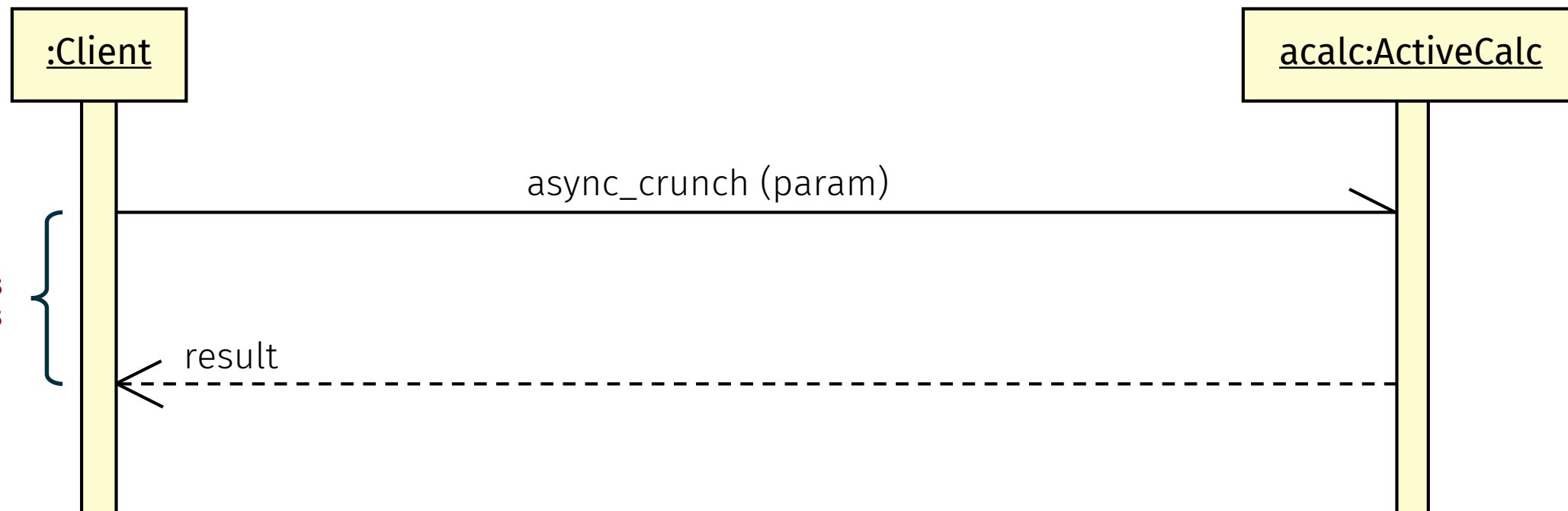




```

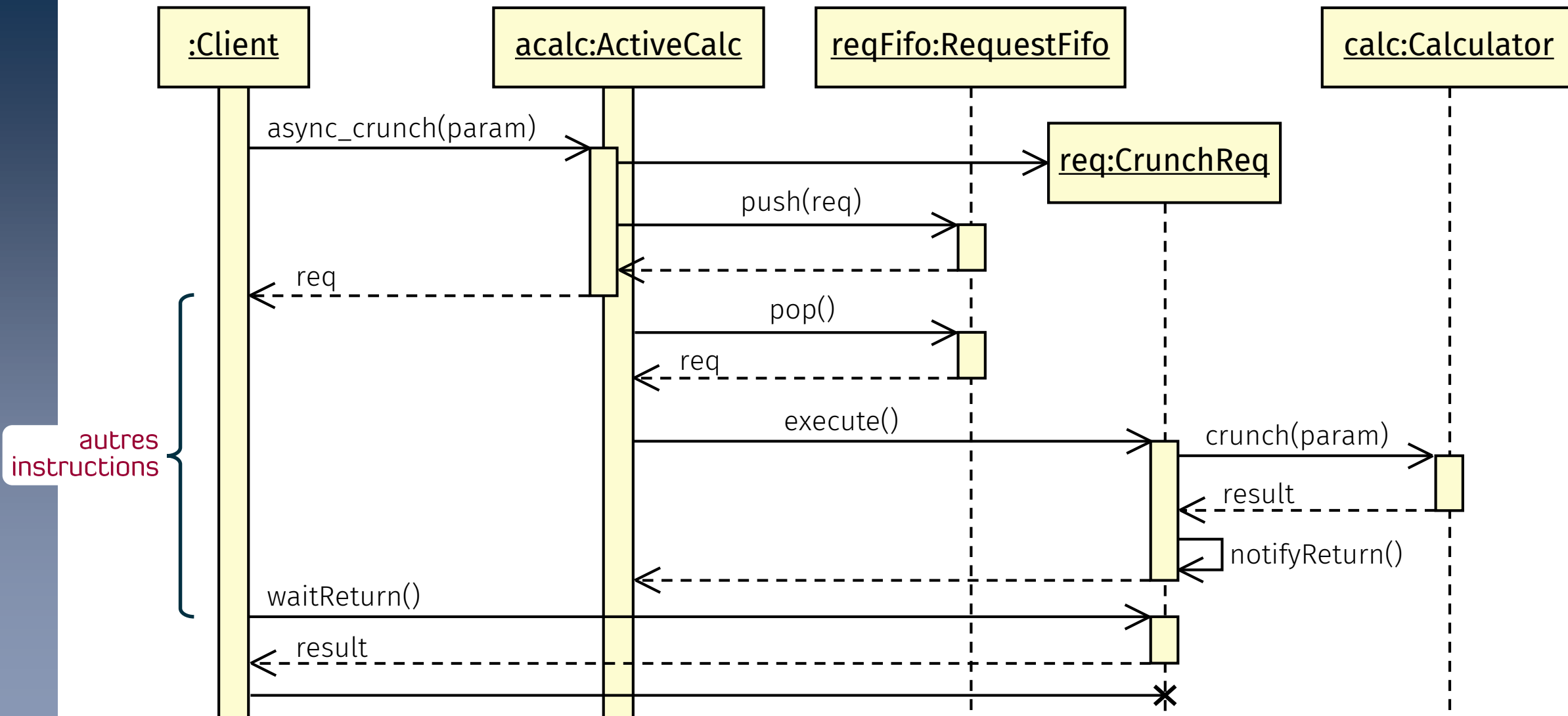
Client::run(Calculator* calc)
{
    int param = 10;

    double result = calc->crunch(param);
}
  
```

```

Client::run(Calculator* calc)
{
    int param = 10;
    Request* req = acalc->async_crunch(param);
    // ... Autres instructions
    double result = req->waitReturn();
}
    
```



```
void Client::main(ActiveCalculator* acalc) {
    CrunchReq* req = acalc->async_crunch(10); // requête
    // ..... // Autres instructions
    double result = req->waitReturn(); // Attente result
}
```

```
CrunchReq* ActiveCalc::async_crunch(double param) {
    CrunchReq* req = new CrunchReq(param); // Création de la requête d'exécution
    reqFifo.push(req); // Envoi de la requête
    return req; // Transmission au client
}
```

```
void ActiveCalc::run() {
    while(true) {
        CrunchReq* req = reqFifo.pop(); // Réception de la requête
        req->execute(); // Exécution de la requête
    }
}
```

```
double CrunchReq::waitReturn() {
    returnSema.take(); // attente de la fin d'exécution du calcul
    return result; // renvoi du résultat de calcul à l'appelant
}
```

```
void CrunchReq::execute() {
    result = calc->crunch(param); // exécution effective du calcul
    returnSema.give(); // notification du sémaphore « fin de calcul »
}
```

