Multitâche & objets

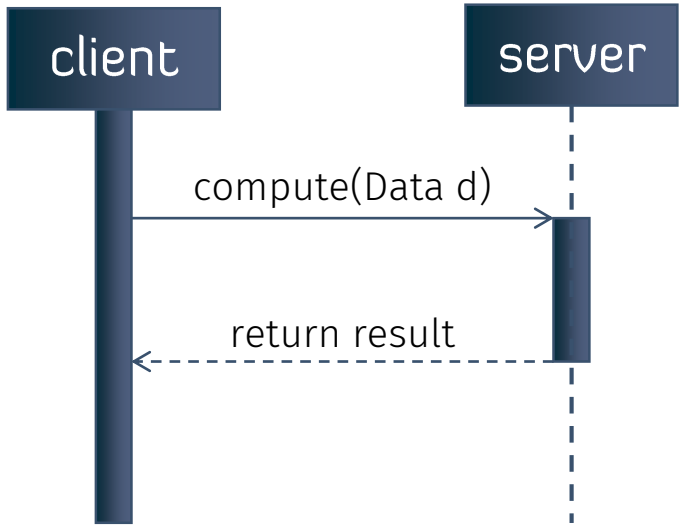
Modélisation objet des paradigmes multitâches

© Shebli Anvar
CEA — Institut de recherche sur les lois fondamentales de l'Univers
Centre de Saclay — 91191 Gif-sur-Yvette — France
shebli.anvar@cea.fr

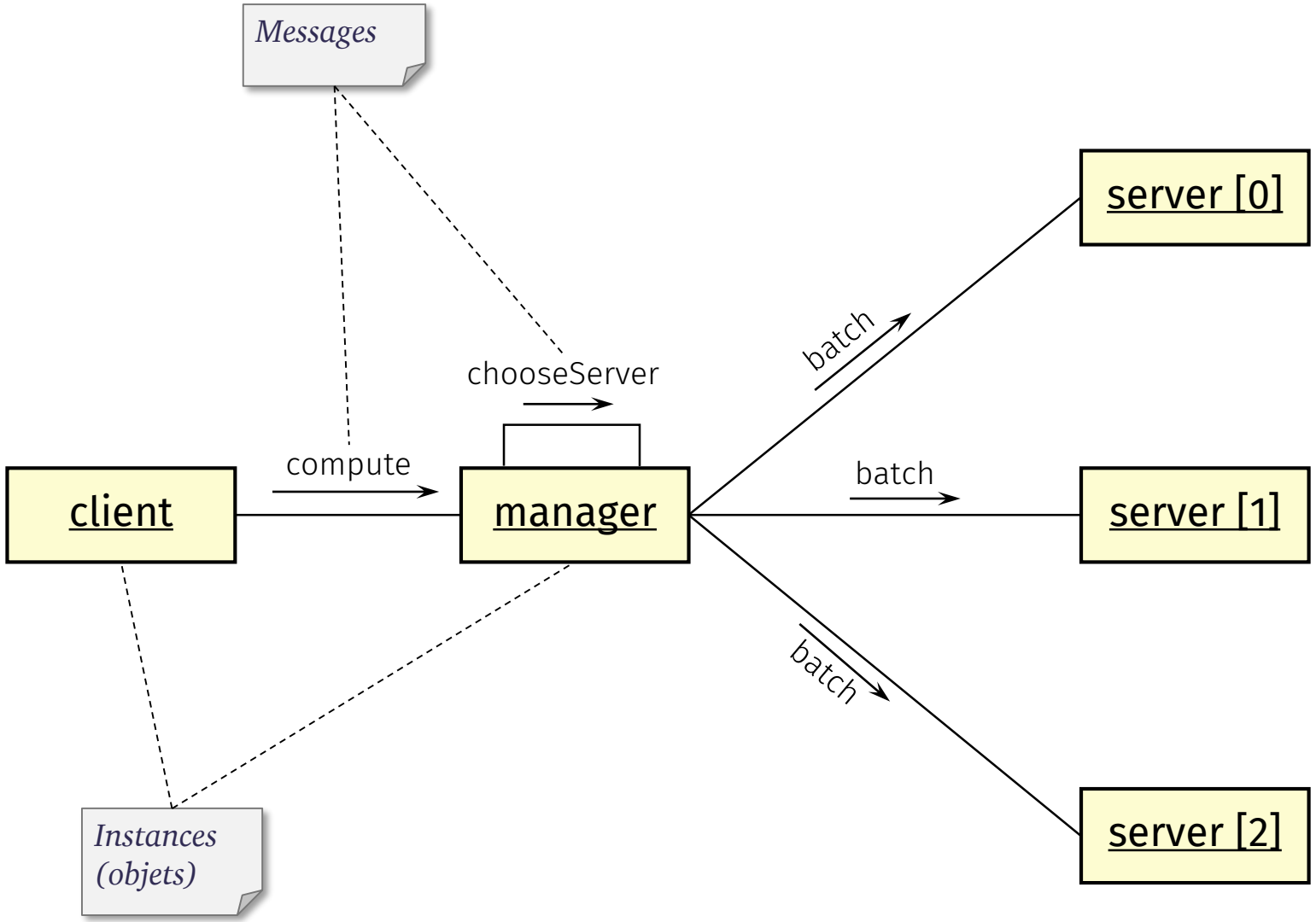
Communication inter-objets

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

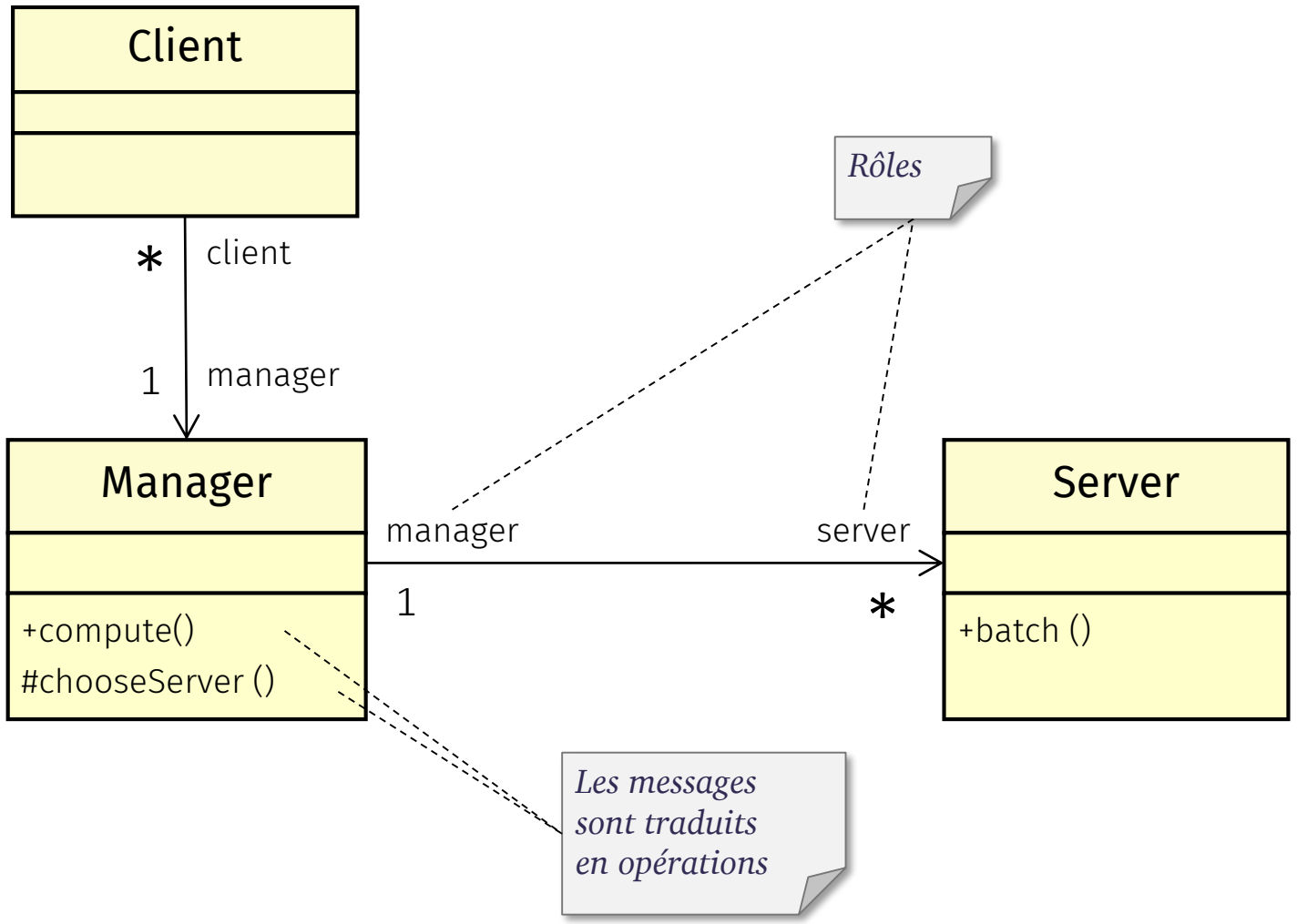
Communication inter-objets
- appels d'opérations
- données échangées = paramètres



Collaboration entre instances



Abstraction des classes



C++

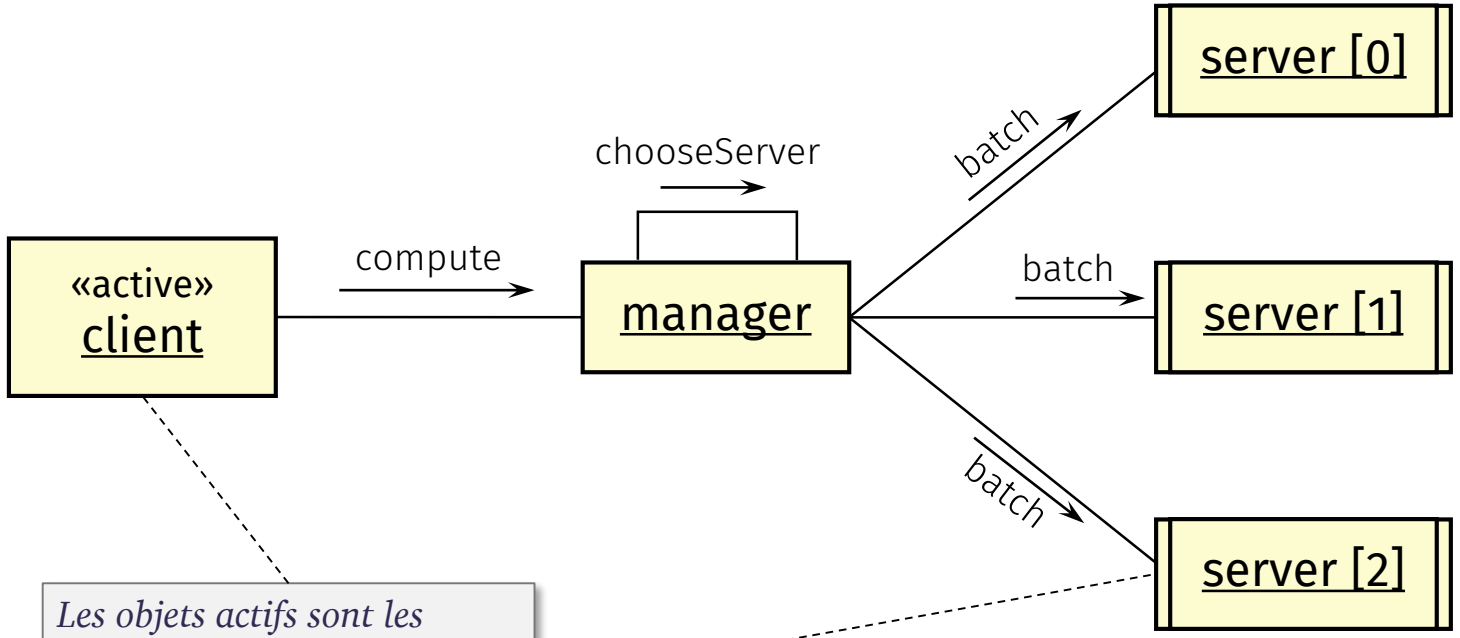
```
class Client
{
private:
    Manager* manager;
};
```

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

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

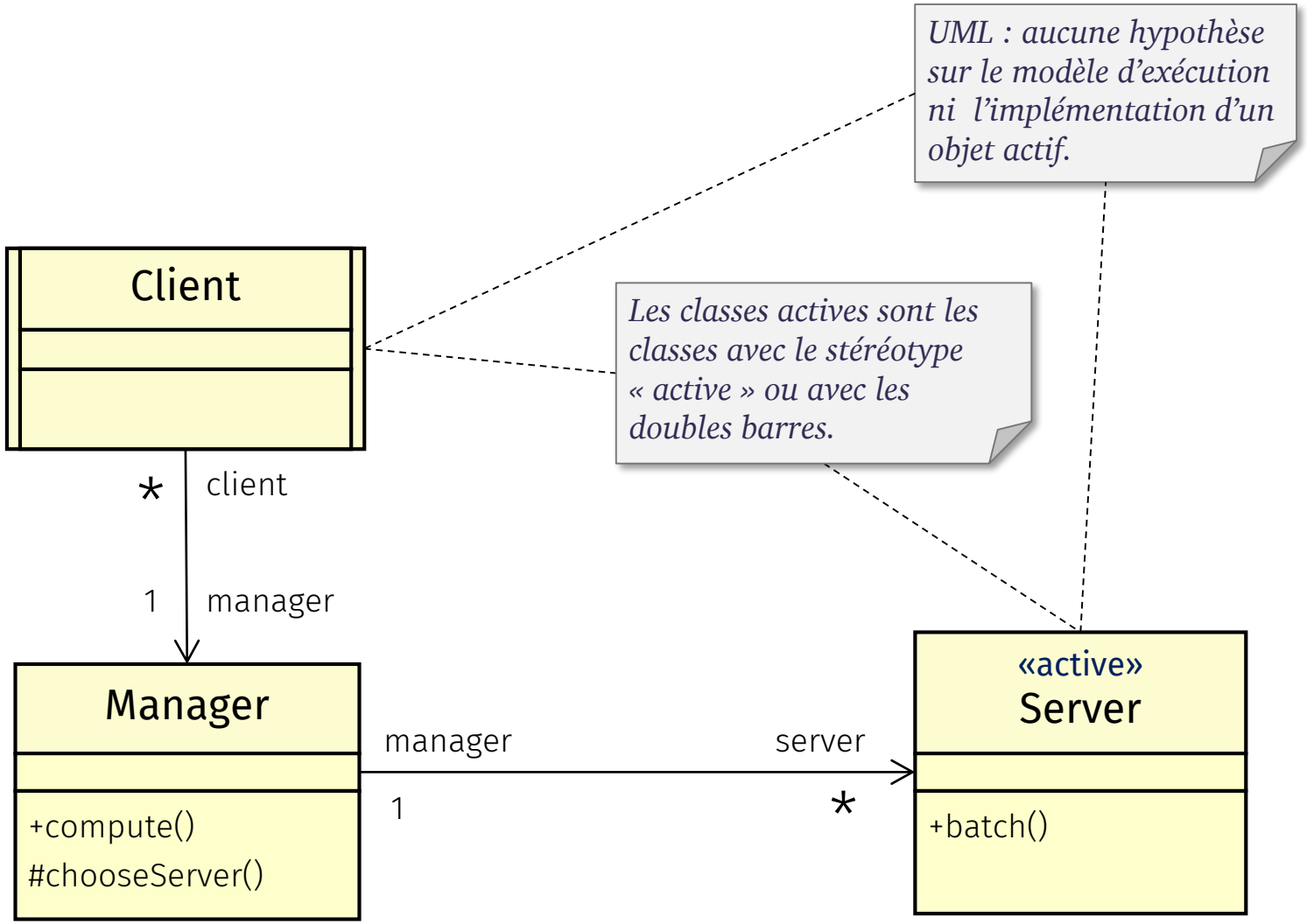
Objets actifs

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.

Abstraction des classes objets actifs



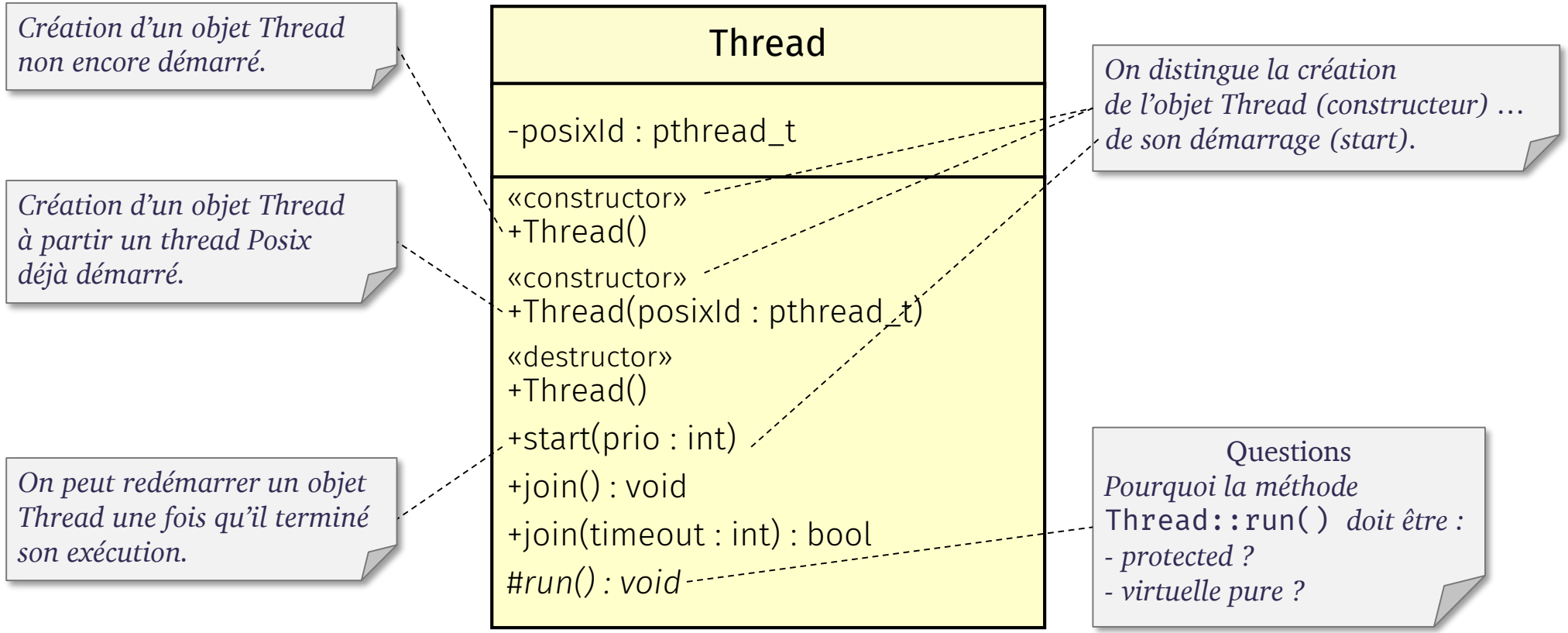
Notions à transposer

- 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

La notion d'objet actif

- 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

Exemple de classe Thread



Les classes Mutex et Lock

Le problème du rendu de mutex lors des exceptions

```
try
{
    ......
    mutex.lock();
    ......
    mutex.unlock();
    ......
}
catch(const std::exception& e)
{
    ......
}
```

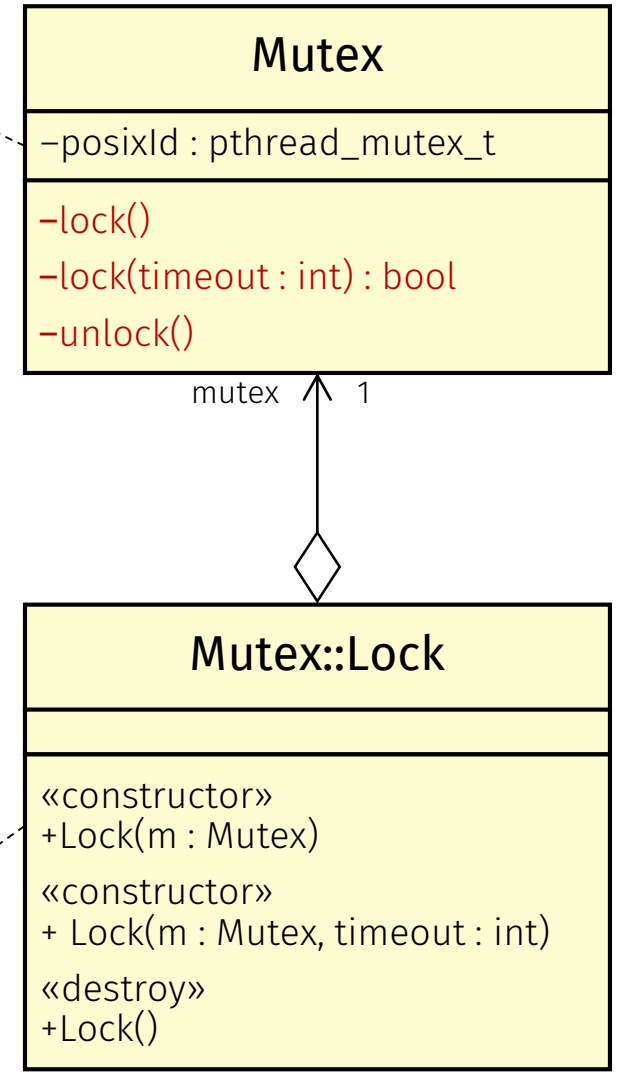
SURVENUE DE L'EXCEPTION

MUTEX NON RENDU

section critique

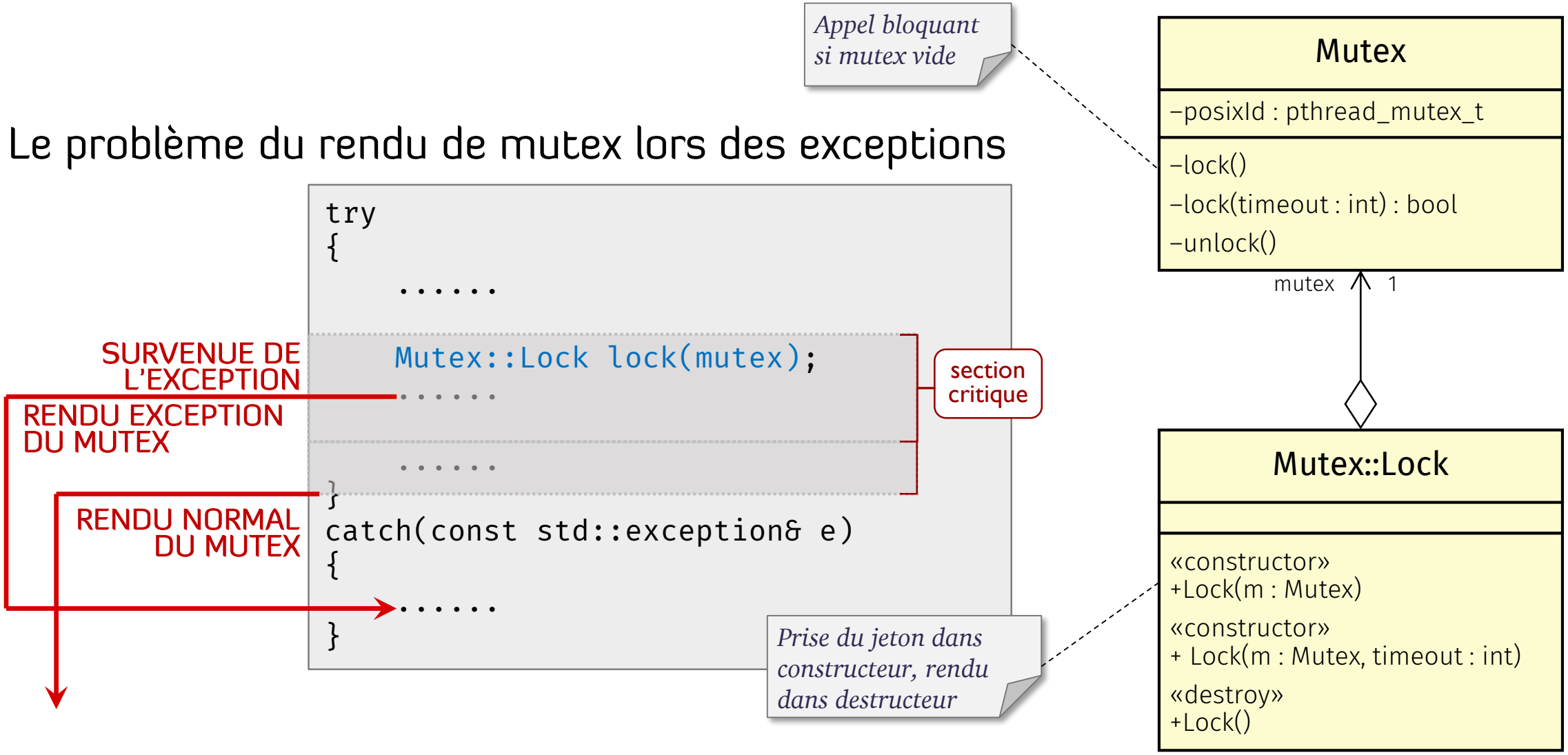
Appel bloquant si mutex vide

Prise du jeton dans constructeur, rendu dans destructeur



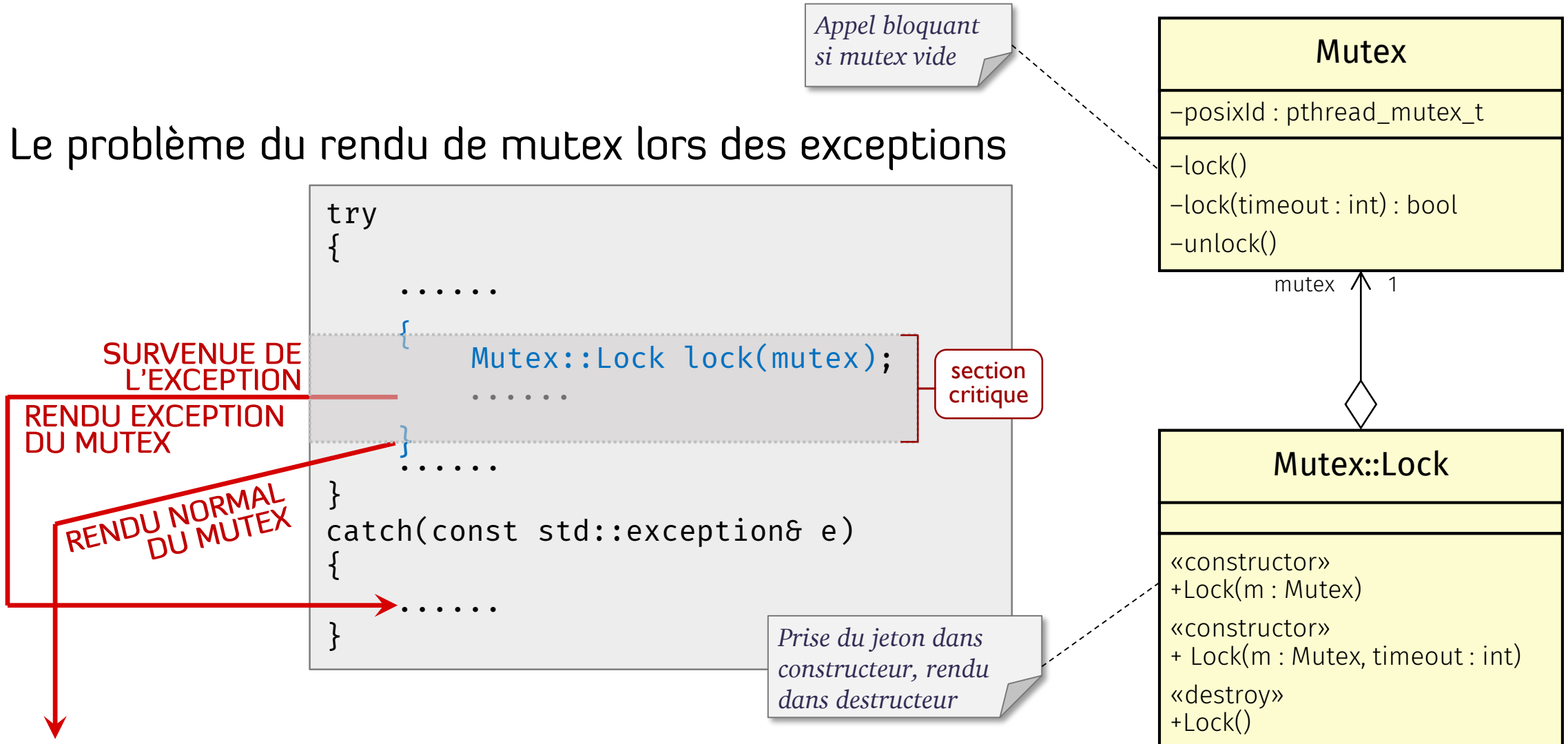
Les classes Mutex et Lock

Le problème du rendu de mutex lors des exceptions



Les classes Mutex et Lock

Le problème du rendu de mutex lors des exceptions



Classe Lock : principe RAII

- Resource Acquisition Is Initialization
  - encapsulate each resource into a class, where:
    - the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
    - the destructor releases the resource and never throws exceptions;
  - always use the resource via an instance of a RAII-class that either:
    - has automatic storage duration or temporary lifetime itself, or
    - has lifetime that is bounded by the lifetime of an automatic or temporary object.

Classes with `open()/close()`, `lock()/unlock()`, or `init()/copyFrom()/destroy()` member functions are typical examples of ***non-RAII*** *classes.*

Intégration de la condition

L'architecture des classes doit imposer les contraintes suivantes :

- un mutex et une condition doivent toujours être utilisés ensemble
- le couple {mutex,condition} doit être partagé entre toutes les tâches qui en ont besoin
- un même mutex ne doit pas être utilisé avec plusieurs conditions : un même mutex ne peut participer qu'à un seul couple {mutex,condition}
- on ne peut utiliser une condition (i.e. faire un wait() ou un notify() dessus) que si le mutex a été verrouillé au préalable

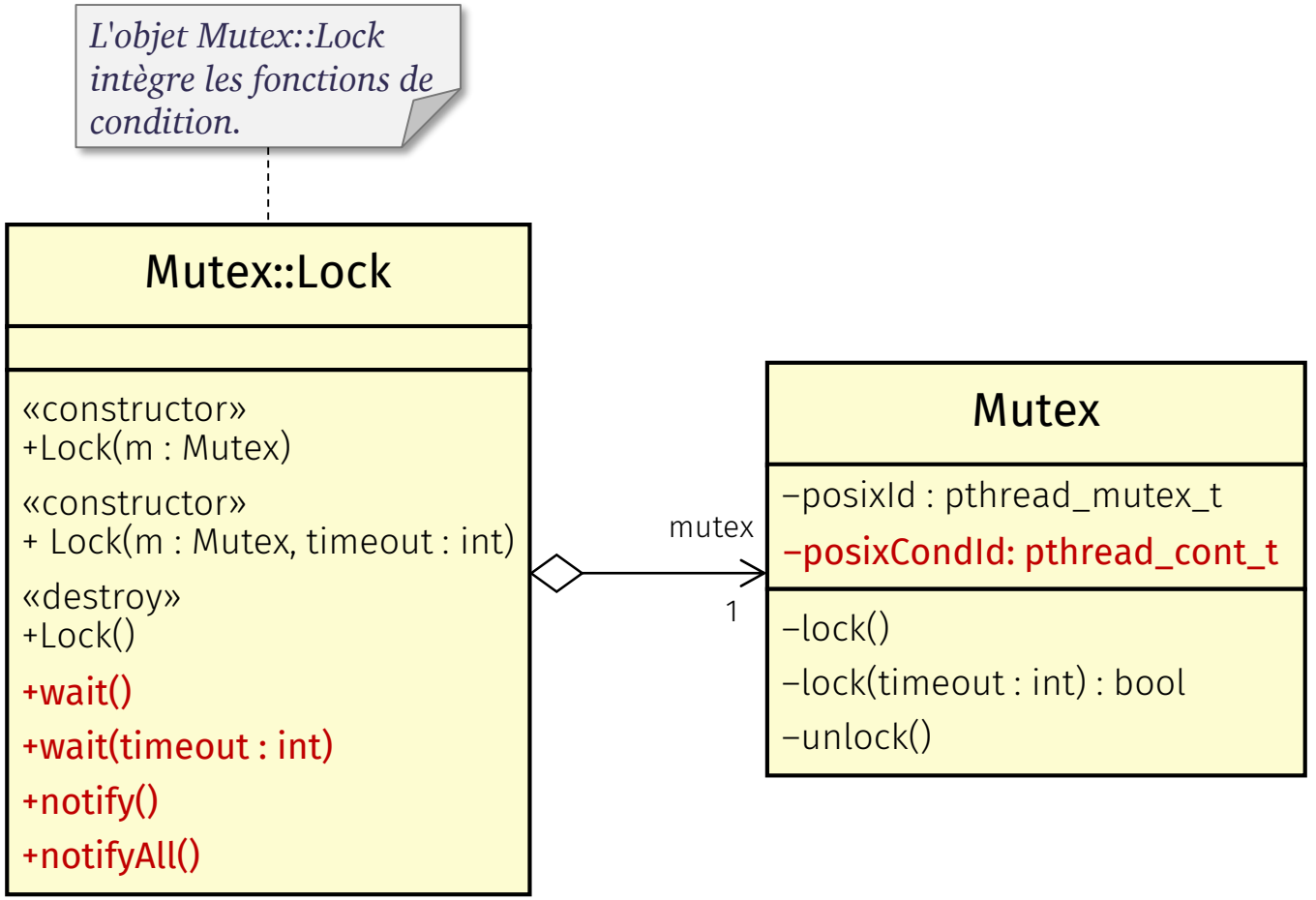
tâche A

```
void waitStop (
     volatile int* pCommand,
     Mutex* mtx
)
{
    Lock lock(mutex);
    while(*pCommand != STOP)
    {
        lock.wait();
    }
}
```

tâche B

```
void doStop (
    volatile int* pCommand,
    Mutex* mtx
)
{
    Lock lock(mutex);
    *pCommand = STOP;
    lock.notify();
}
```

Intégration de la condition



tâche A

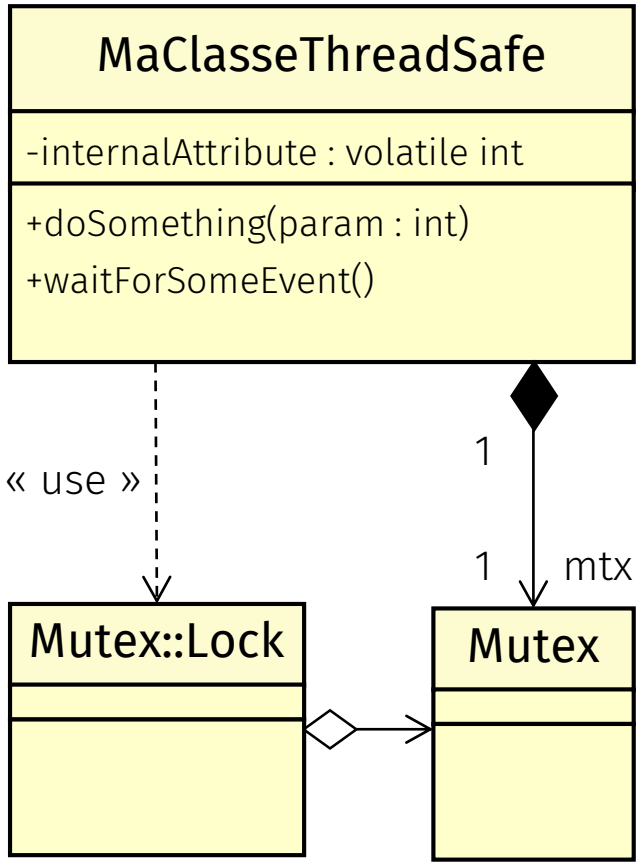
```
void waitStop (
     volatile int* pCommand,
     Mutex* mtx
)
{
    Lock lock(mutex);
    while(*pCommand != STOP)
    {
        lock.wait();
    }
}
```

tâche B

```
void doStop (
    volatile int* pCommand,
    Mutex* mtx
)
{
    Lock lock(mutex);
    *pCommand = STOP;
    lock.notify();
}
```

Objet thread-safe

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



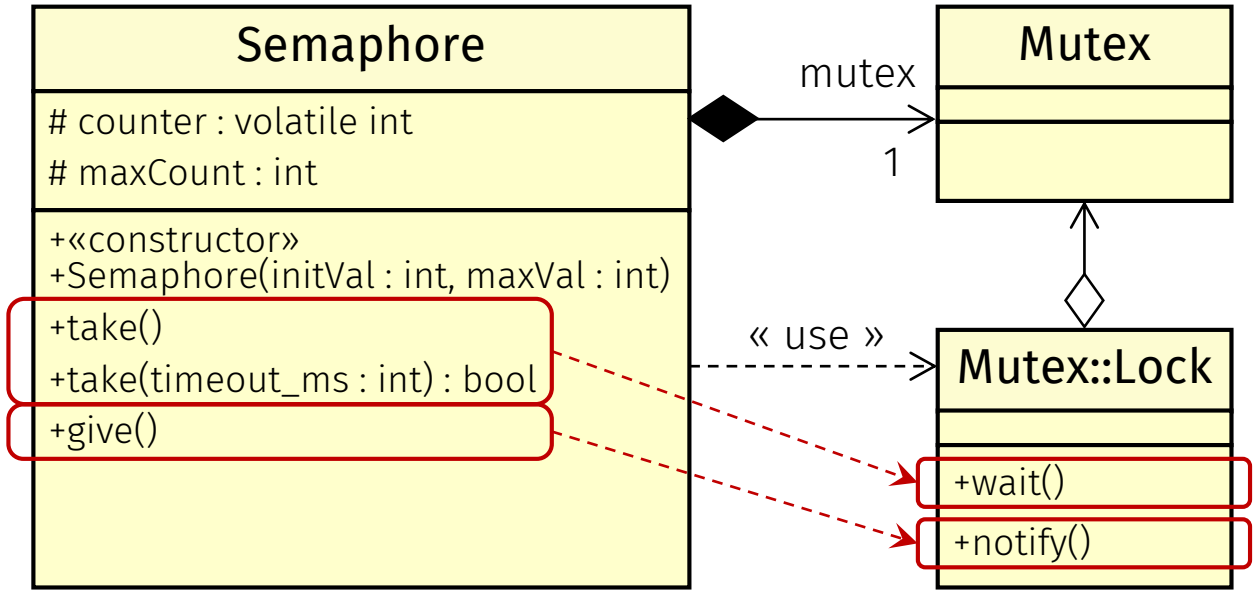

```
void MaClasseThreadSafe::doSomething(int param)
{
    Mutex::Lock lock(mtx);
    internalAttribute += param;
    lock.notifyAll();
}
```

```
void MaClasseThreadSafe::waitForSomeEvent()
{
    Mutex::Lock lock(mtx);
    while(internalAttribute != 42)
    {
        lock.wait();
    }
}
```

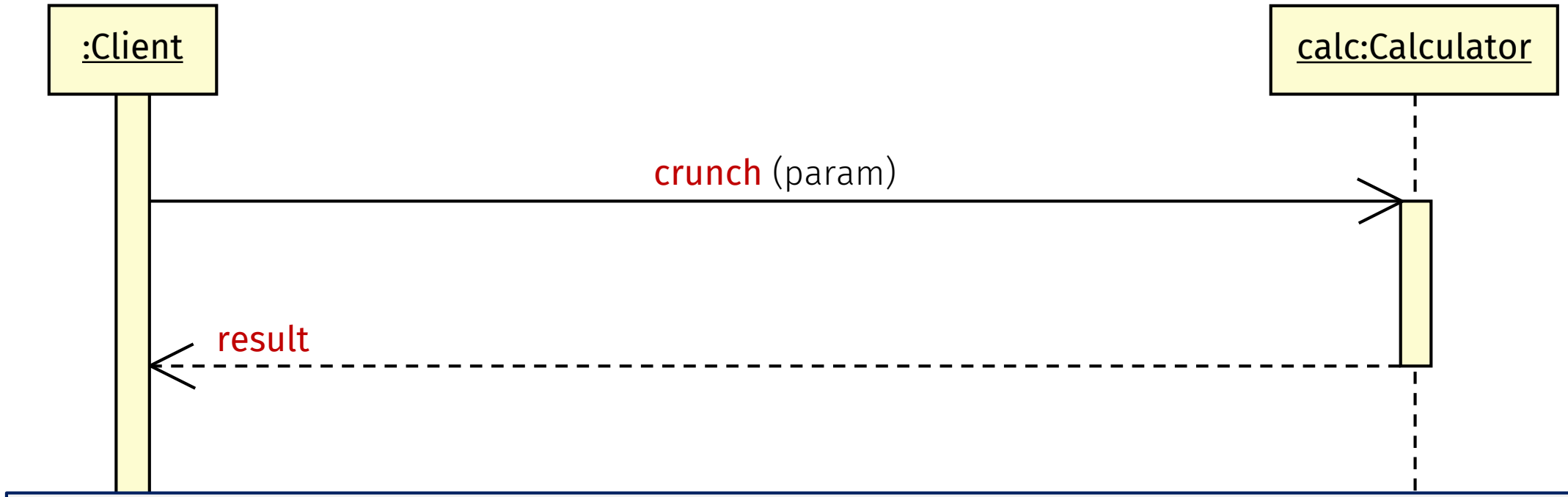
Classe Semaphore

- 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'une condition

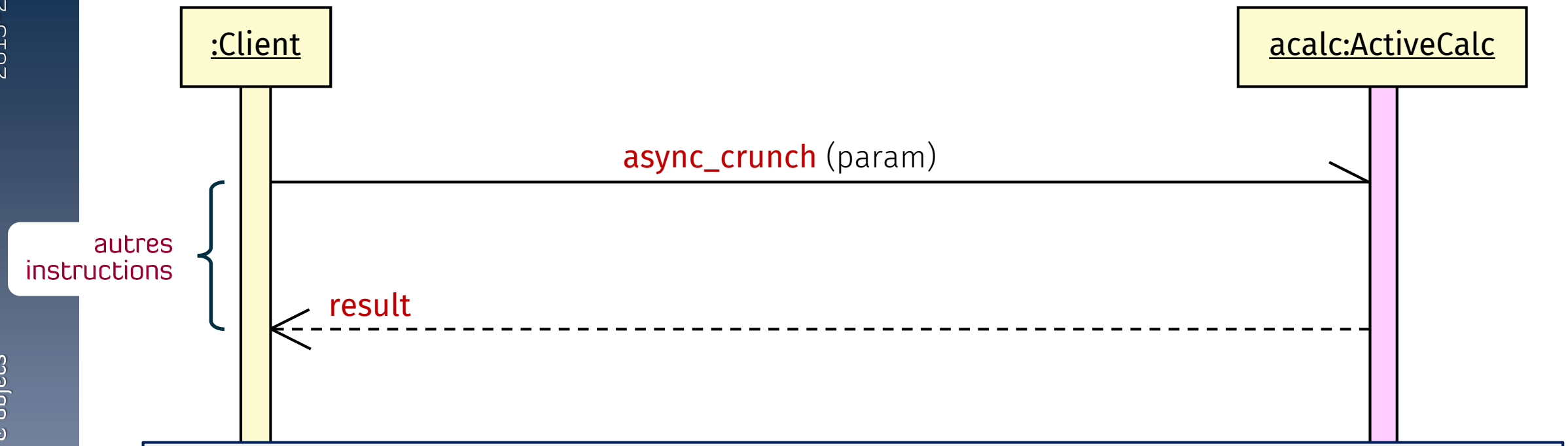


Objet passif : appel d'opération synchrone



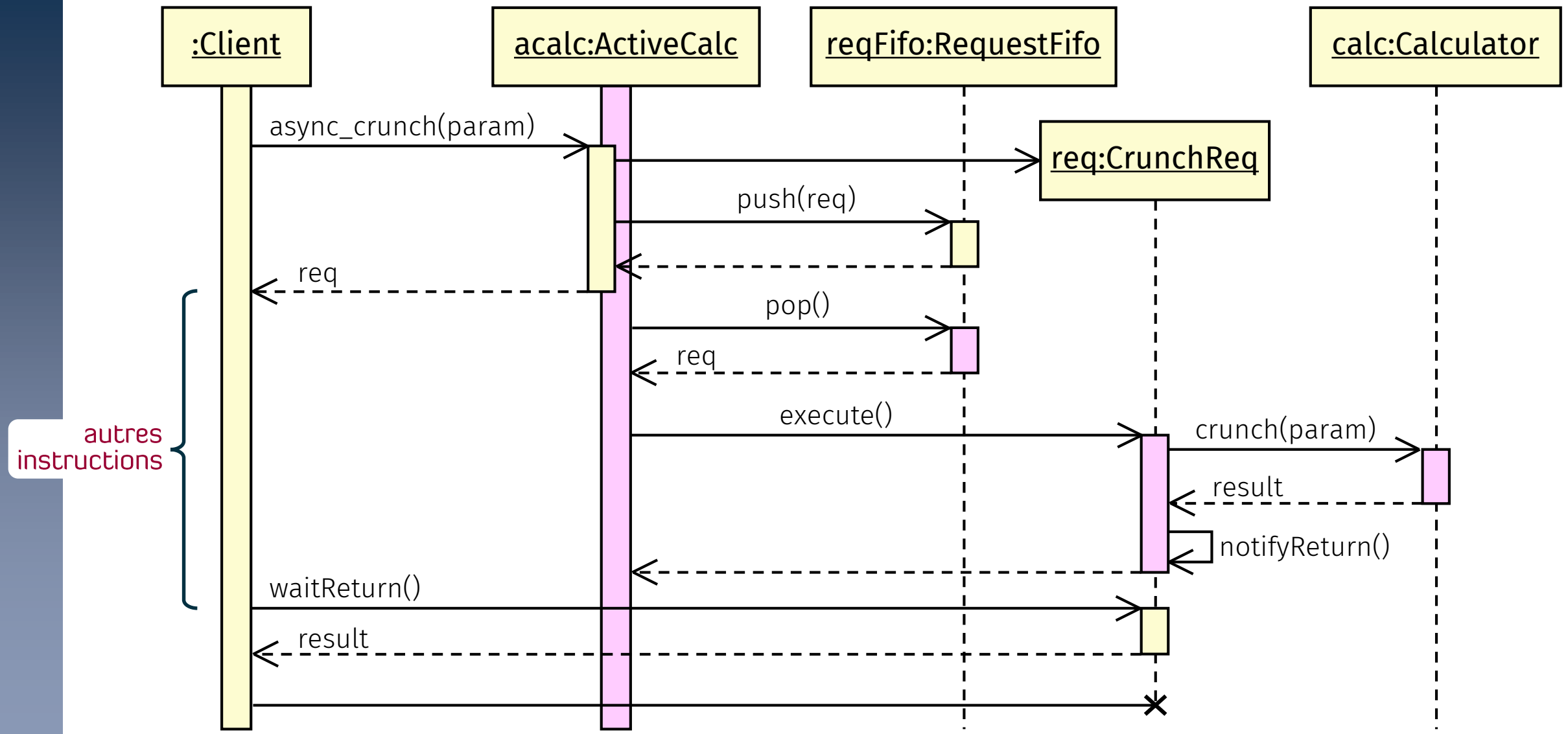

```
Client::run(Calculator* calc)
{
        int param = 10;


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

Objet actif : appel d'opération asynchrone




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

Objet actif : décomposition de l'appel asynchrone



Objet actif : décomposition de l'appel asynchrone


```
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 fin d'exécution du calcul (sémaphore)
    return result;                            // renvoi du résultat de calcul à l'appelant
}
```

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

Objet actif : classes principales