Table des matières

Bibliographie

Le guide de l'utilisateur UML ; Grady Booch, James Rumbaugh, Ivar Jacobson ; Ed. Eyrolles
UML 2 et les design patterns ; Craig Larman ; Ed. Pearson Education
UML 2 par la pratique; Pascal Roques ; Ed. Eyrolles

Présentation

Ce document présente un exemple d'utilisation de **threads** dans un environnement de développement intégré (EDI **Qt** et Borland **Builder C++**). Il montre aussi l'utilisation du langage de modélisation **UML** pour décrire cette application.

Structure de l'application

Il y aura 3 threads en plus du thread principal qui gère l'IHM :

- **TLectureConfiguration** : ce thread a en charge la prise en compte des paramètres de configuration (ici les périodicité des autres threads) et doit le signaler, en cas de changement, au thread concerné. C'est une tâche périodique activée par un *timer*. La priorité de cette tâche est fixée à « basse » (en dessous de la normale).

- **TAcquisitionMesures** : ce thread réalise périodiquement l'acquisition des mesures (simulée dans l'exemple). La périodicité est modifiable en cours d'exécution. La priorité de cette tâche est fixée à « haute » (au dessus de la normale).

- **TRecuperationFichiers** : ce thread réalise périodiquement la récupération de fichiers par le réseau (simulée dans l'exemple). La périodicité est modifiable en cours d'exécution. La priorité de cette tâche est fixée à « normale » (priorité normale).

Fonctionnement :
Les deux threads TAcquisitionMesures et TRecuperationFichiers doivent réaliser leur traitement tous les x ms (périodicité de chaque tâche). Pour réaliser cela, on les place en attente (le temps d'attente est égale à leur périodicté). A l'expériration de ce temps d'attente (le *timeout*), les deux threads réalisent leur traitement dédié (acquisition des mesures pour l'un et récupération de fichiers pour l'autre). Si l'attente est interrompue, déclenchée par le thread TLectureConfiguration, le thread en question met à jour sa périodicité, réalise son traitement et se replace en attente (voir chronogrammes page 15).

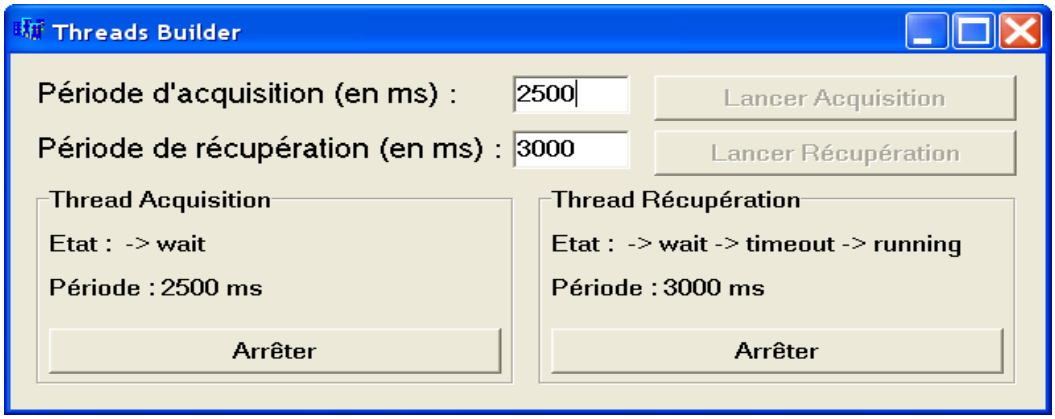
Remarque : Certains choix implémentés sont « discutables » et ne sont présents que pour montrer leur utilisation dans un exemple simple.

Qt ou Builder

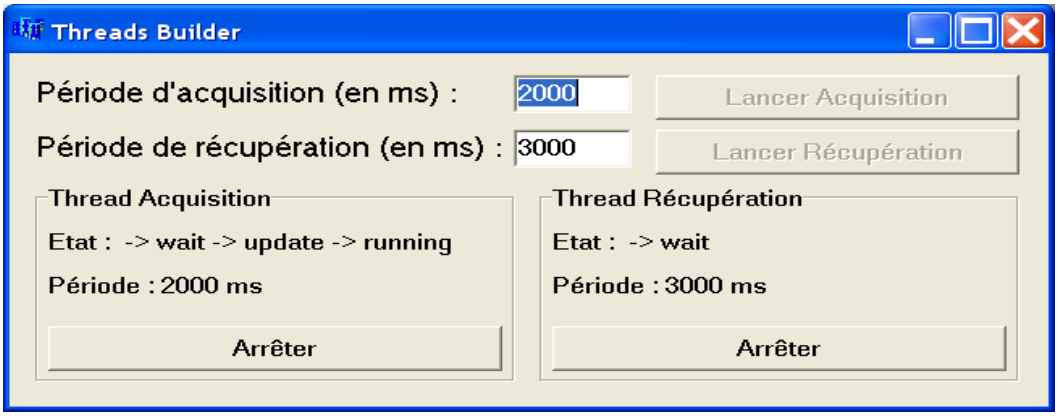
Il existe deux versions de cette application pour montrer les spécificités de deux environnements intégrés : Qt ou Builder C++. Leur fonctionnement est identique.

Version Builder C++

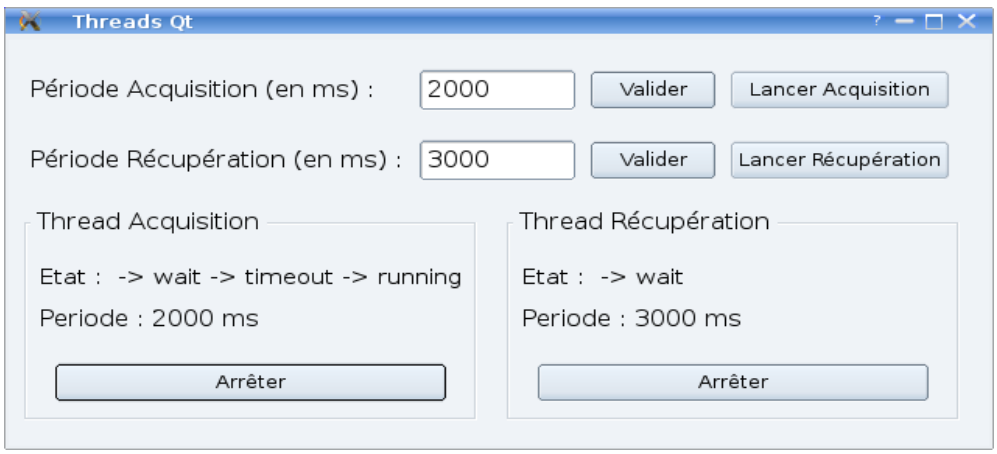
Les deux threads ont pris en compte leur périodicité : le thread TAcquisitionMesures est en attente et le thread TRecuperationFichiers est en train de récupérer les fichiers sur le réseau



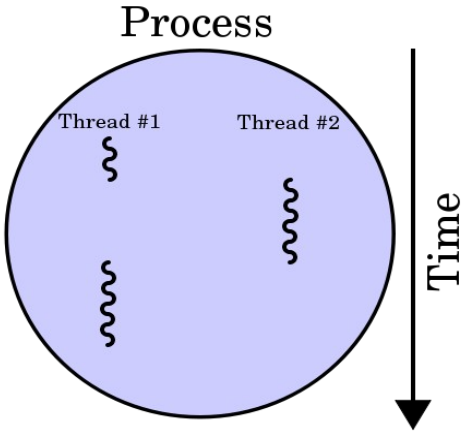
Puis, par exemple : le thread TAcquisitionMesures vient de prendre en compte la nouvelle périodicité et exécute une acquisition de mesures et le thread TRecuperationFichiers est en attente pour 3s avant sa prochaine récupération



Version Qt



Thread



Un ***thread*** ou fil (d'exécution) (autre appellation connue : **processus léger**), est similaire à un processus car tous deux représentent l'**exécution d'un ensemble d'instructions du langage machine d'un processeur**. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle (son espace mémoire adressable), les threads d'un même processus se partagent sa mémoire virtuelle.

Besoin

On aura besoin de thread(s) dans une application lorsqu'on devra **parallèliser des traitements**.

Avantages

- **Multi-tâche moins coûteux** : puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux threads est moins coûteuse que la commutation de contexte entre deux processus. On peut estimer (cela est très variable) que la création d'un processus lourd est plus de 30 fois plus gourmande en temps que la création d'un processus léger.
- **Communication entre threads plus rapide et plus efficace** : hormis le problème du coût de la commutation de contexte, le principal surcoût dû à l'utilisation de processus multiples provient de la communication entre processus séparés. En effet, le partage de certaines ressources entre threads permet une communication plus efficace entre les différents threads d'un processus. Là où deux processus séparés et indépendants doivent utiliser un mécanisme fourni par le système pour communiquer telle que IPC (*Inter Processus Communication*), les threads partagent une partie de l'état du processus.

Inconvénients

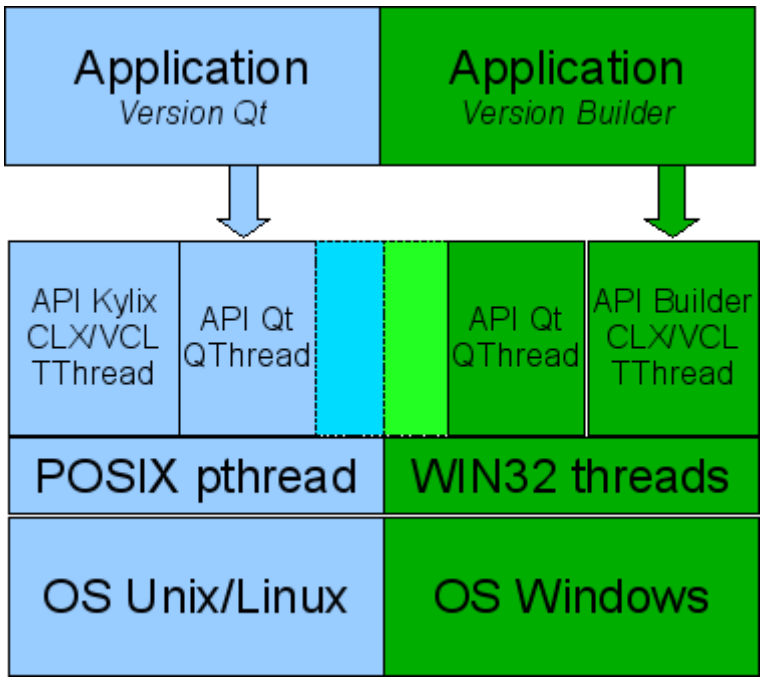
- **Programmation utilisant des threads est toutefois plus difficile**, et l'accès à certaines ressources partagées doit être restreint par le programme lui-même, pour éviter que l'état d'un processus ne devienne temporairement incohérent, tandis qu'un autre thread va avoir besoin de consulter cette portion de l'état du processus. Il est donc obligatoire de mettre en place des **mécanismes de synchronisation** (à l'aide de sémaphores par exemple), tout en conservant à l'esprit que l'utilisation de la synchronisation peut aboutir à des situations d'**interblocage**. La complexité des programmes utilisant des threads est aussi nettement plus grande que celle des programmes déférant le travail à faire à plusieurs processus plus simples. Cette complexité accrue, lorsqu'elle est mal gérée lors de la phase de conception ou de mise en œuvre d'un programme, peut conduire à de multiples problèmes.

Support

Les systèmes d'exploitation mettent en œuvre généralement les threads. Le standard des processus légers POSIX est connu sous le nom de *pthread*. Le standard POSIX est largement mis en oeuvre sur les systèmes UNIX. Microsoft fournit aussi une API pour les processus légers : WIN32 threads (Microsoft Win32 API threads).

Qt ou Builder

Dans le cadre d'un développement C++, l'utilisation d'une API simplifie la programmation :



La mise en place des threads revient à **dériver** une classe de base fournie par l'API choisie et à écrire le code du thread dans une méthode spécifique :

- sous **Qt**, on dérive une classe **QThread** et on écrit le code du thread dans la méthode **run()** :

```
#include <QThread>
class TAcquisitionMesures : public QThread
{
public:
    TAcquisitionMesures();
    void run();
};
```

- sous **Builder**, on dérive une classe **TThread** et on écrit le code du thread dans la méthode **Execute()** :

```
class TAcquisitionMesures : public TThread
{
protected:
    void __fastcall Execute();

public:
    __fastcall TAcquisitionMesures(bool CreateSuspended);
};
```

L'utilisation des threads se fera de la manière suivante :

- sous **Qt** :

```
// Crée un thread
TAcquisitionMesures *acquisitionMesures = new
TAcquisitionMesures();

// Démarre un thread (appelle la méthode run())
acquisitionMesures->start();

// Arrête un thread
acquisitionMesures->stop();
```

- sous **Builder** :

```
// Crée un thread à l'état suspendu
TAcquisitionMesures *acquisitionMesures = new
TAcquisitionMesures(true);

// Démarre un thread (appelle la méthode Execute())
acquisitionMesures->Resume();

// Arrête un thread
acquisitionMesures->Terminate();
```

Mécanismes de synchronisation

L'utilisation de threads dans une application oblige à mettre en place des **mécanismes de synchronisation** dans la communication entre tâches.

Sémaphore

Un sémaphore permet de **protéger l'accès à une variable partagée** (ou une zone de mémoire partagée) et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées (par exemple un espace de stockage) dans un environnement de programmation concurrente. Le sémaphore a été inventé par Edsger Dijkstra.

La valeur d'un sémaphore est le nombre d'unités de ressource (exemple : imprimantes...) libres ; s'il n'y a qu'une ressource, un sémaphore à système numérique binaire avec les valeurs 0 ou 1 est utilisé.

Sémaphores bloquants

Un sémaphore bloquant est un sémaphore qui est initialisé avec la valeur 0. Ceci a pour effet de bloquer n'importe quel thread qui effectue P(S) tant qu'un autre thread n'aura pas fait un V(S). Cette utilisation des sémaphores permet de réaliser des barrières de synchronisation.

Exclusion mutuelle

Il existe également le sémaphore binaire qui est une exclusion mutuelle (voir mutex). Il est toujours initialisé avec la valeur 1.

Problème des lecteurs/rédacteurs

Ce problème traite de l'accès concurrent en lecture et en écriture à une ressource. Plusieurs processus légers (thread) peuvent lire en même temps la ressource, mais il ne peut y avoir qu'un et un seul thread en écriture.

Problème des producteurs/consommateurs

Lorsque des processus légers souhaitent communiquer entre eux, ils peuvent le faire par l'intermédiaire d'une file. Il faut définir le comportement à avoir lorsqu'un thread souhaite lire depuis la file lorsque celle-ci est vide et lorsqu'un thread souhaite écrire dans la file mais que celle-ci est pleine.

Mutex

Un mutex (*mutual exclusion*, exclusion mutuelle) est une primitive de synchronisation qui permet d'éviter que des ressources partagées d'un système ne soient utilisées en même temps. Un mutex a deux états : verrouillé ou non verrouillé. La plupart des mutex ou des sémaphores ont des effets secondaires qui peuvent bloquer l'exécution, créer des goulets d'étranglement, voire ne pas remplir leur rôle en permettant tout de même l'accès aux données protégées. Un autre effet est le blocage total des ressources, si le programme qui les utilisait n'a pas informé le système qu'il n'en avait plus besoin.

Variable condition

Une variable-condition est un mécanisme de synchronisation permettant à un thread de suspendre son exécution jusqu'à ce qu'une certaine condition soit vérifiée. Deux opérations sont disponibles : *wait*, qui bloque le processus léger tant que la condition est fausse et *signal* (ou *wake*) qui prévient les processus bloqués que la condition est vraie. Il existe également une attente limitée permettant de signaler automatiquement la variable condition si le réveil ne s'effectue pas pendant une durée déterminée.
Une variable condition doit toujours être associée à un mutex, pour éviter les accès concurrents.

L'interblocage (deadlock)

L'interblocage (*deadlock*) se produit, par exemple, lorsqu'un thread T1 ayant déjà acquis la ressource R1 demande l'accès à une ressource R2, pendant que le thread T2, ayant déjà acquis la ressource R2, demande l'accès à la ressource R1. Chacun des deux threads attend alors la libération de la ressource possédée par l'autre. La situation est donc bloquée.

Chien de garde (watchdog)

Un chien de garde (watchdog), est une technique logicielle utilisée pour s'assurer qu'un programme ne reste pas bloqué à une étape particulière du traitement qu'il effectue. C'est une protection destinée généralement à redémarrer le système, si une action définie n'est pas exécutée dans un délai imparti.

Il s'agit en général d'un compteur qui est régulièrement remis à zéro. Si le compteur dépasse une valeur donnée (***timeout***) alors on procède à un reset (redémarrage) du système. Si une routine entre dans une boucle infinie, le compteur du chien de garde ne sera plus remis à zéro et un *reset* est ordonné.

Section critique

Une section critique est une portion de code dans laquelle il doit être garanti qu'il n'y aura jamais plus d'un thread simultanément. Il est nécessaire d'utiliser des sections critiques lorsqu'il y a accès à des ressources partagées par plusieurs thread. Une section critique peut être protégée par un mutex, un sémaphore ou d'autres primitives de programmation concurrente.

Qt ou Builder

Sous **Qt**, on met en oeuvre les mécanismes suivants :

- des mutex **QMutex** qui dispose de deux méthodes, **lock** (qui empêche les autres threads d'exécuter la section) et **unlock** (qui retire le blocage) et un sémaphore **QSemaphore** qui dispose de deux méthodes, **acquire** (qui empêche les autres threads d'exécuter la section) et **release** (qui retire le blocage)

- un synchronisateur à écriture exclusive et à lecture multiple **QReadWriteLock** associée à une zone mémoire à accès protégé. Cet objet agit comme une section critique qui permet à plusieurs threads de lire la mémoire qu'il protège à condition qu'aucun thread n'y écrive. Les threads doivent disposer d'un accès exclusif en écriture à la mémoire protégée. Chaque thread qui lit cette mémoire doit au préalable appeler la méthode **lockForRead**. BeginRead évite qu'un autre thread n'écrive simultanément dans la mémoire. Lorsqu'un thread a fini de lire la mémoire protégée, il appelle la méthode **unlock**. Tout thread qui écrit dans la mémoire protégée doit au préalable appeler **lockForWrite** qui évite qu'un autre thread ne lise ou n'écrive simultanément dans la mémoire. Lorsqu'un thread a fini d'écrire dans la mémoire protégée, il appelle la méthode **unlock**, de sorte que les threads en attente puissent commencer à lire la mémoire.

- des variables conditions **QWaitCondition** qui agissent comme des signaux visibles pour tous les threads. Quand un thread termine une opération dont dépendent d'autres threads, il le signale en appellant **wakeAll**. wakeAll active le signal afin que les autres threads en attente (**wait**) soit débloqués.

Et sous **Borland Builder C++**, on met en oeuvre les mécanismes suivants :

- une section critique **TCriticalSection** qui dispose de deux méthodes, **Acquire** (qui empêche les autres threads d'exécuter la section) et **Release** (qui retire le blocage)

- un synchronisateur à écriture exclusive et à lecture multiple **TMultiReadExclusiveWriteSynchronizer** associée à une zone mémoire à accès protégé. Cet objet agit comme une section critique qui permet à plusieurs threads de lire la mémoire qu'il protège à condition qu'aucun thread n'y écrive. Les threads doivent disposer d'un accès exclusif en écriture à la mémoire protégée. Chaque thread qui lit cette mémoire doit au préalable appeler la méthode **BeginRead**. BeginRead évite qu'un autre thread n'écrive simultanément dans la mémoire. Lorsqu'un thread a fini de lire la mémoire protégée, il appelle la méthode **EndRead**. Tout thread qui écrit dans la mémoire protégée doit au préalable appeler **BeginWrite** qui évite qu'un autre thread ne lise ou n'écrive simultanément dans la mémoire. Lorsqu'un thread a fini d'écrire dans la mémoire protégée, il appelle la méthode **EndWrite**, de sorte que les threads en attente puissent commencer à lire la mémoire.

- des objets événements **TEvent** qui agissent comme des signaux visibles pour tous les threads. Quand un thread termine une opération dont dépendent d'autres threads, il le signale en appellant **TEvent::SetEvent**. SetEvent active le signal afin que les autres threads le surveillant sachent que l'opération a été achevée. Pour désactiver le signal, on utilise la méthode **ResetEvent**.

Remarque : quand on utilise des objets appartenant aux hiérarchies d'objets VCL (ou CLX), leurs propriétés et méthodes ne sont pas nécessairement adaptées à l'utilisation de threads. C'est-à-dire que l'accès aux propriétés et méthodes peut effectuer des actions utilisant de la mémoire qui n'est pas protégée de l'action d'autres threads. De ce fait, si un thread doit accèder à ces objets, il doit passer par l'appel à la méthode **Synchronize** du thread.

UML

Rappels

L'architecture logicielle d'une application est définie par 5 vues :

- Vue des cas d'utilisation : les cas d'utilisation décrivent le comportement du système.
- Vue de conception : cette vue représente les besoins fonctionnels du système :
- **Vue des processus : cette vue précise les *threads* et les processus qui forme les mécanismes de concurrence et de synchronisation du système. On met l'accent sur les classes actives qui représentent les *threads* et les processus. Elle montre : La décomposition du système en terme de processus (tâches), les interactions entre les processus (leur communication) et la synchronisation et la communication des activités parallèles (threads).**
- Vue d'implémentation : elle englobe les composants et les fichiers utilisés pour assembler et produire le système physique, l'exécutable. On met en œuvre la gestion de la configuration
- Vue de déploiement : elle englobe les nœuds qui forme la topologie du système et indique où sont déployés les composants.

Remarque : Une vue est une description simplifiée d'un système observé d'un point de vue particulier.

Les diagrammes utilisées pour décrire la vue des processus sont :

- les diagrammes de séquence ;
- les diagrammes de collaboration ;
- les diagrammes d'objets ;
- les diagrammes d'états ;
- les diagrammes d'activités.

Classe active et objet actif

UML fournit un repère visuel (bord en trait épais) qui permet de distinguer les éléments actifs (processus ou thread) des éléments passifs. Une instance d'une **classe active** sera nommée **object actif**. Chaque processus ou thread au sein d'un système définit alors un **flot de contrôle** distinct.

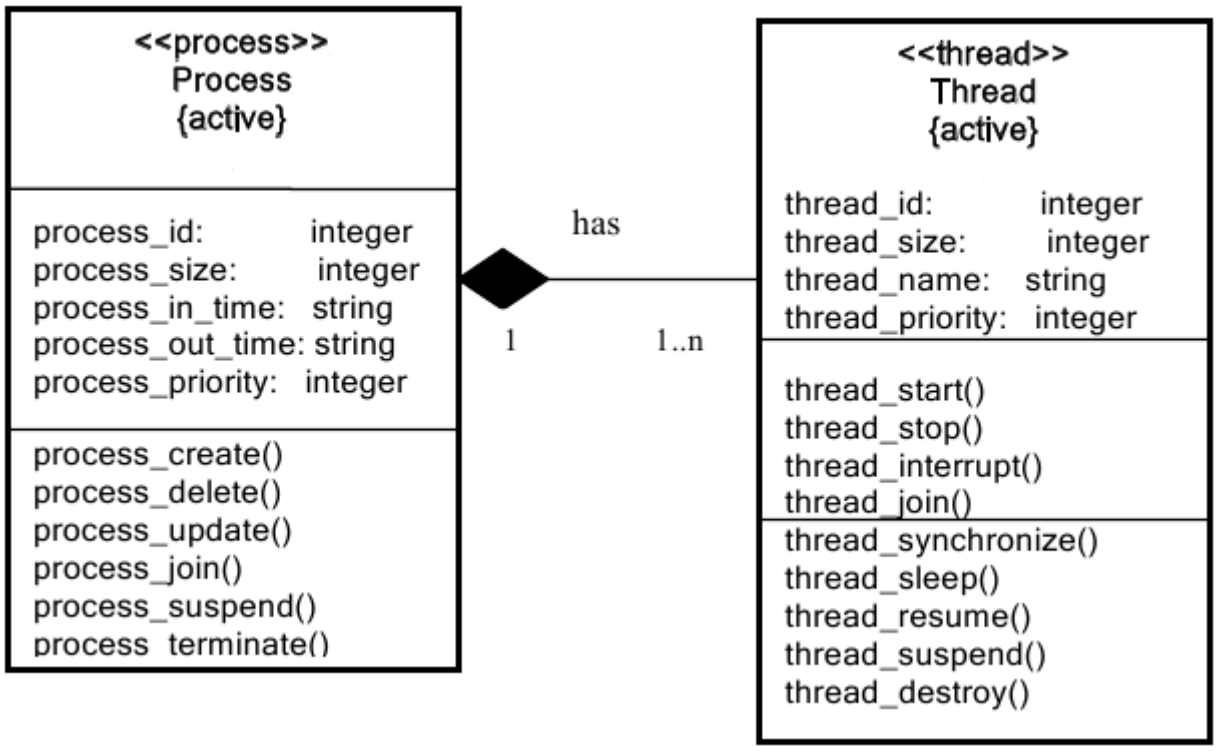
Remarque : bouml ne permet pas encore de visualiser graphiquement les classes actives !

Stéréotype

UML définit deux stéréotypes standards qui s'appliquent aux classes actives :

- **process** : spécifie un flot « lourd » qui peut s'exécuter en concurrence avec d'autres processus.
- **thread** : spécifie un flot « léger » qui peut s'exécuter en concurrence avec d'autres threads à l'intérieur d'un même processus.

Représentation UML des concepts de processus et de thread :



Méthodologie

Le découpage de l'application en threads apparaît lorsqu'on établit :

- les diagrammes d'activités : activités concurrentes et/ou
- les diagrammes d'états : états concurrents et/ou
- les diagrammes de séquences : exécutions concurrentes de plusieurs objets

Le besoin est toujours la parallèlisation :

- le système réalise plusieurs activités en même temps et/ou
- un objet prend plusieurs états en même temps et/ou
- une utilisation du système réalise plusieurs exécutions en même temps

Rappels :

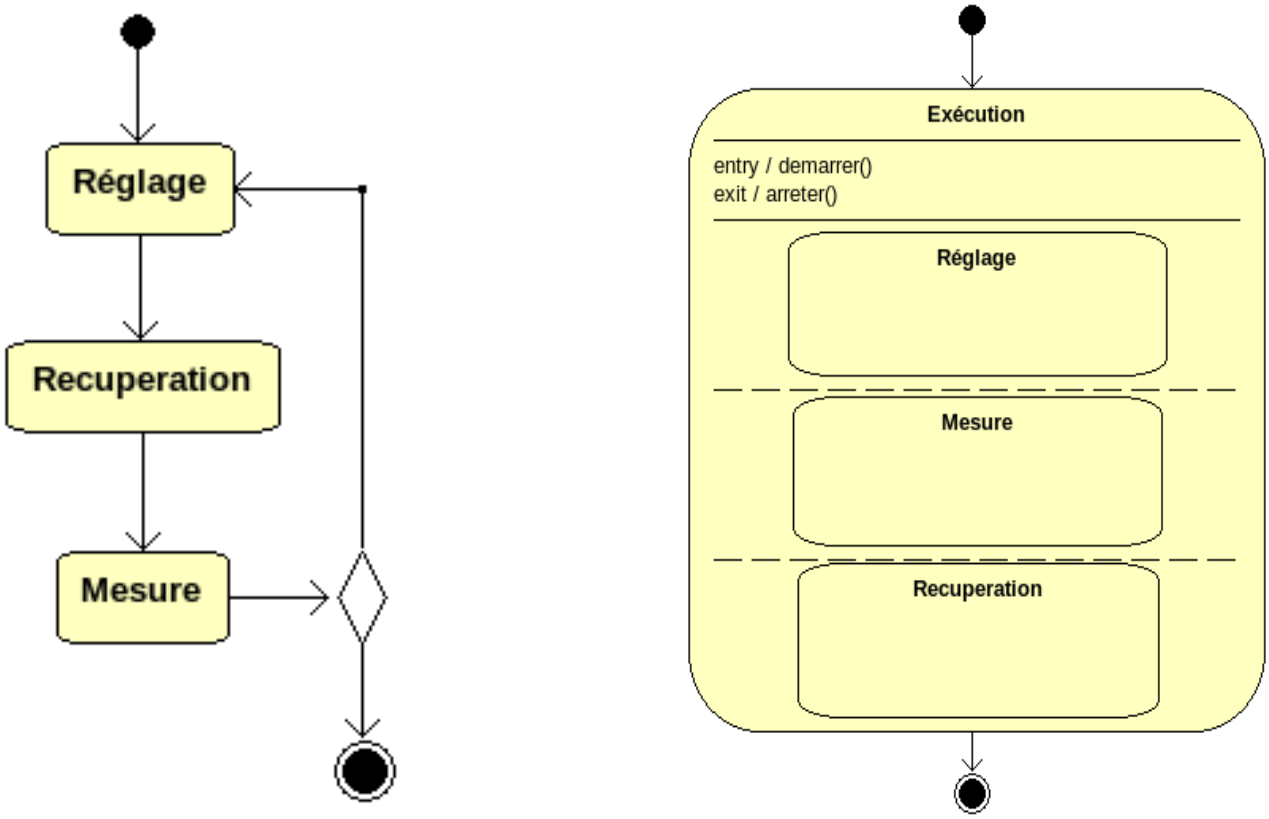
- plusieurs axes de recherche (il n'y a pas d'ordre établi dans les diagrammes à réaliser)
- méthode itérative et incrémentale (un enrichit l'autre, même le codage !)

Séquentiel versus Parallèle

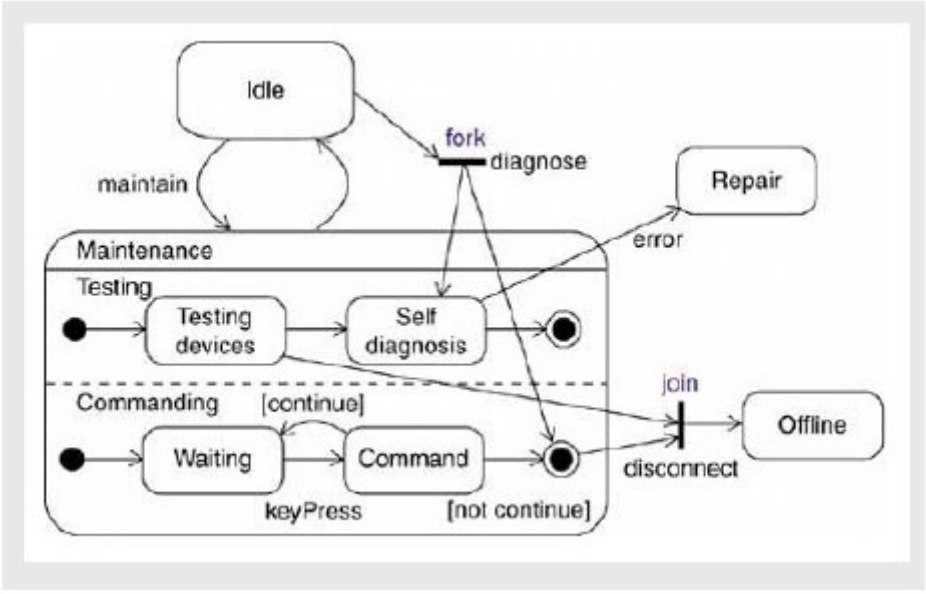
Le besoin de thread apparaît clairement pendant la phase d'analyse lorsqu'on distingue ce qui se produit de manière séquentielle de ce qui se produit de manière parallèle.

On peut illustrer cela de manière simple :

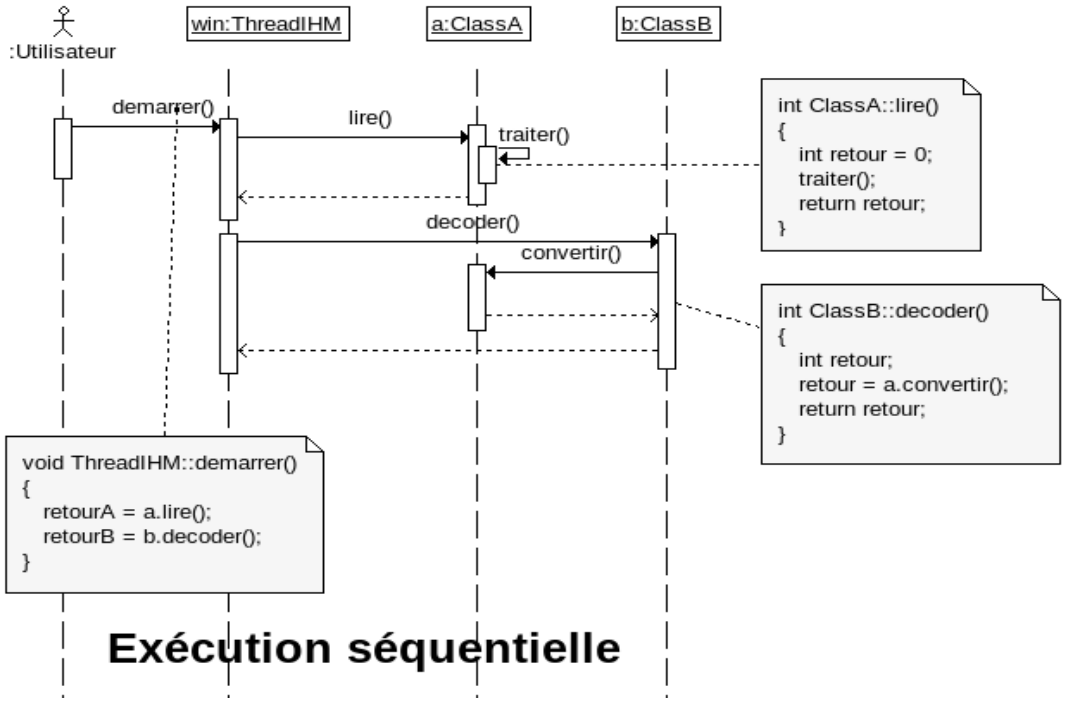
- **dans les diagrammes d'états** : à gauche, le système change d'état successivement (réglage puis récupération puis mesure) : pas besoin de threads. A droite, l'application se retrouve en parallèle dans tois états concurrents : besoin de threads.

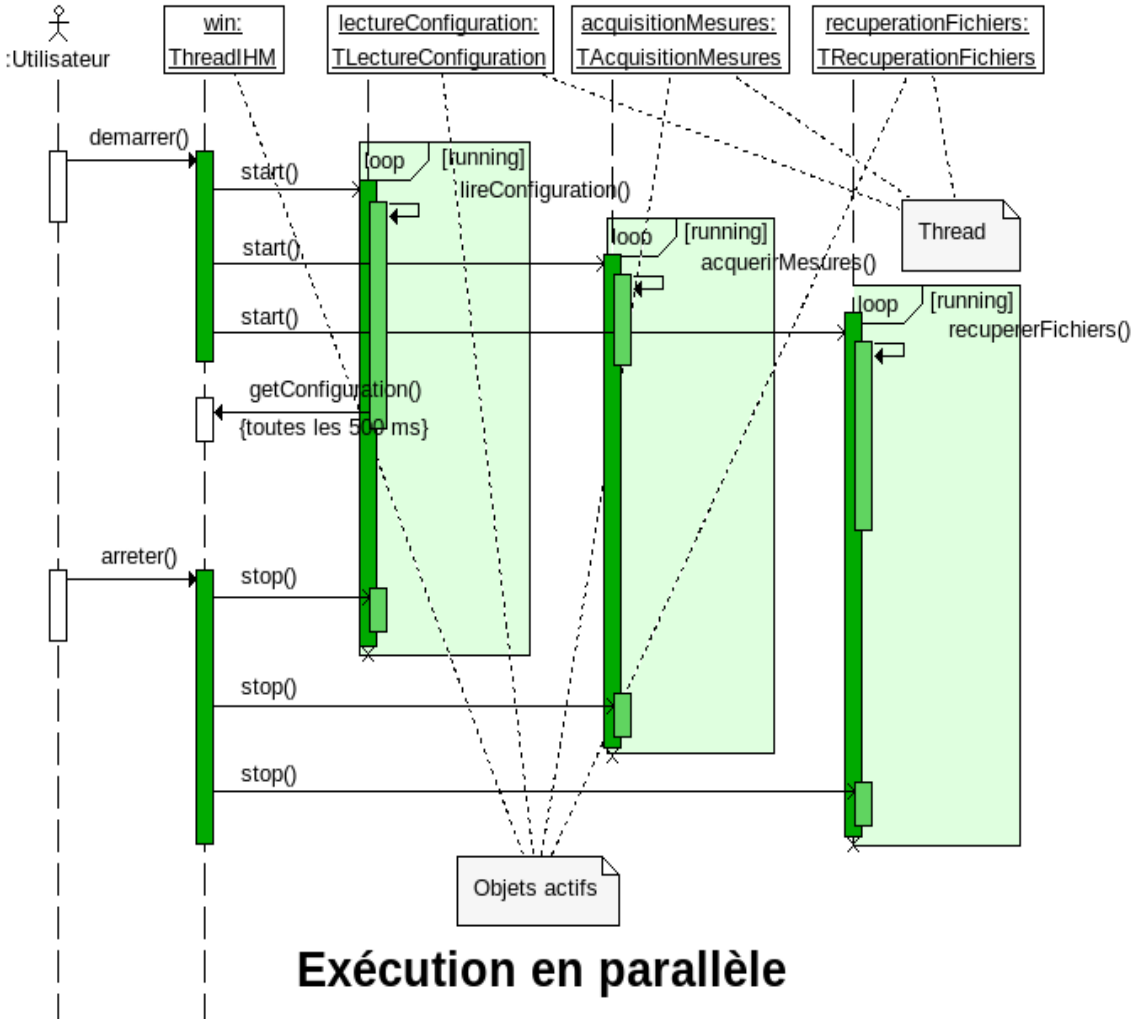


Exemple d'un diagramme d'états concurrents :

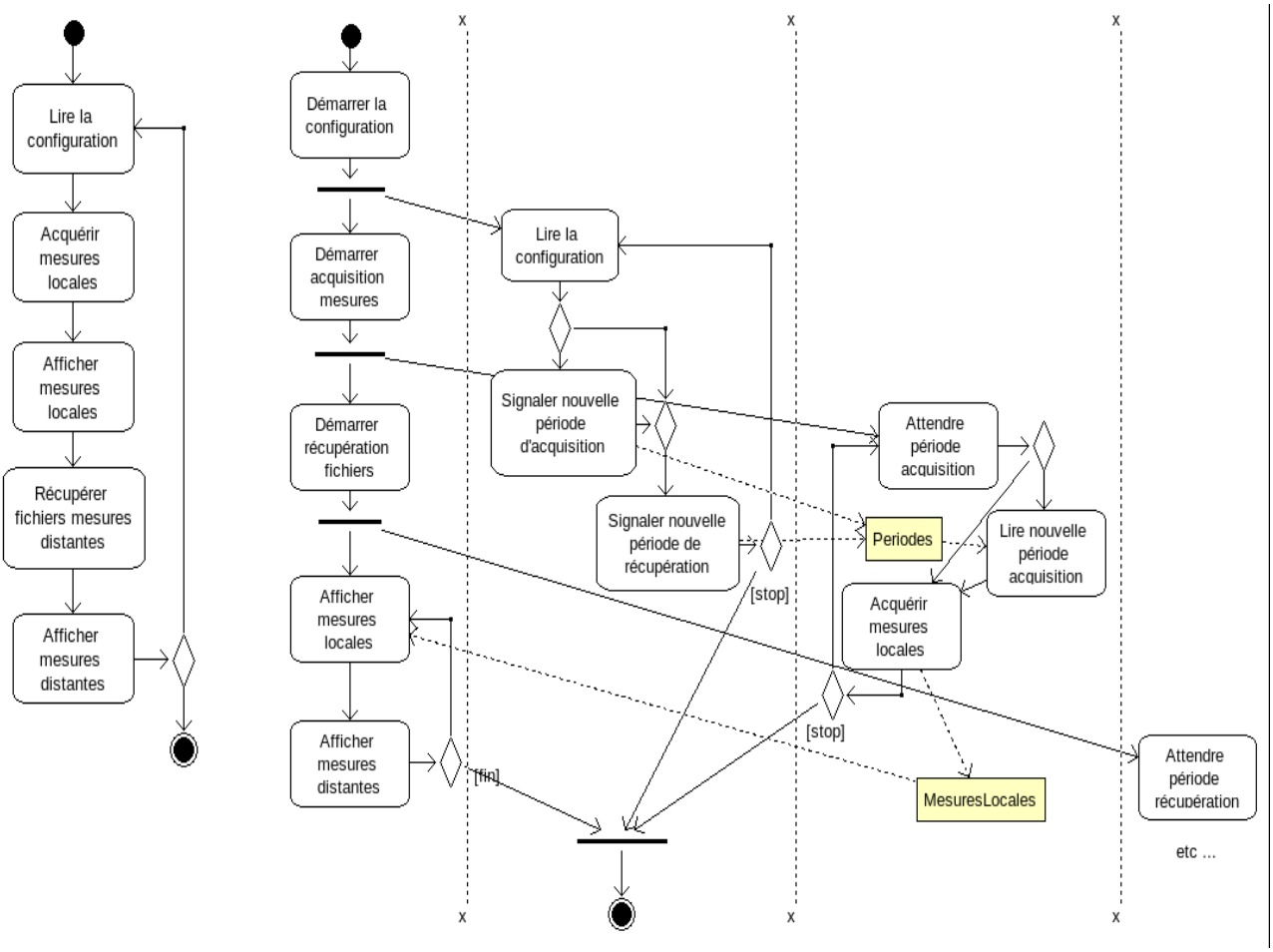


- **dans les diagrammes de séquences** :





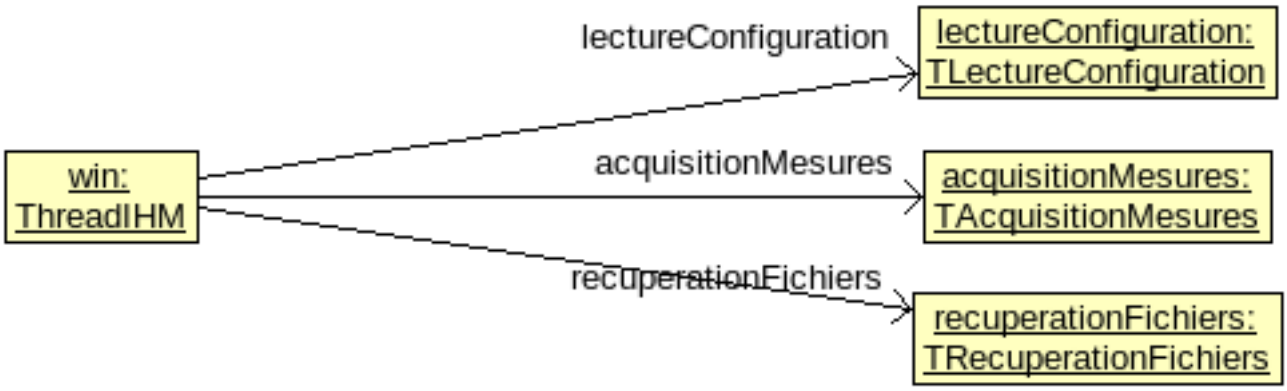
- **dans les diagrammes d'activités** : le diagramme de gauche montre un enchaînement d'actions séquentielles : pas besoin de threads. Celui de droite illustre des activités concurrentes en parallèle : besoin de threads. Une barre de synchronisation permet d'ouvrir et de fermer des branches parallèles au sein d'un flot d'exécution.



Points de vue

Les diagrammes UML peuvent être utilisés pour illustrer un point de vue particulier.

Exemple d'un diagramme d'objets qui montre les relations entre eux :



Remarque : bouml ne montrant pas graphiquement les objets actifs, il faudra ajouter des notes pour le préciser (ici un process et 3 threads).

Le cas d'utilisation Démarrer :
Les systèmes possèdent explicitement ou implicitement un cas d'utilisation que l'on nomme souvent Démarrer qui correspond à l'initialisation de l'application.

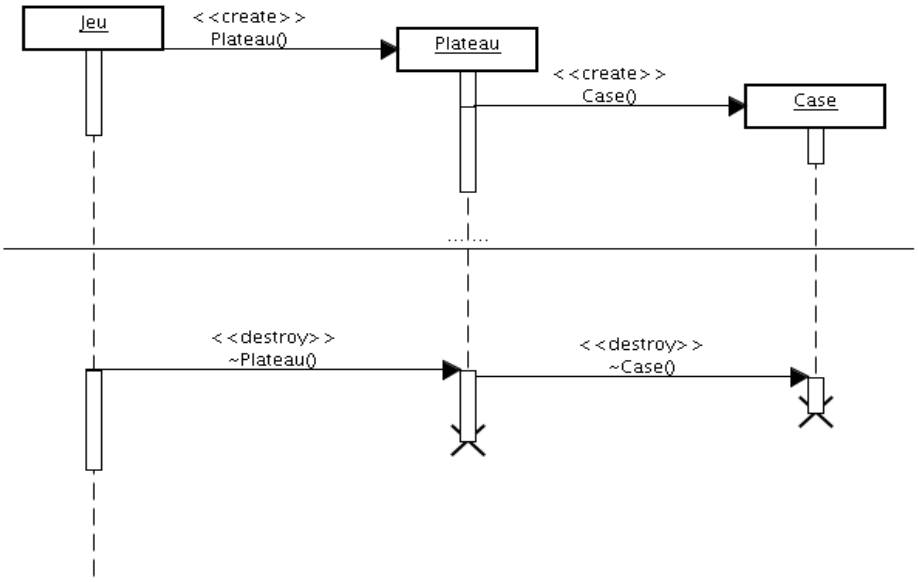
Remarque : concevoir l'initialisation en dernier
Lors de l'implémentation, il faudra coder en premier au moins un cas d'utilisation Démarrer (l'opération système d'initialisation est la première à s'exécuter au lancement de l'application). Mais, pendant la modélisation, il faut le considérer en dernier après avoir découvert ce qui doit être créé et initialisé.

Comment les applications démarrent-elles ?
L'opération système démarrer ou initialiser représente la phase initiale de lancement d'une application. Le lancement d'une application dépend du langage de programmation et du système d'exploitation.

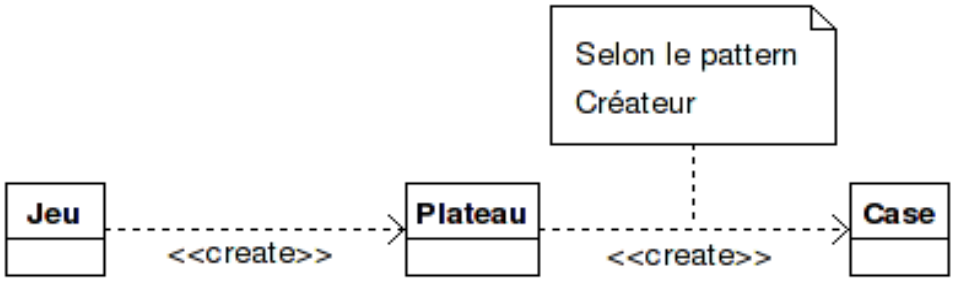
La règle : elle consiste à créer un objet du domaine initial qui sera le premier objet logiciel du « domaine » à être créé. Cette création peut avoir lieu dans la méthode main().

Le choix : on choisi un objet racine approprié qui sera le créateur de certains autres objets.

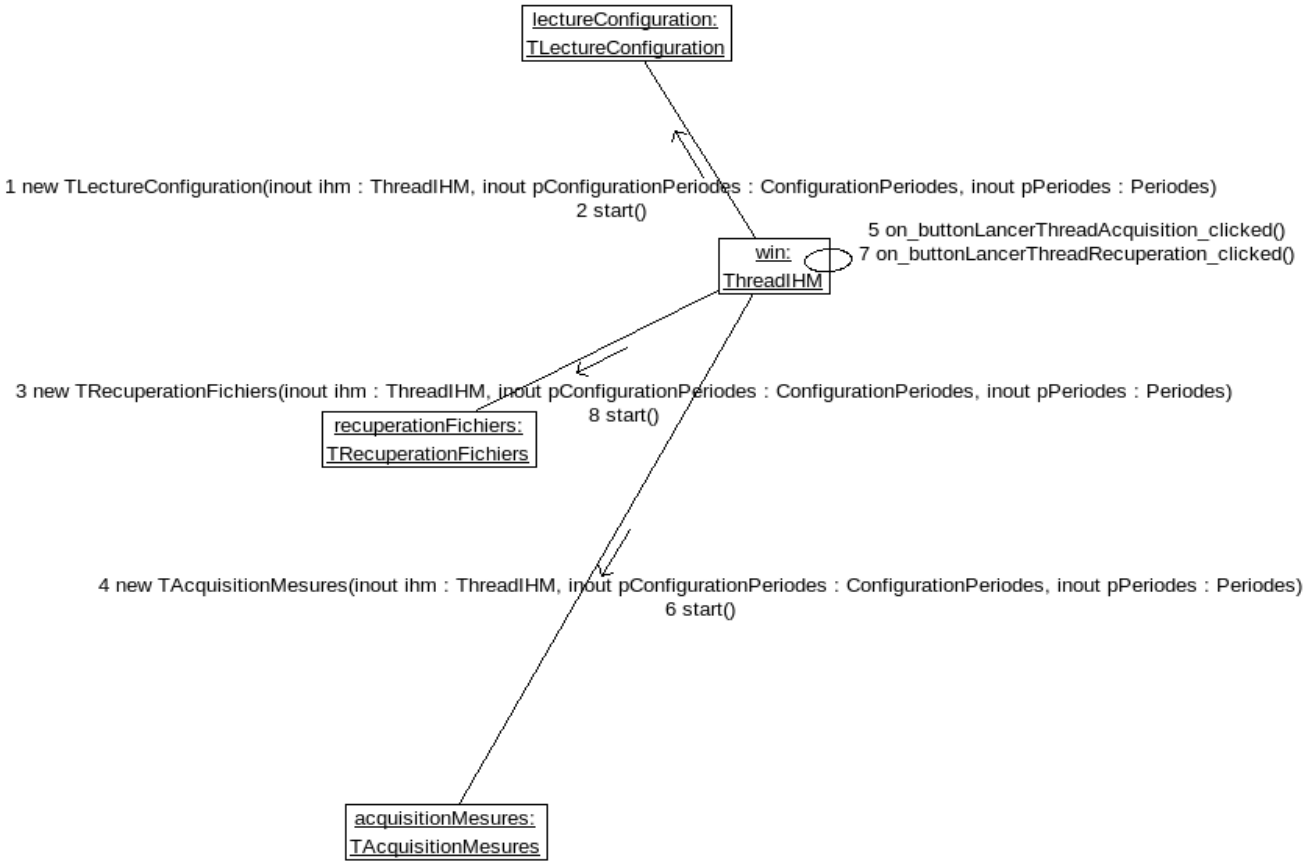
La représentation :
On peut représenter les détails de cette conception dynamique au moyen de diagramme d'interaction (séquence et/ou collaboration). Par exemple :



Une autre approche consiste à utiliser un diagramme (statique) de classes avec des dépendances utilisant le stéréotype « create ». Cette solution donne une bonne vue de l'ensemble des créations. Par exemple :



Un diagramme de communication (ou collaboration) pourra aussi être utilisé, permettant d'indiquer l'ordre des messages :



De la même manière, on pourra simplifier un diagramme de classe suivant ce que l'on veut montrer :

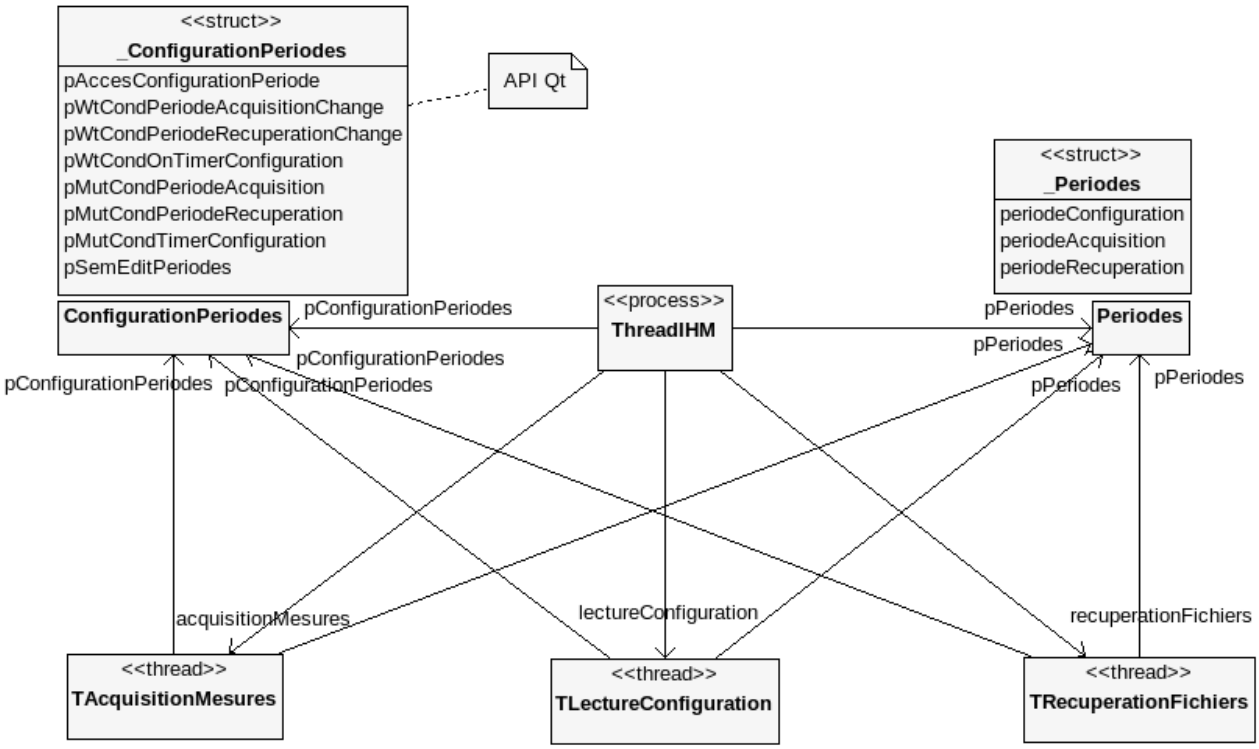
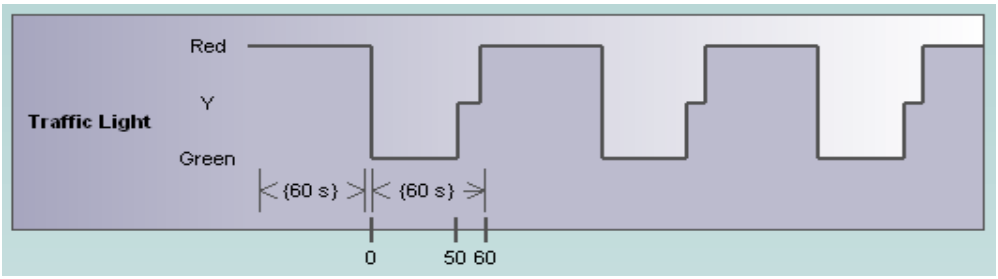
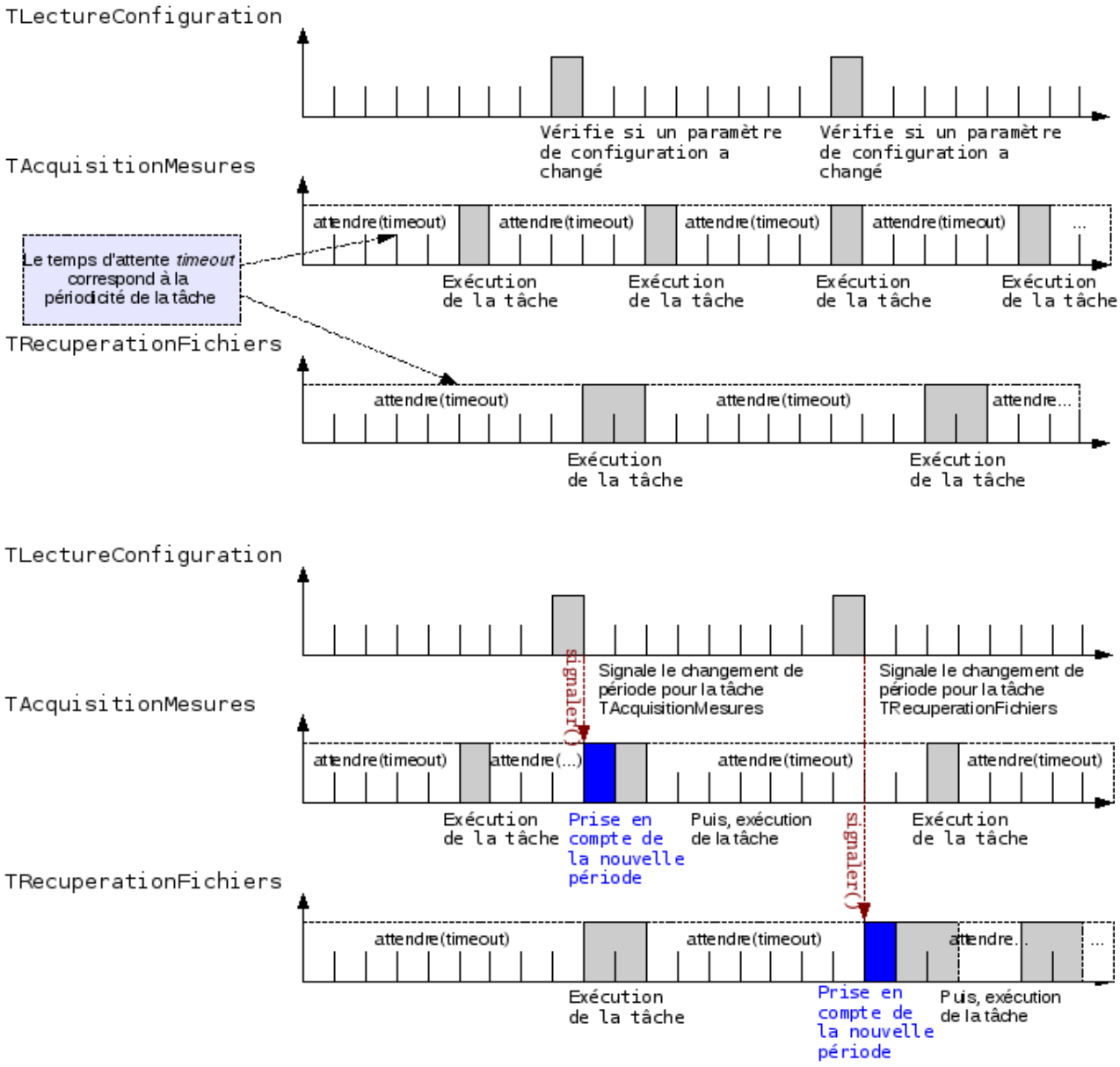


Diagramme de temps

UML2 a introduit l'utilisation d'un nouveau diagramme « *timing diagram* » :



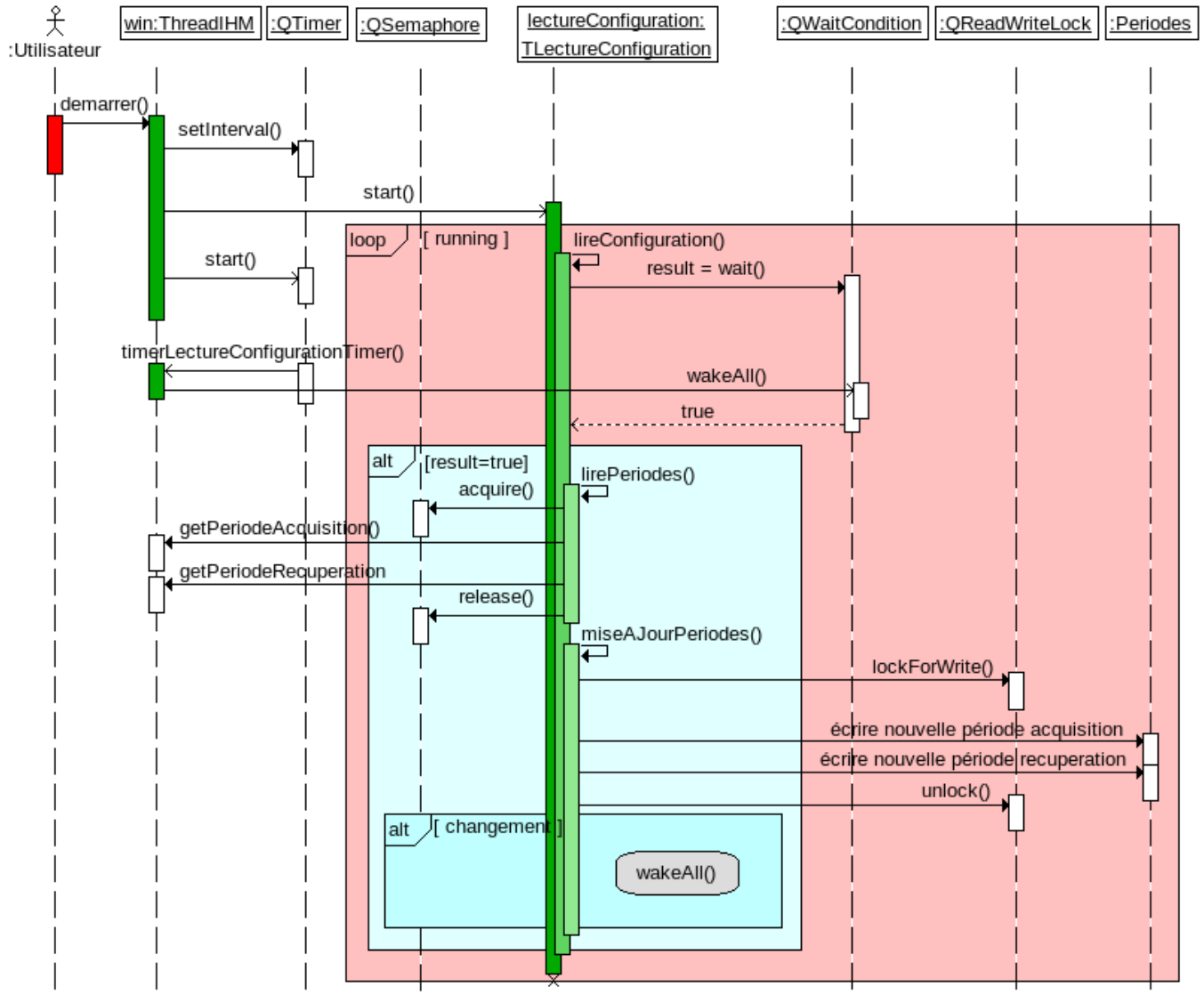
Bouml ne supportant pas encore ce type de diagramme, on pourra utiliser des chronogrammes pour représenter une vue temporelle :



- Sous **Qt** : attendre() = **`wait()`** et signaler() = **`wakeAll()`**
- Sous **Builder** : attendre() = **`WaitFor()`** et signaler() = **`SetEvent()`**

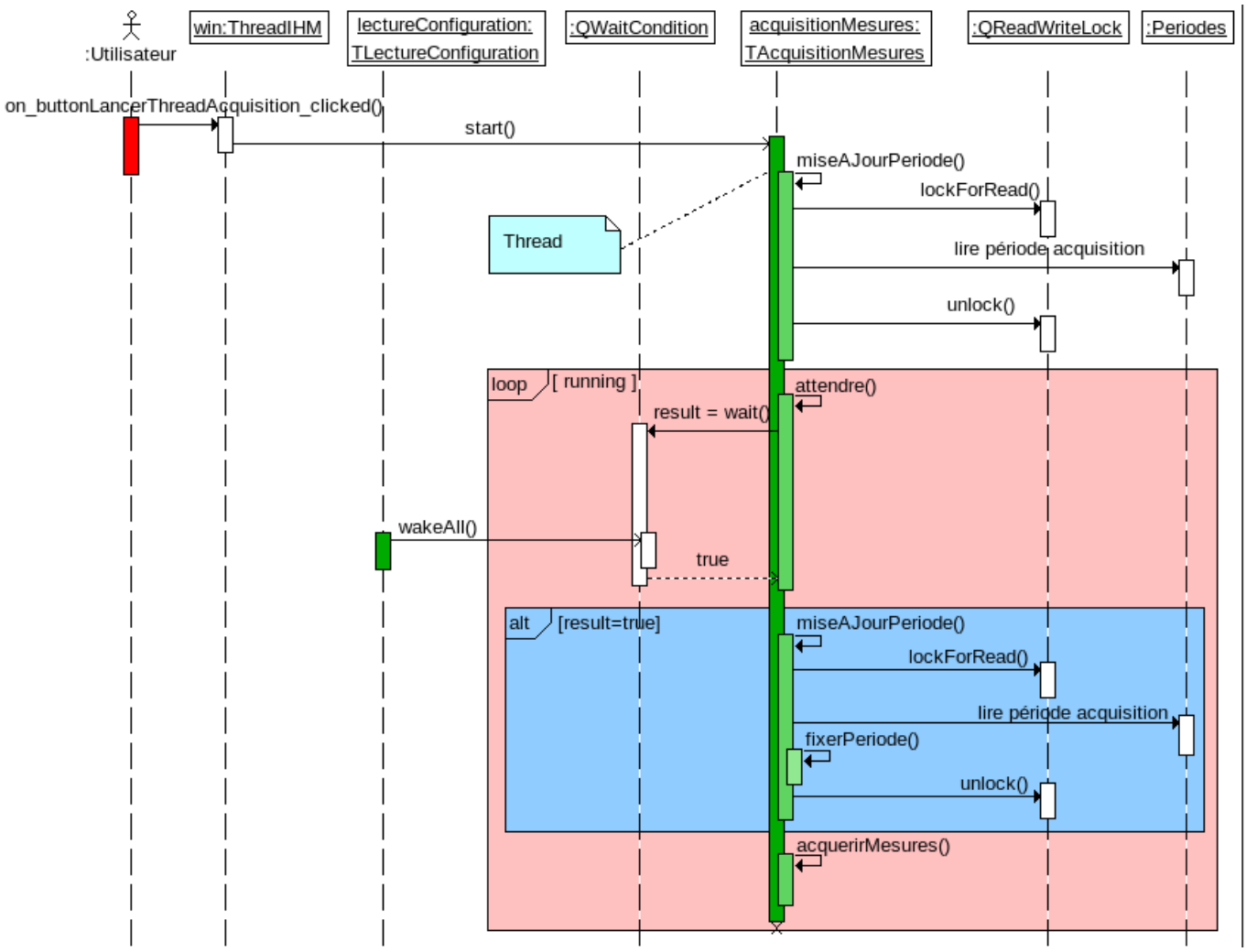
Annexe : Diagramme de séquence du scénario « démarrer »

Exemple pour QT :



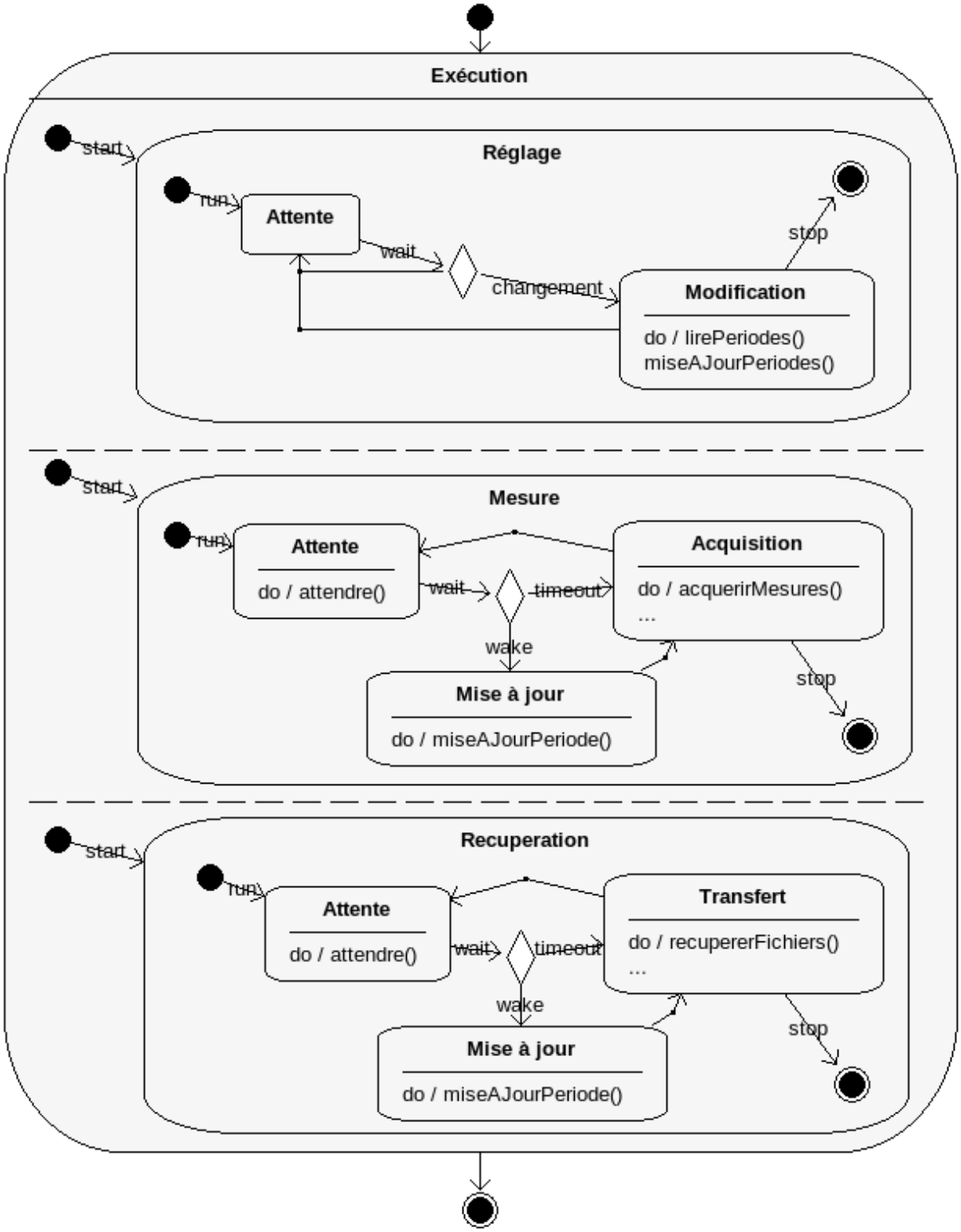
Annexe : Diagramme de séquence du scénario « acquérir les mesures locales »

Exemple pour Qt :



Annexe : Diagramme d'états de l'application

Exemple pour Qt :



Annexe : Structures de données

Les mécanismes de synchronisation entre threads sont regroupés dans une structure :

- sous **Qt** :

```
typedef struct
{
  QReadWriteLock *pAccesConfigurationPeriode;
  QWaitCondition *pWtCondPeriodeAcquisitionChange;
  QWaitCondition *pWtCondPeriodeRecuperationChange;
  QWaitCondition *pWtCondOnTimerConfiguration;
  QMutex         *pMutCondPeriodeAcquisition;
  QMutex         *pMutCondPeriodeRecuperation;
  QMutex         *pMutCondTimerConfiguration;
  QSemaphore     *pSemEditPeriodes;
} ConfigurationPeriodes;
```

- sous **Builder** :

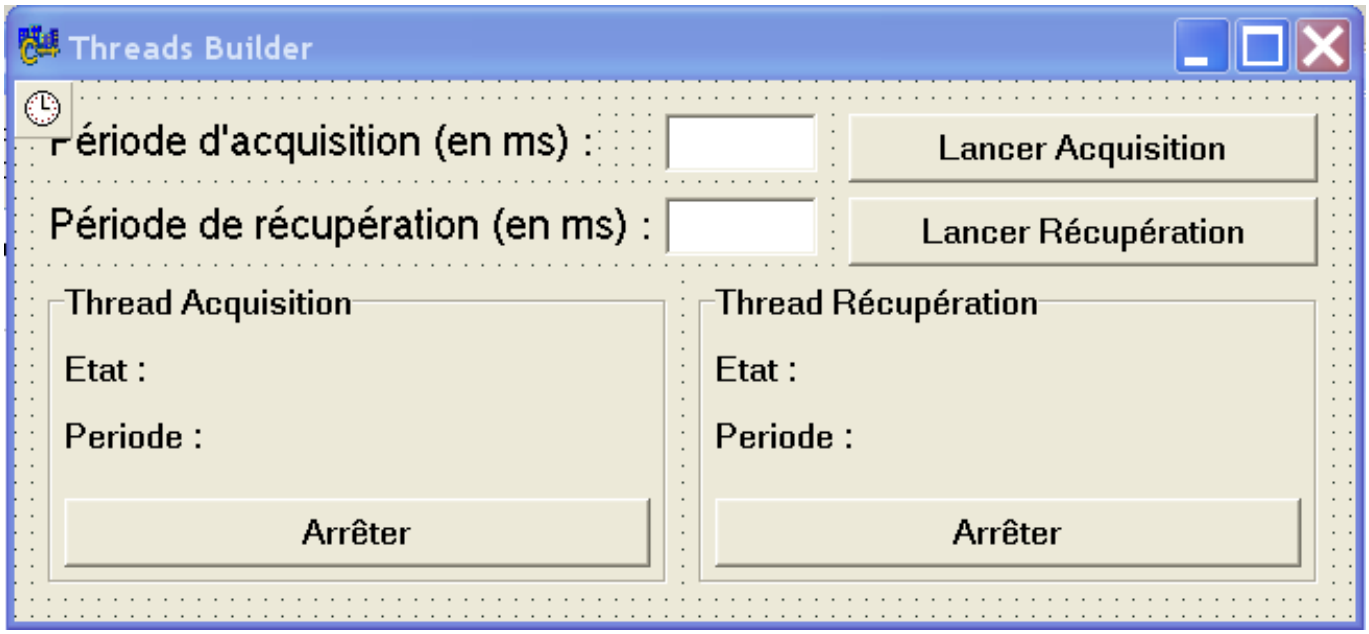
```
typedef struct
{
  TCriticalSection *pLockEditPeriodes;
  TMultiReadExclusiveWriteSynchronizer
*pAccesConfigurationPeriode;
  TEvent *pEvtPeriodeAcquisitionChange;
  TEvent *pEvtPeriodeRecuperationChange;
  TEvent *pEvtOnTimerConfiguration;
} ConfigurationPeriodes;
```

De la même manière, la zone de mémoire partagée est définie sous la forme d'une structure de données :

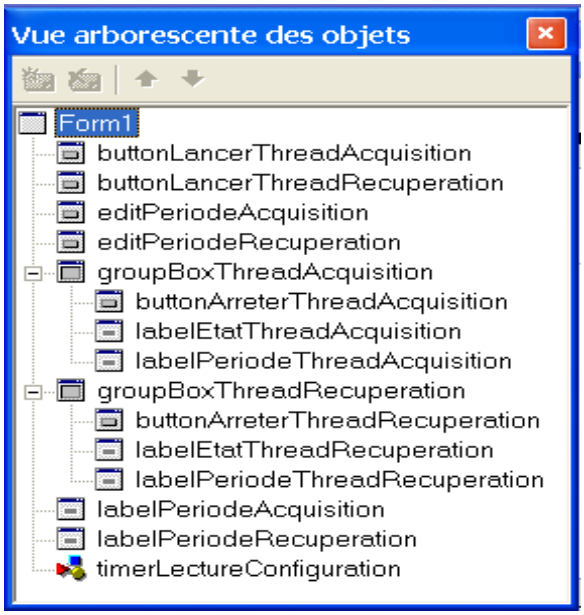
```
typedef struct
{
  int periodeConfiguration;
  int periodeAcquisition;
  bool periodeAcquisitionOk;
  int periodeRecuperation;
  bool periodeRecuperationOk;
} Periodes;
```

Annexe : Borland Builder C++

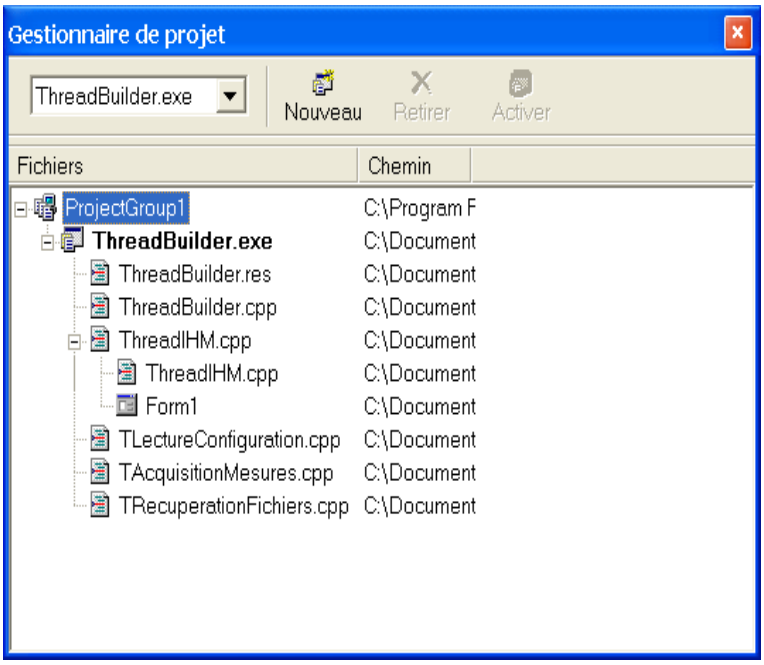
On commence par prototyper l'ihm :



Chaque composant est identifié, en conservant son type Builder en préfixe :

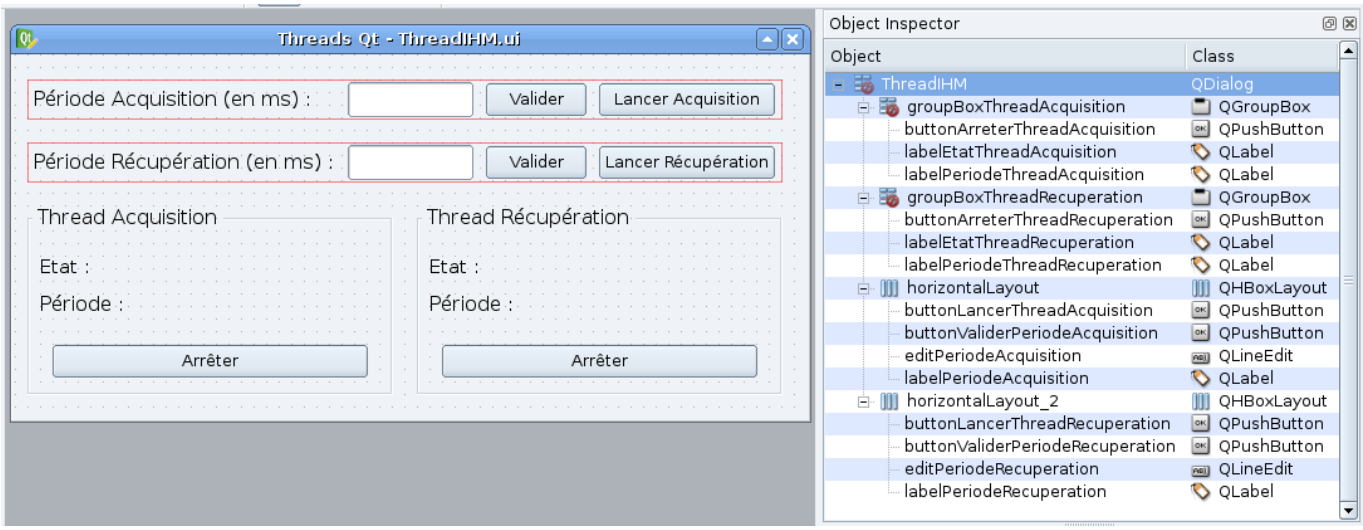


Les fichiers constituant le projet Builder :



Annexe : Qt

On commence par prototyper l'ihm où chaque composant est identifié, en conservant son type en préfixe :



Les fichiers constituant le projet sous QDevelop :