[TD-3] Classes pour la gestion du temps

Attention: les diagrammes de classes UML ne constituent qu'une vue partielle qui indique comment architecturer vos classes, mais ne représentent pas nécessairement ces classes de manière exhaustive.

a) Classe Chrono

La classe `Chrono` implémente les fonctionnalités de mesures de temps d'un chronomètre. Son interface est définie par la **figure 1**. L'implémentation des méthodes de la classe `Chrono` doit utiliser la librairie des fonctions `timespec_` développée à la **question (a) du TD-1**.

- À sa création, le chronomètre initialise son temps de démarrage `m_startTime` au temps courant par un appel à `restart()`.
- Ce temps de démarrage peut toujours être **réinitialisé** au temps courant par un appel à la méthode `restart()`.
- L'appel de `stop()` fixe la valeur de `m_stopTime` au temps courant.
- On considère que le chronomètre est actif tant que `stop()` n'a pas été appelé après un appel à `restart()`.
- On considère que le chronomètre est désactivé tant `restart()` n'a pas été appelé après un appel à `stop()`.
- Si le chronomètre est **actif**, l'appel de `lap()` renvoie le temps courant (en millisecondes) écoulé depuis le dernier appel à `restart()`.
- Si le chronomètre est **désactivé**, l'appel de `lap()` renvoie le temps (en millisecondes) écoulé entre les derniers appels à `restart()` et `stop()`.

> Implémentez la classe `Chrono` en C++ en utilisant vos fonctions`timespec_`.
> Testez votre classe par exemple en utilisant votre propre montre.

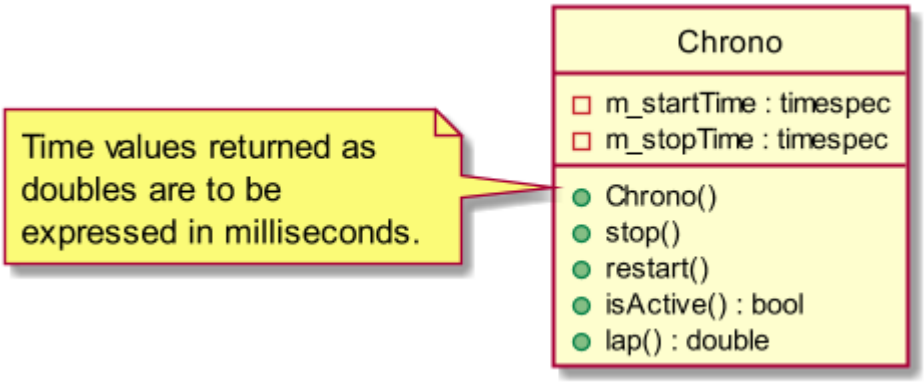

Figure 1 : Spécification de la classe Chrono.

b) Classe Timer

L'objectif est de programmer une classe `Timer` qui encapsule les fonctionnalités d'un timer Posix. La classe `Timer` doit implémenter l'interface définie par la **figure 2**. Cette classe est abstraite puisque l'opération `callback()` ne peut être implémentée au niveau de la classe `Timer` . En vous référant à la partie du cours concernant les timers POSIX, écrivez le code de la classe `Timer` en respectant les spécifications suivantes:

- Toutes les initialisations jusqu'à la création du timer POSIX doivent être effectuées dans le constructeur `Timer()`.
- Tous les objets créés dans les constructeurs doivent être détruits dans le destructeur `~Timer()`.

- Les 2 paramètres de la méthode `start()` indiquent la durée du compte à rebours du timer et si le timer doit se réarmer périodiquement.
- Si le timer est démarré, la méthode `stop()` arrête le compte à rebours du timer. S'il n'est pas démarré, l'appel de `stop()` n'a aucun effet.
- Pour chaque application, l'opération `callback()` doit être implémentée au niveau d'une classe spécifique dérivant de la classe `Timer`.
- La fonction `call_callback()` est une **fonction de classe** qui doit donc être déclarée `static` en C++. En effet, la librairie des timers POSIX impose que le handler d'un timer doit être une fonction C ayant une signature bien précise. On ne peut pas utiliser `callback()` comme handler pour les timers POSIX car toute méthode non statique d'une classe possède (en tant que fonction C) un premier paramètre implicite de type pointeur de la classe. La signature de `callback()` en tant que fonction C est `void Timer::callback(Timer* this)`. Mais si on déclare une opération comme `static`, alors, sa signature sera exactement telle que spécifiée, comme dans notre cas `void Timer::call_callback(int, siginfo_t*, void*)` qui est la signature attendue pour un handler de timer POSIX.
- La conséquence est que la fonction `call_callback()` n'a pas accès à l'objet `Timer` (pas de variable `this`), il faudra donc passer le pointeur de l'objet `Timer` en paramètre de la fonction `call_callback` en utilisant la structure `siginfo_t` dont l'adresse est passée en 2e paramètre d'un handler POSIX. Le mécanisme à utiliser est indiqué dans le cours (pointeur `myData`).

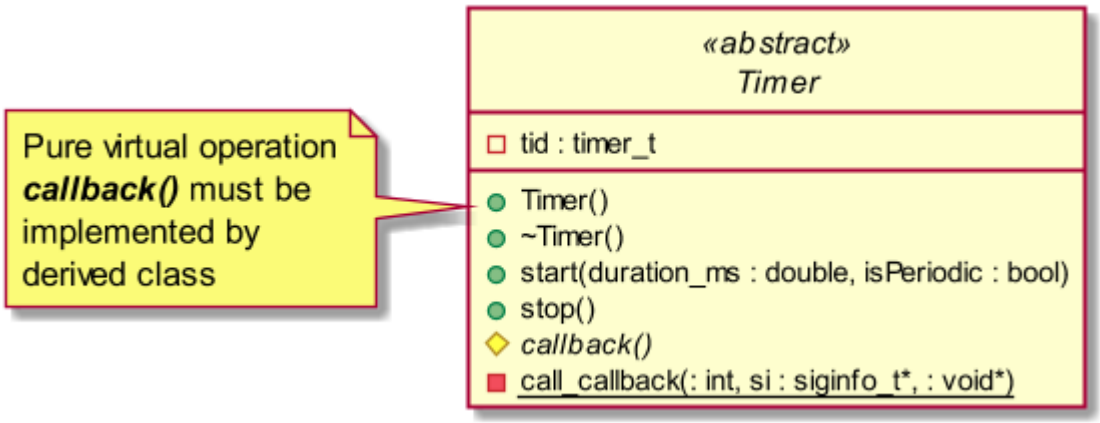

Figure 2 : Spécification des classes Timer.

> Pour chaque élément de classe de la **figure 2**, expliquez pourquoi il est public, privé ou protégé.
> Expliquez quelle est l'utilité de la méthode de classe (statique) `call_callback()`.
> Spécifiez quelles opérations doivent être définies comme virtuelles.
> Implémentez le code de la classe `Timer` en C++ .
> Testez-la en implémentant une classe dérivée `CountDown` imprimant à l'écran un compte à rebours à 1 Hz depuis un nombre `n` jusqu'à `0`.

c) Calibration en temps d'une boucle

La **figure 3** propose une architecture orientée objets pour refaire l'exercice du **TD-1** :

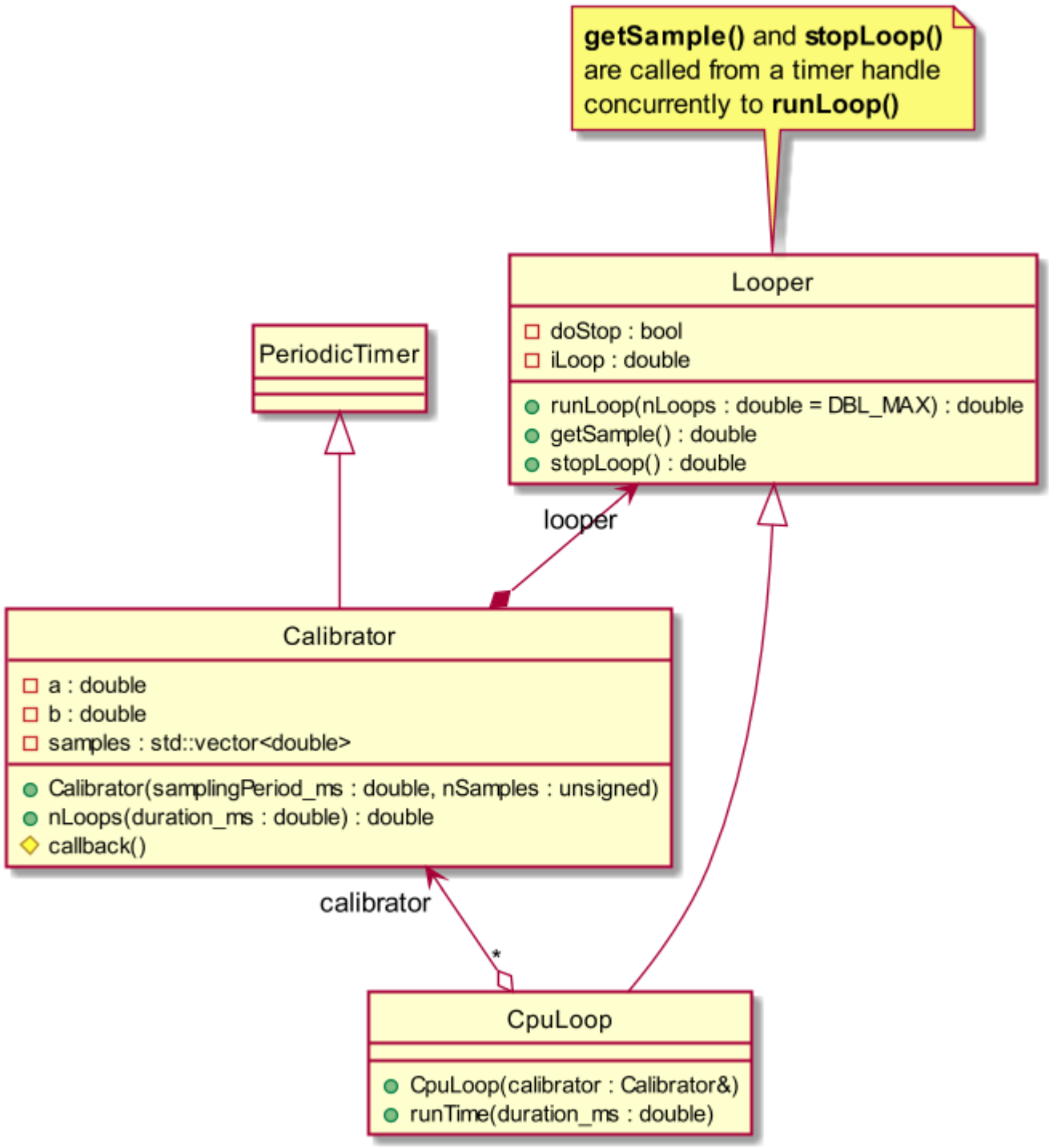

Figure 3 : Spécification des classes Looper, Calibrator et CpuLoop.

- `Calibrator` dérive de `PeriodicTimer` et implémente une méthode `callback()` lui permettant de mesurer les paramètres a et b de l'équation $l(t)=a\times t+b$ où l est le nombre de boucles effectuées par la méthode `Looper::runLoop()` pendant le temps t.
- Cette mesure doit s'effectuer dans le constructeur de `Calibrator` et utiliser les méthodes `getSample()` et `stopLoop()` d'un objet `Looper`. Normalement, il ne doit y avoir qu'une seule instance de `Calibrator` dans votre programme, utilisé par tous les objets de type `CpuLoop`.
- `Calibrator::nLoops()` est la méthode qui convertit son paramètre `duration_ms` en nombre de boucles grâce à $l(t)$.
- `CpuLoop::runTime(duration_ms : double)` fait appel à `Calibrator::nLoops()` puis appelle la méthode `runLoop` héritée de `Looper`. Chaque instance de `CpuLoop` est utilisable par une seule tâche dont on veut contrôler le temps d'exécution.

> Faites un programme analogue à celui du **TD-1** en implémentant et en testant les classes de la **figure 3**.