École Temps Réel IN2P3 – 2009
TP Programmation multitâches orientée objets

(Shebli ANVAR, shebli.anvar@cea.fr)

L'ensemble des TP utilisera l'outil Papyrus, à copier depuis le serveur sur votre répertoire utilisateur. Utilisez le raccouci « Papyrus » pour lancer le programme.

Après avoir copié le répertoire « workspace » depuis le serveur sur votre répertoire utilisateur, lancez Papyrus en choisissant ce répertoire comme espace de travail.

0. Familiarisation avec Papyrus

Ouvrez le projet Papyrus `MTOO_0`. Dans la fenêtre **Navigator** à gauche, double cliquez sur le fichier `MTOO.di2` pour ouvrir les diagrammes Papyrus. La fenêtre qui s'ouvre comporte plusieurs onglets, un par diagramme du modèle. Le diagramme par défaut (**Default diagram**), présente l'organisation du projet en packages.

La fenêtre **Outline** à gauche comprend la totalité du modèle sous forme d'arborescence d'éléments.

La fenêtre **Palette** à droite comprend les commandes pour rajouter des éléments UML. Les 2 rubriques les plus importantes sont **UML Elements** et **UML Links**.

La fenêtre en bas présente plusieurs onglets dont :

- l'onglet **Properties** qui permet d'éditer de manière complète les propriétés de l'élément séléctionné dans un diagramme ou dans le modèle ;
- l'onglet **C/C++** qui permet d'écrire de code C++ additionnel spécifique associé à chaque élément;
- l'onlglet **Console** qui affiche les sorties console de toute commande externe (exemple : une compilation C++).

Examinez le diagramme par défaut du projet

Le package `Utils` contient tous les élements utilitaires communs à l'ensemble du projet. Ainsi, la classe `Exception` constitue la classe mère de toutes les exceptions utilisées dans le projet.

Le package `Core` contient les éléments qui représentent les concepts fondamentaux de la programmation multitâches orientée objets.

Les sous-packages du package `Externals` regroupent toutes les déclarations d'éléments importés de librairies extérieures. Lorsque ces éléments sont directements issus de fichiers `*.h` externes au projet, ils ne donnent lieu à aucune génération de code.

Les liens `« use »` indiquent les dépendances entre packages. Les bonnes pratiques de modélisation et de programmation recommandent d'éviter les dépendances croisées.

- ☞ Les éléments du package `Externals` devront être utilisées par les éléments du package `Core`. Il y a déjà un lien `« use »` entre le package `Core` et le sous-package `PThread`. Rajoutez un tel lien entre le package `Core` et le sous-package `UMLPrimitiveTypes`.
- ☞ Dans le package `Core`, créez un sous-package `Mutex`.
- ☞ Sélectionnez le package `Mutex` ainsi créé et rajoutez lui le commentaire suivant à travers l'onglet **Properties** (sous-onglet **Comments**) en bas de l'écran : ***All classes related to mutexes and mutex manipulation***. Sélectionnez ce commentaire dans le modèle, à travers la fenêtre **Outline** puis faites-le apparaître dans le diagramme en effectuant un « glisser-déposer ». Adaptez la taille du cartouche du commentaire à son contenu textuel. Enfin, créez un lien en pointillés entre le cartouche et le package `Mutex`.

1. Implémentation de la classe Thread

Fermez le projet Papyrus `MTOO_0` et ouvrez `MTOO_1`. Il s'agit de programmer une classe `Thread` qui encapsule les appels de l'API multitâches POSIX et qui servira de classe de base à toute classe devant être dotée de son propre fil d'exécution. Le squelette incomplet de la classe `Thread` est fourni. Il est visible sur le diagramme **Thread classes**.

- ☞ L'attribut de thread POSIX de type `pthread_attr_t` est déjà déclaré comme un champ (Property) de la classe `Thread`. Rajoutez à travers le diagramme le champ `posixThreadID` de type `pthread_t`. En cliquant dessus et en utilisant l'onglet **Properties**, faites-en une variable protégée (protected).
- ☞ Le constructeur est l'opération portant le même nom que la classe et affectée du stéréotype `« create »`. Il est déjà presque entièrement renseigné. Le constructeur de la classe devra prendre en paramètre la priorité de la tâche, avec une valeur par défaut de `0`; en utilisant l'onglet **Properties**, rajoutez un paramètre au constructeur. Puis, en cliquant sur ce paramètre dans le modèle de la fenêtre **Outline** et en revenant à l'onglet **Properties**, nommez ce paramètre `priority` et spécifiez-lui le type `Integer`. Puis, en utilisant le sous-onglet **Profile** de l'onglet **Properties**, affectez ce paramètre du stéréotype « `CppDefault` » auquel vous donnerez la valeur 0 (priorité par défaut).
- ☞ Dans le cas où, la priorité est `0`, l'ordonnancement de la tâche devra être de type `SCHED_OTHER`, et pour des valeurs de 1 ou plus, de type `SCHED_RR` (round-robin). Le moyen le plus simple d'obtenir un thread par défaut est de lui transmettre comme attribut, soit un pointeur nul, soit l'adresse d'un attribut n'ayant reçu qu'une initialisation simple. En cliquant sur l'onglet **C/C++** puis sur le constructeur de `Thread` dans le diagramme, examinez son code et vérifiez que le code C++ du constructeur respecte bien ces spécifications.
- ☞ Le destructeur est l'opération portant le même nom que la classe et affectée du stéréotype `« destroy »`. Créez-le en ajoutant à la classe `Thread` une opération de même nom et en lui rajoutant le stéréotype `« destroy »` à travers le sous-onglet **Profile** de l'onglet **Properties**. Écrivez le code du destructeur (1 ligne) en vous référant à la documentation POSIX quant à la destruction des attributs de pthreads.
- ☞ La classe `Thread` doit comporter une opération `Thread::priority()` renvoyant la priorité de la tâche. Rajoutez cette opération sur le diagramme. En utilisant l'onglet **Properties**, rajoutez un paramètre à cette opération. Puis, en cliquant sur ce paramètre dans le modèle de la fenêtre **Outline**, nommez ce paramètre `prio`, spécifiez-lui une le type `Integer` et précisez qu'il s'agit d'une valeur de retour. En utilisant la fonction `pthread_attr_getschedparam()`, le type `struct sched_param` de l'API POSIX ainsi que le champ de type `pthread_attr_t` défini pour la classe `Thread`, écrivez les 3 lignes de code de la méthode `Thread::priority()`.
- ☞ La classe `Thread` s'utilise de la manière suivante : pour créer une tâche particulière, l'utilisateur crée une classe applicative dérivant de `Thread` et implémentant la méthode `run()`, déclarée comme abstraite dans `Thread` (`Thread::run()` est une ***méthode virtuelle pure***). La classe applicative pourra alors être instanciée et sa tâche associée démarrée en appelant l'opération `start()` héritée de `Thread`. C'est donc la méthode `run()` qui contient le code utilisateur exécuté dans la tâche ; elle devra être elle-même appelée par une ***méthode statique*** de la classe `Thread` qu'on appellera `Thread::call_run()`. Cette dernière méthode sera celle dont l'exécution devra être lancée dans une nouvelle tâche par l'appel POSIX `pthread_create()`. La création d'un objet de type `Thread` ne devra pas provoquer l'exécution de la tâche associée : ce lancement ne se fera qu'à l'appel de l'opération

`Thread::start()`, une fois l'objet instancié. Rajoutez donc les opérations `call_run()`, `start()` et `run()` à la classe `Thread`. À travers l'onglet **Properties**, précisez que `run()` est une méthode abstraite et que `call_run()` est statique. Pour que `call_run()` puisse appeler l'opération `run()`, il est nécessaire qu'on lui passe en paramètre l'instance de l'objet `Thread` : il faut donc lui rajouter un paramètre de type `Thread` ; en C++, ce paramètre devra référencer l'objet de type `Thread`, il devra donc être un pointeur de `Thread` : en utilisant le sous-onglet **Profile**, affectez-le du stéréotype `« CppPtr »`.

☞ Complétez le code C++ de `call_run()` en une seule ligne (onglet **C/C++**). Quant au code de `start()`, si l'on met de côté la gestion des erreurs, ce sera également une seule ligne consistant en un appel de `pthread_create()`. Écrivez cette ligne en mettant les bon paramètres (en vous référant à la documentation POSIX). Rajoutez à la classe `Thread`, une méthode `stop()` dont le code C++ consistera en un appel de la fonction POSIX `pthread_cancel()`.

☞ Enfin, la classe `Thread` comportera une opération `Thread::join()` permettant à une autre tâche de bloquer en attendant la fin de celle-ci (Cf. API POSIX). Rajoutez l'opération à la classe ainsi que son code C++ (1 ligne, 3 lignes si on gère la valeur de sortie de la tâche).

☞ Créez un nouveau projet de type C++ appelé `MTOO_1_Cpp`. Dans le menu **Run**, cliquez sur **Open Run Dialog**, puis sur **Papyrus generation**, puis sur l'icône « New » ; cliquez sur **New_configuration** et renommez-le **MTOO_1_Generate**. Dans la boîte de dialogue, des boutons **Browse...** permettent d'effectuer différents choix : dans la zone « Module », choisissez **Papyrus C++ Generation**, dans la zone « Configuration », choisissez comme « Source model » le modèle **\MTOO_1\MTOO.uml** et comme « Target folder » le répertoire **\ MTOO_1_Cpp**. Cliquez sur les boutons **Apply** puis **Run**. Cette commande génère le code C++ dans le projet `MTOO_1_Cpp`.

☞ À l'aide d'un clic droit sur le projet `MTOO_1_Cpp`, puis sur la commande de menu **New...**, puis **Other...**, créez un fichier source nommé `Main.cpp` puis une classe nommée `Incrementer`. Ouvrez les fichiers `Main.cpp`, `Incrementer.h` et `Incrementer.cpp` du projet `MTOO_1_Cpp`. Il s'agit de programmer une classe `Incrementer` dérivant de `Thread` et implémentant une méthode `run()` dans laquelle une boucle paramétrable (`nLoops`) incrémentera un compteur (`cnt`) partagé entre toutes les instances de `Incrementer`. Pour le comptage on définit le type `Count_t` comme équivalant à `long long` à l'aide d'une déclaration `typedef` dans `Incrementer.h`. Le constructeur de la classe `Incrementer` prendra en paramètre : un identifiant permettant de distinguer chacune de ses instances (`int id`), sa priorité en tant que tâche (`int prio`), un pointeur sur le compteur partagé (`Count_t* cnt`), le nombre d'incrémentations (`Count_t nLoops`). Codez complètement la classe `Incrementer` sachant que l'identifiant de la tâche doit être imprimée à l'écran avant le démarrage de la boucle d'incrémentation du compteur et à la fin de la tâche. Dans le fichier `Main.cpp`, programmez la fonction `main()` ainsi : déclarez le compteur partagé `cnt` initialisé à `0`, le nombre de tâches `nTasks` initialisé à `5`, le nombre de boucles `nLoops` initialisé à `1000000`. Ensuite, instanciez `nTasks` fois `Incrementer` (stockez les pointeurs dans un `vector<Incrementer*>`), puis démarrez-les, puis attendez que les 5 se terminent, puis détruisez toutes les tâches, puis imprimez la valeur du compteur. Une fois le fichier prêt, cliquez sur le bouton **Build All** du menu **Project** et exécutez le programme en utilisant à nouveau la commande **Open Run Dialog** du menu **Run**, mais cette fois-ci sur l'item **C++ local application**. Comment expliquez-vous la valeur du compteur ?

2. Implémentation de la classe Mutex

Fermez les projets `MTOO_1` et ouvrez `MTOO_2`. Il s'agit de programmer une classe `Mutex` qui

encapsule le mutex l'API POSIX.

- ☞ Dans l'onglet **Outline**, sélectionnez le package `MTOO::Core::Mutex` ; cliquez dessus avec le bouton droit de la souris ; dans le menu contextuel, cliquez sur **Add a diagram** puis sur **Create a new Class diagram** ; un nouveau diagramme se crée et s'ouvre ; renommez-le `Mutex classes` dans l'onglet **Properties** ; fermez-le (clic droit sur le diagramme, puis clic sur **Close diagram**) puis rouvrez-le (double clic sur le diagramme dans l'arborescence du modèle) pour tenir compte du changement de nom. Créez-y une nouvelle classe appelée `Mutex`. Ajoutez-lui les 2 champs (Property) protégés `posixMutexID` et `posixMutexAttr` respectivement de type `pthread_mutex_t` et `pthread_mutexattr_t` représentant un identifiant de mutex POSIX et un attribut de mutex POSIX (types définis dans le package `MTOO/Externals/PThread`).
- ☞ Rajoutez à la classe `Mutex` un contructeur et un destructeur (n'oubliez pas les stéréotypes `« create »` et `« destroy »`). Écrivez-en le code C++ (2 lignes chacun).
- ☞ Un mutex possède un jeton unique qu'une seule tâche à la fois peut posséder afin d'obtenir l'exclusivité de l'accès à une ressource. L'obtention du jeton se fait par appel à l'opération `Mutex::lock()` et le rendu du jeton par l'opération `Mutex::unlock()`. Dans un premier temps, rajoutez une version sans paramètre et sans gestion d'erreur de ces 2 opérations à la classe `Mutex` en utilisant les appels POSIX correspondants.
- ☞ Ouvrez le projet C++ `MTOO_2_Cpp` et rajoutez un objet `Mutex` partagé au contructeur et aux champs de `Incrementer`. Dans le code de `Incrementer::run()`, au cœur de la boucle, encadrez l'instruction d'incrémentation du compteur par une prise et un rendu du mutex dans le cas où le pointeur de mutex est non nul. Dans, `Main.cpp`, instanciez un mutex et transmettez son adresse aux 5 tâches, puis exécutez le programme. Que remarque-t-on concernant la valeur finale du compteur ?
- ☞ En réalité, si l'on mène la logique OO jusqu'au bout, il faut intégrer tout mutex à la structure de données (la classe) à laquelle il est associé et ne le manipuler qu'à travers les opérations de celle-ci. Dans notre exemple, la donnée est constituée par le compteur `cnt` de type `Count_t` : il faut donc remplacer celui-ci par un objet protégé. Pour cela on définit une classe `Counter` dont l'un des champs est le compteur `cnt` de type `Count_t` ***et qui dérive de la classe*** `Mutex`. La classe `Counter` doit implémenter une opération `Counter::increment()` effectuant l'incrémentation du champ `cnt` protégée par le mutex hérité ainsi qu'une opération `Counter::val()` renvoyant la valeur du champ `cnt`. Créez les fichier `.h` et `.cpp` de la classe `Counter` en utilisant les commandes de menu **New...**, **Other...**, **C++** et **class**. Modifiez le reste du code de manière à obtenir le même fonctionnement qu'avant.

3. Implémentation de la classe Lock

Fermez les projets `MTOO_2` et ouvrez `MTOO_3`. Un problème important dans les programmes multitâches est la question de la libération des mutex. La règle est que la section critique de code entre un appel de `Mutex::lock()` et `Mutex::unlock()` doit être réduite au strict minimum afin de maintenir au maximum le parallélisme entre les tâches et minimiser la durée des appels bloquants. Le problème, c'est que même en ayant une section critique petite, il est possible que le `unlock()` ne soit pas appelé en raison de l'émission d'une exception dans ce code : dans ce cas l'exécution n'arrive pas à la fin du bloc, `unlock()` n'est pas exécuté et le mutex n'est pas libéré. La solution est de recourir à une classe `Lock` dont le constructeur appelle `lock()` et le destructeur `unlock()`. Ainsi la libération du mutex est assurée automatiquement par la destruction de l'instance de `Lock`, destruction automatique, même en cas d'exception, pourvu que `Lock` soit toujours instancié sous forme de variable automatique

(pas de `new`).

- ☞ Dans le diagramme **Mutex classes**, créez une classe `Lock`, puis une association entre celle-ci et la classe `Mutex`. Cette association doit être navigable depuis `Lock` vers `Mutex` et sa cardinalité du côté de `Mutex` doit être de `1`. Générez le code C++ associé et examinez les fichiers `Lock.h` et `Lock.cpp` générés : vérifiez notamment qu'un champ `mutex` de type pointeur de `Mutex` a bien été généré dans la classe C++ `Lock`. Sous Papyrus, rajoutez à la classe `Lock` tous les éléments nécessaires pour que l'instanciation d'un objet de type `Lock` provoque la prise du mutex associé et que sa destruction provoque la libération du mutex. Après en avoir complété de code C++, procédez à la génération du code C++ complet.
- ☞ Dans `Counter.cpp`, remplacez les appels à `lock()` et `unlock()` autour de l'incrémentation du compteur par une déclaration d'un object de type `Lock` juste avant. Compilez et vérifiez que le programme s'exécute correctement. À quel moment le mutex est-il pris ? À quel moment est-il libéré ?

4. Implémentation de la classe Condition

Fermez les projets `MTOO_3` et ouvrez `MTOO_4` ainsi que le projet C++ associé `MTOO_4_Cpp`. Une « condition » est un objet destiné à bloquer une tâche dans l'attente qu'une condition se réalise. Un objet « condition » est toujours associé à un mutex en lien avec la ressource sur laquelle porte la condition. Il s'agit ici de créer une classe `Condition` encapsulant l'usage des conditions POSIX de la librairie pthread.

- ☞ Dans le diagramme **Mutex classes**, créez une classe `Condition` ***spécialisant*** la classe `Mutex` ; on dit aussi que la classe `Condition` ***dérive*** de la classe `Mutex` ou que la classe `Mutex` ***généralise*** la classe `Condition` : il faut donc créer un lien de type `Generalization` depuis `Condition` vers `Mutex`. Dans la classe `Condition`, créez l'attribut `posixCondID` de type `pthread_cond_t`. On laissera de côté les attributs de condition POSIX (de type `pthread_condattr_t`).
- ☞ Outre un constructeur et un destructeur appelant les fonctions POSIX d'initialisation et de destruction sur la variable `posixCondID`, la classe `Condition` doit comporter les 4 opérations `wait()`, `wait(int timeout_ms)`, `notify()`, et `notifyAll()`. Les méthodes `wait()` sont des appels qui bloquent la tâche appelante jusqu'à ce qu'une autre tâche effectue un `notify()` sur la condition. La méthode `wait()` avec timeout, se débloque soit parce que la condition a reçu une notification, soit parce que le temps d'attente `timeout_ms` (en millisecondes) a été dépassé. Dans le premier cas, la méthode renvoie la valeur booléenne `true`, sinon (en cas de timeout) elle renvoie `false`. Enfin, voici, à titre d'exemple, une utilisation typique d'un objet de type `Condition` entre 2 tâches distinctes :

```
/* Tâche X */
{
      Lock lock(condition);
      while (command != STOP) // Ici s'exprime la condition recherchée
      {
            condition.wait();
      }
}

/* Tâche Y */
{
      Lock lock(condition);
```

```
        command = STOP; //  Ici la condition recherchée est réalisée
        condition.notify();
}
```

Complétez la classe `Condition` ainsi que le code C++ de ses 6 opérations en utilisant la documentation POSIX. Attention, vous aurez également à utiliser le champ `posixMutexID` hérité de la classe `Mutex`, ainsi que les fonction POSIX `pthread_cond_timedwait()` et `clock_gettime()`.

☞ Générez le code C++ du modèle. Dans la déclaration de la classe `Counter`, remplacez dans un premier temps la classe `Mutex` par la classe `Condition` (`Counter` dérive maintenant de `Condition`) et vérifiez que le comportement du programme reste identique. Ensuite, juste après la boucle de lancement des threads et en utilisant l'objet `counter` en tant que `Condition`, programmez une boucle qui attende que soit réalisée la condition `counter.val() == nTaks*nLoops` et vérifiez que le programme se comporte comme prévu.

5. Programmation de sémaphores

Fermez les projets `MTOO_4` et ouvrez `MTOO_5` ainsi que le projet C++ associé `MTOO_5_Cpp`.

☞ En examinant le diagramme par défaut, vérifiez que le package `Core` contient bien un sous-package `Semaphore` (créez-le sinon). Assurez-vous également que le sous-package `Semaphore` comprend un diagramme **Semaphore classes** ; ouvrez-le. Sur ce diagramme, faites apparaître la classe `Condition` (par glisser-déposer depuis l'arborescence du modèle) puis créez une classe `Semaphore`, associée à la classe `Condition` (nom du champ : `nonEmpty`). À travers l'onglet **Properties** de l'association, assurez-vous que, côté `nonEmpty`, la cardinalité est de `1` et l'association de type (Kind) `Composition`. Cela signifie que l'allocation de l'objet `nonEmpty` est automatiquement assujettie à celle de la classe `Semaphore`. En C++, cela se traduira dans la classe `Semaphore` par une déclaration `Condition nonEmpty` au lieu de `Condition* nonEmpty`. Spécifiez également que `nonEmpty` est un champ protégé (`protected`).

☞ Un sémaphore est un objet de synchronisation doté d'un compteur représentant le nombre de « jetons » détenus par le sémaphore. Ce compteur est limité par une valeur maximale ; si cette valeur maximale est égale à 1, il s'agit d'un sémaphore binaire. Rajoutez à la classe `Semaphore` les champs `counter` et `maxCount` de type `Integer`. Faites de `counter` une variable protégée et de `maxCount` une variable publique constante (stéréotype `CppConst` dans le sous-onglet **Profiles**.

☞ Ajoutez un constructeur prenant comme paramètres `initCount` et `maxCount` représentant respectivement la valeur initiale de `counter` et la valeur du champ `maxCount`. En utilisant le sous-onglet **Profiles**, utilisez le stéréotype `CppDefault` pour spécifier des valeurs par défaut de `0` et `INT_MAX`. `INT_MAX` est une macro prédéfinie du C désignant la valeur maximale pouvant être prise par une variable de type `int`. Elle est déclarée dans le fichier d'en-tête `climits`. En utilisant l'onglet **C++** sur la classe `Semaphore`, rajoutez la directive `#include <climits>` dans le fichier d'en-tête de `Semaphore`. `maxCount` étant constant, il faudra obligatoirement l'initialiser dans le « constructor initialization list ». Dans cette liste, l'expression `y` initialise `x` selon une syntaxe du type `x(y)`. Écrivez le code C++ du constructeur en vous assurant que ni `initCount` ni `maxCount` ne soient jamais négatifs (égalisez à zéro s'ils le sont). Pour exprimer une alternative en une seule expression utilisez le trigraphe C de la forme :

```
expression booléenne ? valeur si l'expression est vraie : valeur alternative.
```

☞ Rajoutez à la classe `Semaphore` les opérations `give()`, `take()`, `take(Integer timeout_ms)` et

`flush()` : `give()` représente la donnée d'un jeton (incrémentation du compteur) ; si le compteur a atteint sa valeur maximale, l'appel de `give()` n'a aucune incidence (le sémaphore est saturé) ; l'opération `take()` représente la prise d'un jeton (décrémentation du compteur) avec bloquage si le compteur est à zéro ; `take(Integer timeout_ms)` représente la même chose avec timeout en cas de blocage : l'opération renvoie `false` si l'appel se débloque suite à un timeout, `true` sinon ; `flush()` est une opération qui débloque l'emsemble des tâches bloquées sur le sémaphore.

Écrivez les méthodes C++ de ces 4 opérations à travers l'onglet **C++**.

☞ En utilisant les classes `Semaphore` et `Thread`, ainsi que la méthode statique `Thread::sleep_ms()` (voir le diagramme **Thread classes**), écrivez un programme qui teste les fonctionnalités correspondant aux 4 opérations de la classe `Semaphore`. Par exemple, deux tâches « consommatrices » de jetons en attente sur un sémaphore, une tâche « productrice » de jetons toutes les secondes...

6. Programmation d'une file d'attente multitâches

Fermez les projets `MTOO_5` et ouvrez `MTOO_6_Cpp`. Il s'agit de spécialiser le conteneur C++ `std::queue` afin de le rendre « thread safe » c'est-à-dire accessible en parallèle par plusieurs tâches à la fois. Cette classe sera nécessaire à l'implémentation de l'objet actif.

☞ Ouvrez le fichier d'implémentation de template `ActiveFifo.hpp`. (dans le projet C++ `MTOO_6_Cpp`). Il y est déclaré la classe conteneur `ActiveFifo` paramétrée par le type de donnée contenue `DataType`. S'y trouve également un début d'implémentation de ses méthodes `push()`, `pop()` et `count()` avec une brève description pour chacune en commentaire. L'appel de `pop()` est bloquant lorsque la fifo est vide, et ce jusqu'à ce qu'une autre tâche y dépose une donnée. Étudiez la structure de la classe et complétez le code de ses trois méthodes afin de vous conformer aux descriptions.

☞ Ouvrez les fichiers `MsgSender.cpp` et `MsgSender.h`. Notez comment, à l'aide de la directive `typedef`, on a défini la classe `CounterFifo` en instantiant la classe template `ActiveFifo` avec pour paramètre `Datatype` le type `Count_t`. On spécifie la tâche de `MsgSender` comme devant envoyer `nLoops` fois le nombre `3` dans la file d'attente référencée par `cntFifo`. Après avoir examiné la déclaration de la classe `MsgSender`, implémentez son constructeur et sa fonction `run()`.

☞ Dans la fonction `main()`, créez un objet de type `CounterFifo`, puis créez et lancez `nTasks` instances de `MsgSender` (avec `nTasks = 10`), puis récupérez les `nTasks*nLoops` messages envoyés en les ajoutant à un compteur initialisé à `0`. Affichez la valeur du compteur : elle devrait être égale à `3*nTasks*nLoops`. Après cet affichage, faites un `join()` sur toutes les tâches, puis détruisez-les.

7. Implémentation d'un objet actif

Fermez le projets `MTOO_6` et ouvrez `MTOO_7` et `MTOO_7_Cpp`. Un objet actif est un objet doté d'un ou plusieurs fils d'exécution propres ainsi qu'une ou plusieurs files d'attente de requêtes d'exécution. Le but de cette architecture est de permettre l'exécution des méthodes de l'objet actif dans un autre fil d'exécution que celui de l'objet appelant. Dans notre cas, nous ne programmerons qu'un objet actif simple, doté d'un seul fil d'exécution ainsi qu'une seule file d'attente. On pourra se reporter à la dernière planche du cours (transparent n° 23).

☞ Ouvrez le diagramme **Active classes**. Faites-y apparaître, par glisser-déposer, la classe `Thread`. Créez une classe `ActiveObject` dérivant de `Thread`. Créez ensuite une classe abstraite `ExecRequest` représentant la classe mère de toutes les requêtes d'exécution,

dotée d'une méthode abstraite `execute(ActiveObject*)` ; cette classe ayant besoin d'un pointeur d'`ActiveObject` en paramètre sans avoir besoin d'en connaître d'autres détails, il suffit de rajouter dans sa section « Header include » de l'onglet C++, la déclaration :

```
class ActiveObject;
```

- ☞ La classe `RequestFifo` représente la file d'attente des requêtes d'exécution, dotée des trois opérations `push()`, `pop()` et `count()` analogues aux opérations de `ActiveFifo`, mais ne s'appliquant qu'à des références de `ExecRequest`. Comme nous allons définir la classe `RequestFifo` directement en C++ en utilisant la classe template `ActiveFifo` créée précédemment, la représentation UML de `RequestFifo` ne doit pas faire l'objet de génération de code : rajoutez-lui en conséquence le stéréotype `« CppNoCodeGen »` à travers le sous-onglet **Profiles**. Puis, en utilisant l'onglet **C++** du package `MTOO::Core::Active`, définissez en C++ la classe `RequestFifo` à travers une déclaration `typedef` (il faudra rajouter les bons `#include`), sachant qu'il s'agit d'une file d'attente s'appliquant à des objets de type pointeurs de `ExecRequest`.
- ☞ Créez une association entre `ActiveObject` et `RequestFifo`. Tout objet actif doit être doté d'une file d'attente de requêtes d'exécution propre : aussi, à travers l'onglet **Properties** de l'association, assurez-vous que, côté `RequestFifo`, la cardinalité est de `1` et l'association de type (Kind) `Composition`. Cela signifie que l'allocation de l'objet `RequestFifo` est automatiquement assujettie à celle de la classe `ActiveObject`.
- ☞ Comme toute file d'attente gère une liste de requêtes d'exécution, créez une association de cardinalité multiple entre `RequestFifo` et `ExecRequest`. La classe `ActiveObject` sera amenée à récupérer les requêtes d'exécution depuis sa file d'attente pour procéder effectivement à leur exécution dans son propre fil d'exécution ; rajoutez donc un lien `« use »` entre `ActiveObject` et `ExecRequest`.
- ☞ Écrivez le code C++ de la méthode `ActiveObject::run()`. Il s'agit d'une boucle infinie recevant des requêtes d'exécution et les exécutant (2 lignes, en ne comptant pas les accolades).
- ☞ Affichez le diagramme `Server classes` du package `MTOO::Server`. Ce package contiendra les classes applicatives correspondant au serveur de calcul vu en cours (Cf. dernier transparent). Complétez le diagramme en créant les classes applicatives conformément au dernier transparent du cours (transparent n° 23). Comme méthode `crunch()`, choisissez par exemple la multiplication de `param` par `3.14159`. Écrivez le code C++ de toutes les méthodes, générez et compilez jusqu'à ce que vous n'ayez plus d'erreurs.
- ☞ Les liens `« use »` croisés ne permettent pas de générer correctement le code C++. Pour contourner cela créez une classe `ActiveServer` dérivée de `Server`. Déplacez la méthode `req_crunch()` dans `ActiveServer` et remplacez le lien `« use » Server-->ReqServer` par un lien `ActiveServer-->ReqServer`. Enfin, créez un fichier `Main.cpp` dans lequel vous instanciez un objet `ActiveServer` et effectuez la séquence décrite dans le transparent 21 du cours. Affichez le résultat pour vous assurer du bon déroulement de la séquence.

8. Le dîner des Shadocks

Cinq Shadocks se sont réunis pour un dîner convivial. Chaque Shadock possède une place attitrée autour d'une table circulaire. Au centre de la table, il y a un grand plat contenant 1000 shg[1] de frites, met simple dont le Shadock raffole. Mais hélas ! Il n'y seulement qu'une fourchette entre deux assiettes, ce qui est un problème, car le Shadock étant peu agile, ils ne peut manger de frites que muni des deux fourchettes situées immédiatement à sa droite et à

1 Abrév. de l'unité de masse Shadock le shadogramme

sa gauche. A chaque fois qu'un Shadock réussit à se saisir de ses deux fourchettes, il parvient à ingurgiter 10 shg de frites en l'espace de 0,1 shs[2], puis repose ses fourchettes et met péniblement 0,2 shs à mâcher et avaler. Notons également que le Shadock met 0,05 shs à se saisir d'une fourchette quand celle-ci est libre !

Ces Shadocks sont des clones, programmés à l'identique. Imaginez que vous êtes le Grand Manitou Shadock : comment programmeriez-vous le Shadock pour que le dîner se finisse un jour ?

☞ Ces Shadocks sont des clones, programmés à l'identique. Imaginez que vous êtes le Grand Manitou Shadock : comment programmeriez-vous le Shadock pour que le dîner se finisse un jour ?

2 Abrév. de l'unité de temps Shadock la shadoseconde