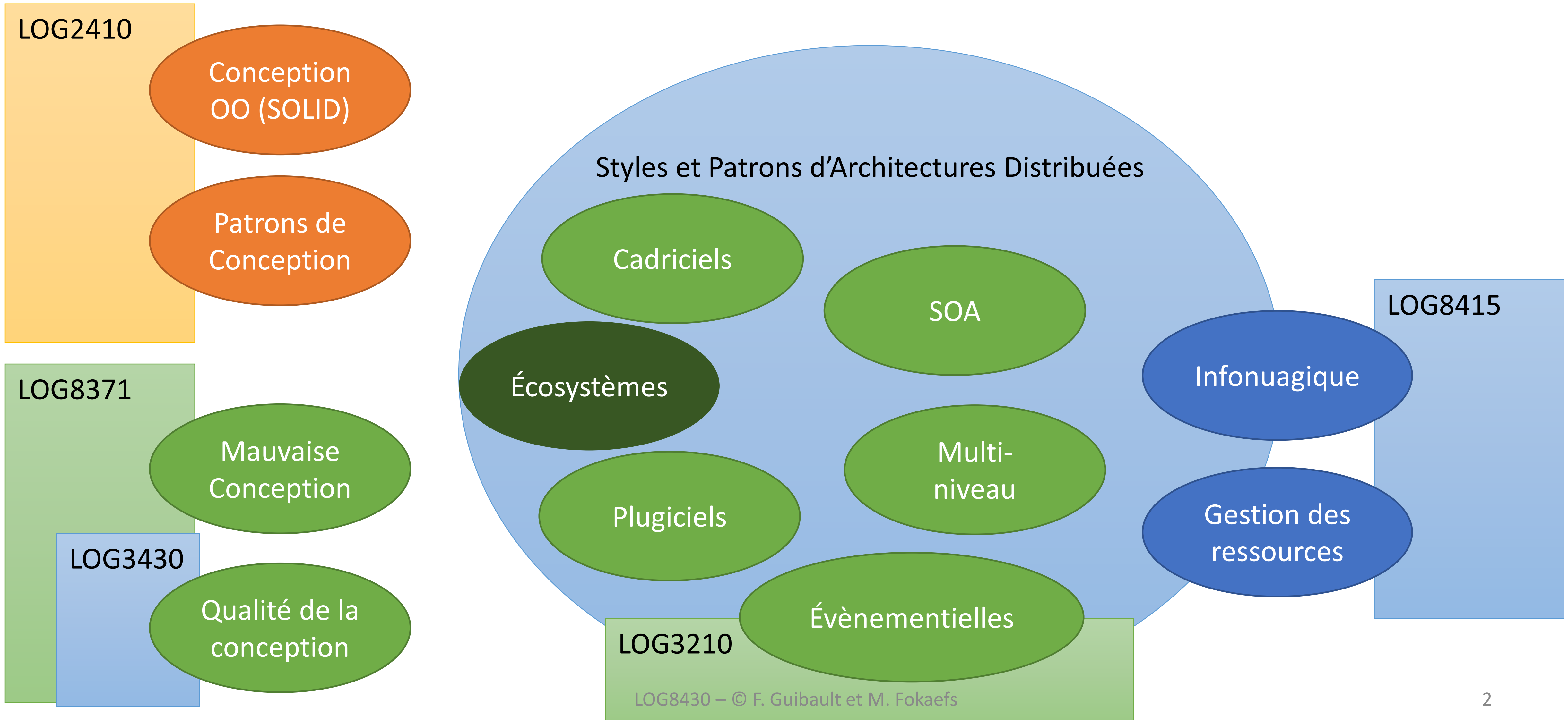
POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

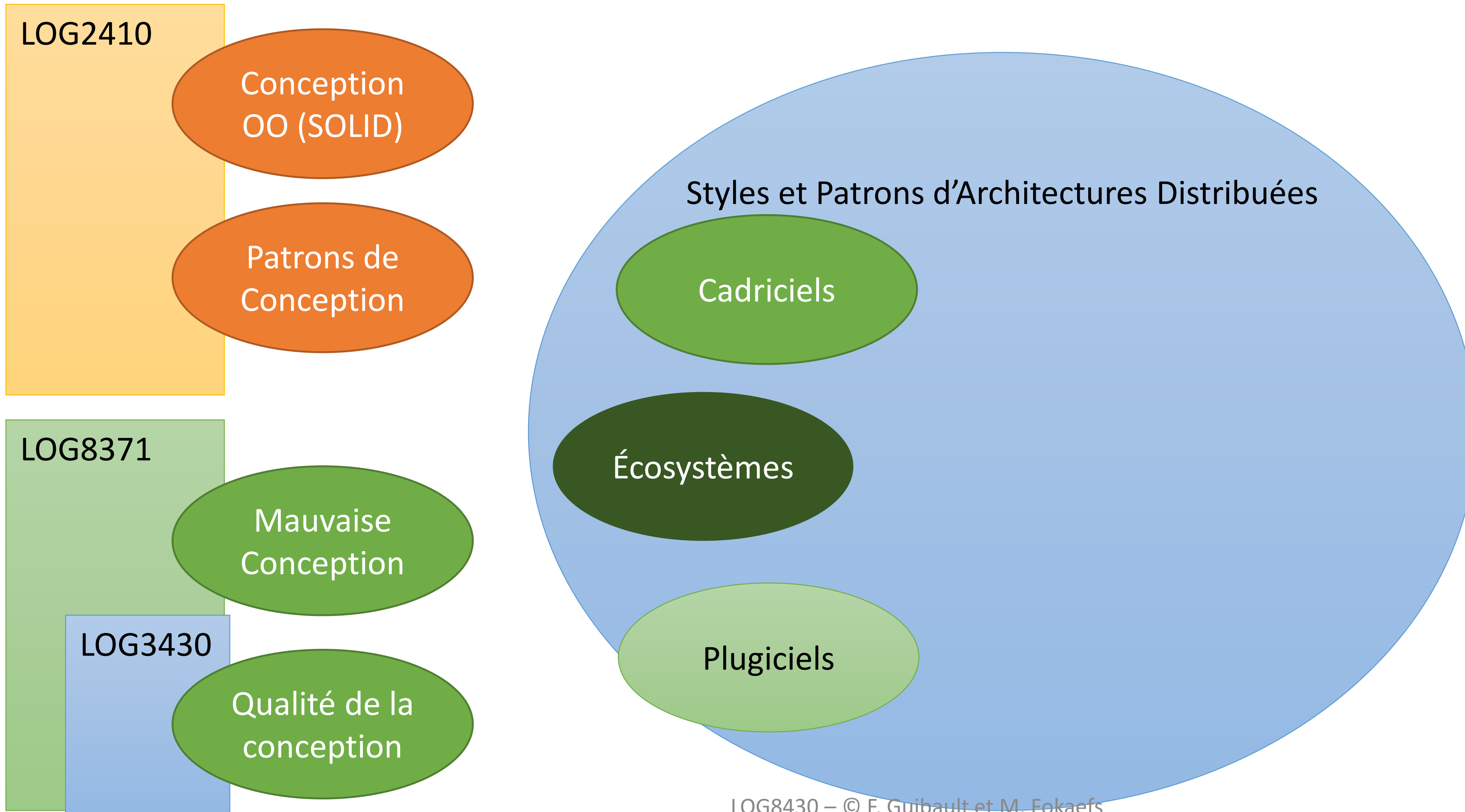
LOG8430: Architectures Évènementielles et Patrons de Concurrence

Carte du cours



POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Précédemment



Aujourd'hui

LOG2410

Conception OO (SOLID)

Patrons de Conception

LOG8371

Mauvaise Conception

LOG3430

Qualité de la conception

Styles et Patrons d'Architectures Distribuées

Cadriciels

Écosystèmes

Plugiciels

Évènementielles

LOG3210

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Scénario…

- Bienvenue à la compagnie!
- Nos produits logiciels suivent une architecture évènementielle.
- Vos responsabilités incluent le développement de nouveaux modules, la maintenance et l'évolution des modules existants.
- Le plupart des décisions sur l'architecture ont déjà été prises, mais n'hésitez pas à en prendre de nouvelles!

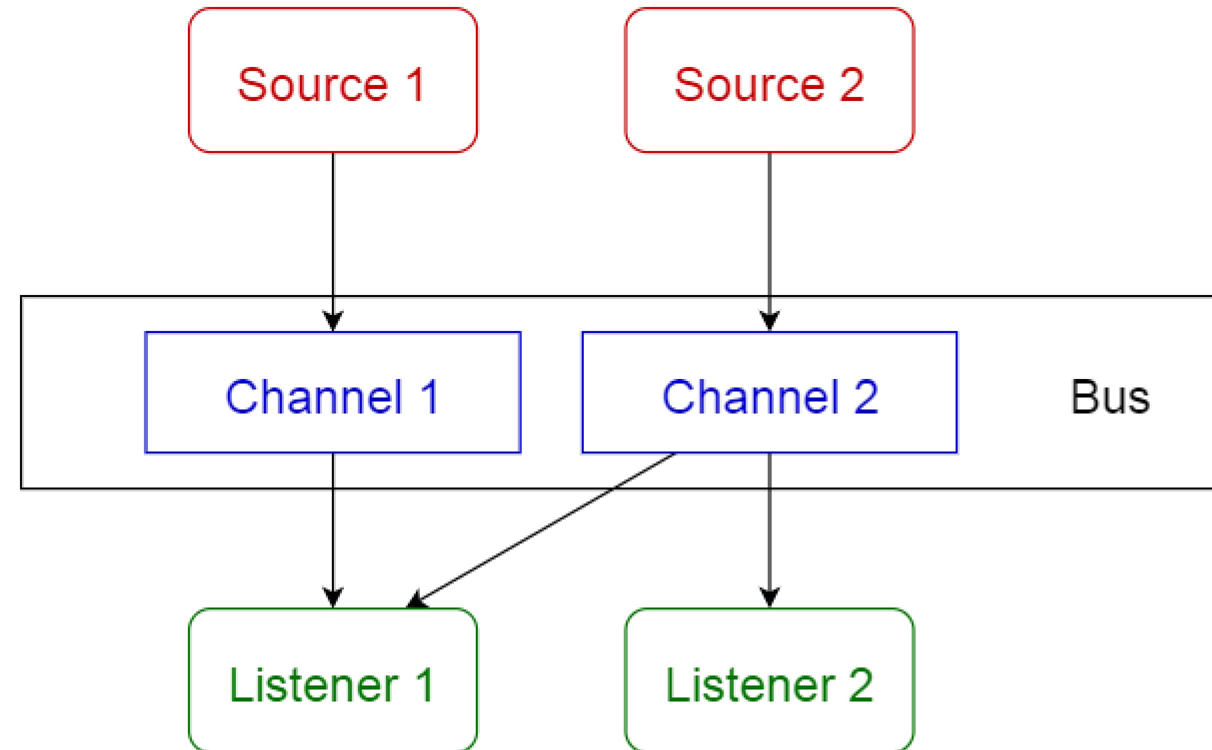
- Mais….
- Qu'est-ce que c'est une architecture évènementielle?
- Pourquoi on l'utilise et quelles sont les avantages?
- Est-ce facile à concevoir et à implémenter?
- Est-ce qu'il y a des exemples ou des patrons pour implémenter une telle architecture?

Architectures Évènementielles

- Définition
- Motivation et Défis
- Patrons de Concurrence et de Distribution

Définition

- Dans ces architectures, les évènements qui se produisent affectent l'état ou le comportement des objets.
- **Évènement** : N'importe quelle action qui est pertinente au système, p.ex. un clic dans une interface graphique, un changement des données.
- **Message** : Une notification qui suit l'évènement, produite par le système ou le producteur de l'évènement et destinée à l'auditeur des évènements. (parfois les messages sont confondus avec les évènements.)
- **Bus d'évènements** : Un canal ou un tampon où les évènements sont enregistrés, temporairement stockés et transmis aux auditeurs.



POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Propriétés

- Capacité de parallélisation augmentée.
- Le bus est commun et il peut être partagé par plusieurs producteurs et auditeurs.
- Le même évènement peut être traité par plusieurs auditeurs.
- Les auditeurs, en tant que processeurs des évènements, peuvent être répliqués pour paralléliser les mêmes taches. Cela augmente l'évolutivité du système.
- Les évènements peuvent être agnostique à la technologie, et alors on peut définir des interfaces différentes pour des langages différents. C'est la base des architectures orientées services.

Motivation

- Le matériel n'étant pas très couteux, on peut ajouter de la puissance de calcul et de traitement sans contraintes d'espace ou financières.
- Les ordinateurs modernes n'ont pas un seul CPU, mais ils fonctionnent à l'aide de CPU *multicœurs.* Cela facilite beaucoup la parallélisation.
- Les technologies de réseau ont aussi évolué. La vitesse et la capacité des réseaux ont beaucoup augmenté.
- Les applications distribuées et multifilaires sont partout aujourd'hui.
- Certains des avantages des systèmes distribués comprennent :
  - La collaboration et la connectivité : des données et des ressources distribuées sont maintenant disponibles.
  - La performance, l'évolutivité et la tolérance aux pannes sont augmentées.
  - L'économie d'échelle et les externalités de réseau améliorent la gestion des frais.

Défis

Quatre défis principaux pour l'adoption d'une architecture évènementielle.

1. Accès et configuration de service
2. Synchronisation
3. Concurrence
4. Gestion des évènements

1. Accès et configuration de service

- Les services distribués peuvent être accédés en utilisant des API à divers niveaux d'abstraction :
  - La communication inter-procédurale basée sur la mémoire partagée.
  - Les protocoles de communication tels que TELNET, FTP, SSH, etc.
  - Les appels éloignées aux méthodes en utilisant des intergiciels, comme COM+, CORBA, ou JAVA-RMI.
- L'évolution statique ou dynamique des services est un problème complexe qui exige des outils pour gérer :
  - l'évolution des interfaces et des relations entre les composantes.
  - La reconfiguration dynamique des ressources pour répondre aux changements de la charge.
- Il faut que les clients ne soient pas affectés pendant l'évolution.

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

2. Synchronisation

- Comment peut-on synchroniser correctement l'accès aux ressources partagées?
    - Acquisition et déverrouillage adéquats des verrous
    - Flexibilité dans les choix de mécanismes de synchronisation
    - Minimisation du surcoût de verrouillage et prévention de l'auto-blocage dans les appels de méthode intra-composant,
    - Réduction des conflits et des surcoûts pour les sections critiques rarement exécutées.

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

3. Concurrence

- L'exécution simultanée des fils et de processus multiples requiert la gestion des difficultés habituelles liées à la concurrence tels que :
  - Les blocages et l'état de concurrence.
  - Des API multifilaires non portables.
  - Des contradictions dans la sémantique des fils entre les systèmes d'exploitation.
- Les problèmes fondamentaux de la conception des systèmes concurrents incluent :
  - Le choix d'une architecture efficace qui minimise la surcharge.
  - La combinaison de tâches synchrones et asynchrones.
  - La sélection des primitives de synchronisation.
  - L'élimination des fils et des verrous inutiles dans les applications concurrentes en temps réel.

4. Gestion des évènements

- Trois caractéristiques importantes distinguent les applications évènementielles des applications ayant un flux de contrôle auto-dirigé :

1. Le comportement de l'application est déclenché par des évènements internes ou externes qui se produisent en façon asynchrone.
2. Le évènements doivent, pour la plupart, être traités immédiatement pour éviter la pénurie de ressources et l'augmentation du temps de réponse.
3. Les automates à état fini peuvent être nécessaires pour contrôler le traitement des évènements et détecter les transitions illégales.

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Patrons de concurrence

Service access and configuration patterns

- **Wrapper Façade**
- **Component configurator**
- **Interceptor**
- **Extension Interface**

Synchronisation patterns

- **Scoped locking**
- **Strategized locking**
- **Thread-safe interface**
- **Double-checked locking optimization**

Concurrency patterns

- **Active Object**
- **Half-Sync/Half-Async**
- Monitor Object
- Leader/Followers
- Thread-Specific Storage

Event handling patterns

- Reactor
- Proactor
- Asynchronous Completion Token
- Acceptor-Connector

Service access and configuration patterns

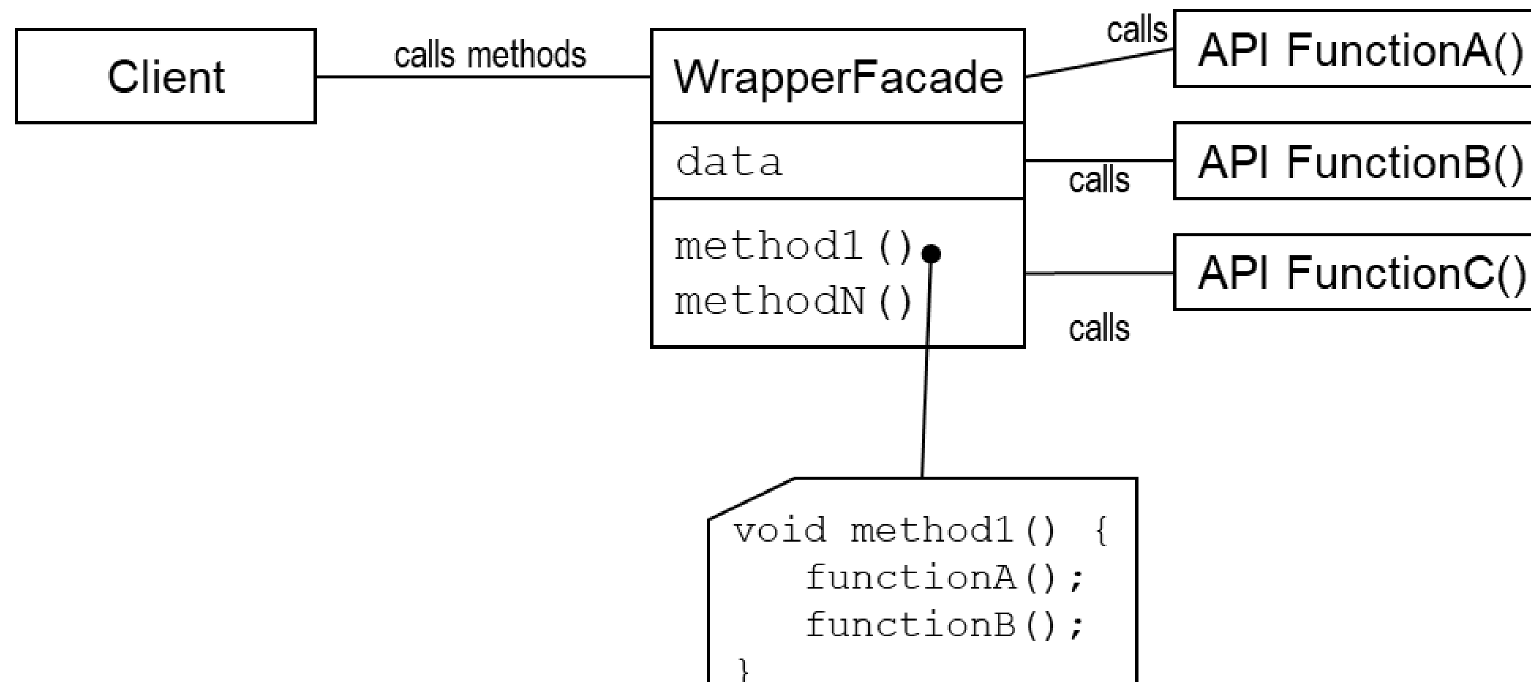
1. Wrapper Façade
2. Component configurator
3. Interceptor
4. Extension Interface

Wrapper Façade

- **Objectif** : Encapsuler les fonctionnalités et les données fournies par des bibliothèques existantes (souvent non orientées-objets) par des interfaces plus cohésives, fiables, portables et facilement maintenables.
- **Application/exemple** : Encapsuler les fonctionnalités de bas niveau (p.ex. du système d'exploitation) dans une interface de haut niveau pour les rendre plus portables et réutilisables. Voyez l'exemple d'Android.

Wrapper Façade

- **Structure** :



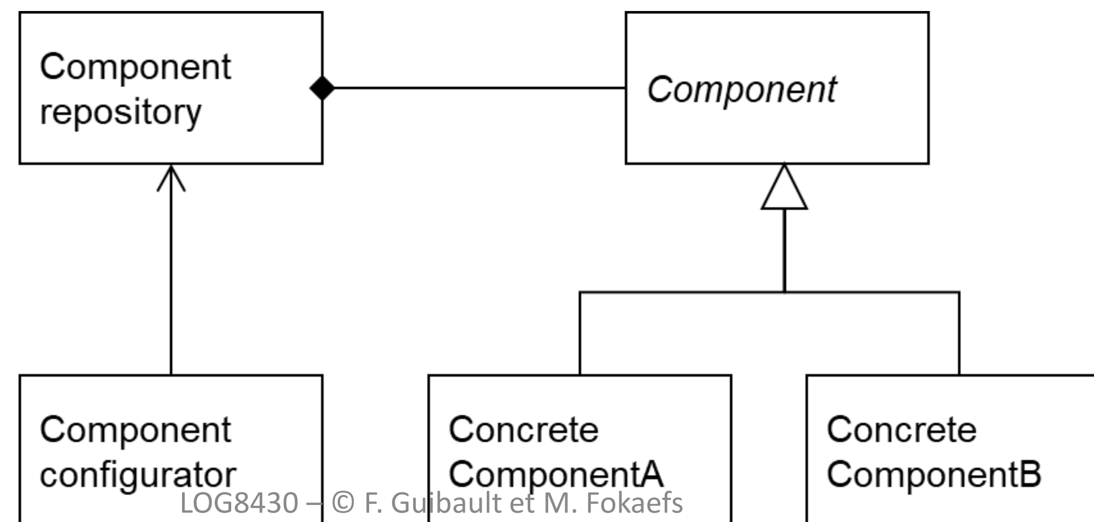
POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Wrapper Façade

- **Conséquences**
  + Des interfaces orientées-objets de haut niveau concises, cohésives et fiables.
  + Portabilité, maintenabilité, modularité et réutilisabilité augmentées.
  - Perte de fonctionnalité.
  - Performance diminuée.
  - Limitations imposées par les langages et les compilateurs.

Component configurator

- **Objectif** : Permettre à une application de lier et de dissocier les implémentations de ses composantes au moment de l'exécution en évitant de modifier, de recompiler ou de relier de façon statique l'application. En plus, le patron permet la reconfiguration de composantes dans des processus distincts sans devoir terminer et de relancer les processus actifs.
- **Application** : Le patron est appliqué lorsque on veut qu'une composante change son implémentation dynamiquement, mais en évitant de charger toutes les configurations possibles en mémoire au démarrage de l'application.
- **Structure** :

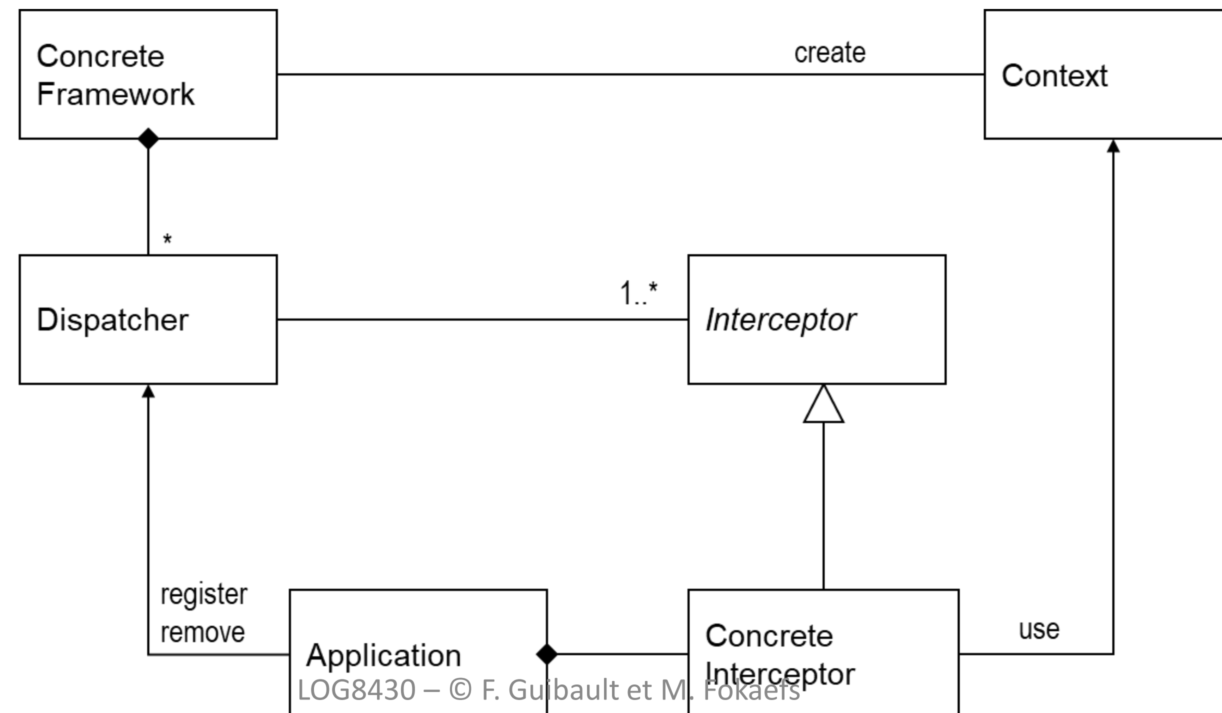


Component configurator

- **Conséquences**
  + Uniformité de la configuration et contrôle de l'interface
  + Administration centralisée
  + Modularité, testabilité et réutilisabilité augmentées
  + Configuration dynamique
  + Optimisation augmenté grâce à plusieurs possibilités de configuration
  - Accroissement de l'incertitude à cause de toutes les interactions possibles entre les différentes composantes configurées dynamiquement
  - Sécurité et fiabilité réduites
  - Complexité augmentée et performance réduite

Interceptor

- **Objectif** : Permet à des services d'être ajoutés à un cadriciel de manière transparente et d'être déclenchés automatiquement lorsque certains évènements se produisent.
- **Application** : Lorsque un cadriciel doit être capable d'enregistrer et de déclencher de nouveaux services qui n'était pas planifiés originalement. Aussi, pour permettre à des applications de contrôler le comportement et la fonctionnalité du cadriciel.
- **Structure** :

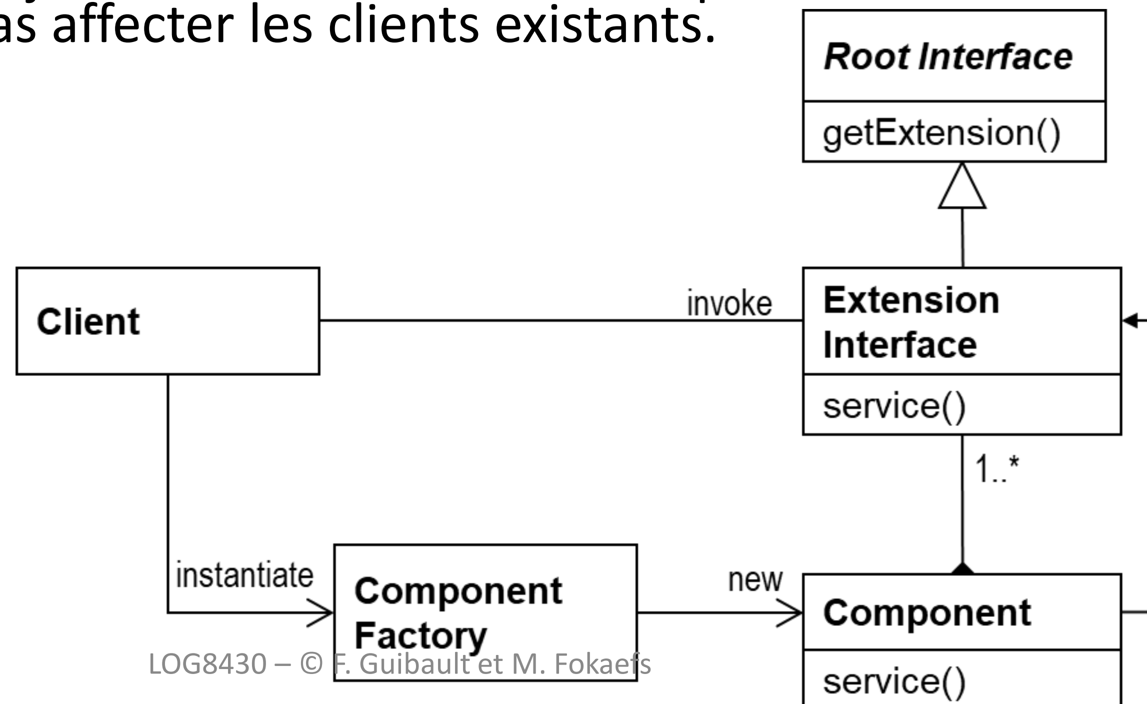


Interceptor

- **Conséquences**
  - + Extensibilité et flexibilité du cadriciel augmentées
  - + Séparation des responsabilités entre les services
  - + Capacité de monitorer et de contrôler les cadriciels
  - + Réutilisabilité des intercepteurs
  - - Difficulté à anticiper les évènements qui doivent être interceptés et à séparer les interfaces des intercepteurs
  - - Les intercepteurs sont des points d'insertions de vulnérabilités ou de fautes
  - - Possibilité d'interception en cascade à cause des changements au cadriciel

Extension Interface

- **Objectif** : Permettre à une composante d'exposer plusieurs interfaces pour garantir le principe de la ségrégation d'interface, éviter d'encombrer les interfaces et d'affecter les clients lorsque les développeurs étendent ou modifient une composante.
- **Application** : Lorsque on s'attend qu'il faudra changer les interfaces des composantes de façon impossible à anticiper après la livraison de la composante et son intégration dans des applications. Lorsque l'ajout de nouvelles interfaces pour accommoder les besoins de nouveaux clients ne doit pas affecter les clients existants.
- **Structure** :



Extension Interface

- Conséquences
  - + Fournit des mécanismes d'extension des composantes
  - + Promeut la séparation des responsabilités entre les rôles
  - + Supporte du polymorphisme entre des classes qui ne sont pas liées hiérarchiquement
  - + Découple les composantes de leurs clients
  - + Supporte l'agrégation et la délégation d'interfaces
  - - Effort augmenté pour la conception et l'implémentation des composantes
  - - Complexité des clients augmentées
  - - Indirection additionnelle et surcoût à l'exécution

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Synchronization patterns

1. Scoped locking
2. Strategized locking
3. Thread-safe interface
4. Double-checked locking optimization

Scoped locking

- **Objectif** : Assurer qu'un verrou est acquis lorsque le contrôle entre dans une portée spécifique et qu'il est libéré automatiquement lorsque celui-ci quitte la portée, quel que soit le chemin emprunté pour quitter la portée.
- **Application** : Lorsque une section critique d'une méthode doit être protégée par un mécanisme de verrouillage. Pour éviter de sortir d'une méthode par un return ou parce qu'une exception a été lancée sans libérer le verrou.
- **Problème** : Du code qui doit s'exécuter concurremment doit être protégé par un verrou qui est acquis et libéré lorsque le contrôle entre et sort d'une section critique. Si les développeurs doivent acquérir et libérer des verrous explicitement, il est difficile de s'assurer que les verrous seront libérés dans tout les chemins définis par le code.
- **Solution** : Définir une classe de garde dont le constructeur acquiert automatiquement un verrou lorsque le contrôle entre dans la portée et dont le destructeur libère automatiquement le verrou lorsque le contrôle quitte la portée. Instancier la classe de garde pour acquérir/libérer des verrous dans des portées de méthode ou de bloc définissant des sections critiques.

Scoped locking

- **Conséquences**
  - \+ Robustesse et fiabilité augmentées
  - \- Blocage potentiel lorsque le patron est utilisé récursivement
  - \- Limitations imposées par la sémantique du langage. Il assume des destructeurs, qui ne sont pas toujours appelés en cas de terminaison inattendue (exit ou abort) ou de constructions particulières du langage (p.ex. `longjmp()` en C).

Strategized locking

- **Objectif** : Paramétrer les mécanismes de synchronisation qui protègent les sections critiques du code contre des accès simultanés.
- **Application** : Lorsque un système a des composantes qui doivent s'exécuter efficacement dans une variété d'architectures concurrentes.
- **Problème** : Les composantes qui s'exécutent dans des environnements multifilaires doivent protéger leurs sections critiques contre l'accès simultané par plusieurs clients. L'intégration des mécanismes de synchronisation aux fonctionnalités des composantes doit trouver un juste équilibre entre la nécessité de respecter les besoins des différentes applications (mutex, verrous de lecture / écriture, sémaphores) et la nécessité d'éviter la duplication.
- **Solution** : Paramétrer les aspects de synchronisation d'une composante en en faisant des types enfichables (*plugins*). Chaque type représente une stratégie de synchronisation particulière. Définir des occurrences de ces types de plug-in en tant qu'objets contenus dans une composante pouvant utiliser les objets pour synchroniser efficacement ses implémentations.

Strategized locking

- **Conséquences**

  + Flexibilité et personnalisation augmentées

  + Effort réduit pour la maintenance des composantes

  + Réutilisabilité augmentée

  - Verrouillage intrusif

  - Sur ingénerie

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Thread-safe interface

- **Objectif** : Minimiser la surcharge de verrouillage et s'assurer que les appels de méthode intra-composante ne sont pas «auto-bloqués» en essayant de réacquérir un verrou déjà détenu par la composante.
- **Application** : Lorsque le code critique d'une composante qui doivent être protégées de l'accès simultané est répandu parmi plusieurs méthodes qui appellent d'autres méthodes intra-composante ou récursivement.
- **Problème** : Des composantes multifilaires contiennent parfois plusieurs méthodes publiques et privées qui peuvent changer l'état de la composante. Les méthodes peuvent s'appeler l'une l'autre pour faire leurs calculs. Dans ce cas, l'invocation des méthodes doit être conçue afin d'éviter l'auto-blocage et de minimiser le surcoût de verrouillage.
- **Solution** : Structurer toutes les composantes qui traitent les invocations intra-composante selon deux conventions de conception :
  - C'est les méthodes d'interfaces qui contrôlent les verrous.
  - Les méthodes d'implémentation font confiance aux méthodes publiques pour le contrôle des verrous.

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Thread-safe interface

- **Conséquences**
  + Robustesse, fiabilité, performance augmentées
  + Simplification du logiciel
  - Indirection additionnelle et méthodes additionnelles
  - Blocage inter-composante potentiel
  - Blocage intra-composante potentiel entre différents objets
  - Surcoût potentiel

Double-checked locking optimization

- **Objectif** : Réduire les conflits et le surcoût de synchronisation lorsque les sections critiques d'un code doivent acquérir un verrou une seule fois durant l'exécution d'un programme.
- **Application** : Lorsque une composante a une section critique qui doit être protégée contre les accès simultanés, dans le code d'initialisation, qui s'exécute une seule fois, et lorsque scoped locking n'est pas efficace.

Double-checked locking optimization

- **Problème** : Les applications concurrentes doivent s'assurer que certaines parties de leur code s'exécutent en série pour éviter les situations de concurrence lors de l'accès aux ressources partagées et de leur modification. Un moyen courant de protéger les sections critiques consiste à utiliser scoped locking, mais cela peut représenter une surcharge inacceptable lorsque le code à protéger ne doit être exécuté qu'une seule fois.

```
class Singleton {
public:
  static Singleton* instance () {
    if( instance == 0 ) {
      // Enter critical section.
      instance_ = new Singleton();
      // Leave critical section
    }
    return instance_;
  }
};
```

Double-checked locking optimization

- **Solution** : Introduire une variable booléenne indiquant s’il est nécessaire d’exécuter une section critique avant d’acquérir le verrou qui la protège. Si ce code n'a pas besoin d'être exécuté, la section critique est ignorée, ce qui évite le surcoût de verrouillage inutile.

```
// Perform first-check to evaluate 'hint'
If (first_time_in_flag is TRUE) {
  acquire the mutex
  // Perform double-check to avoid race
condition
  If (first_time_in_flag is TRUE) {
    execute critical section
    set first_time_in_flag to FALSE
  }
}
```

```
class Singleton {
public:
  static Singleton* instance () {
    if( instance == 0 ) {
      // Use Scoped Locking to acquire and
      // release lock automatically.
      Guard<Thread_Mutex> guard( singletonLck );
      // Double check
      if( instance_ == 0 )
        instance_ = new Singleton();
    }
    return instance_;
  }
};
```

Double-checked locking optimization

- **Conséquences**
  - \+ Minimiser le surcoût de verrouillage
  - \+ Prévention des situations de concurrence
  - \- L'usage de mutex additionnel
  - \- 2 problèmes potentiels liés à certaines architectures:
    - Assignation non atomique des entiers et des pointeurs
    - Cohérence des caches sur les systèmes multi-processeurs

Concurrency patterns

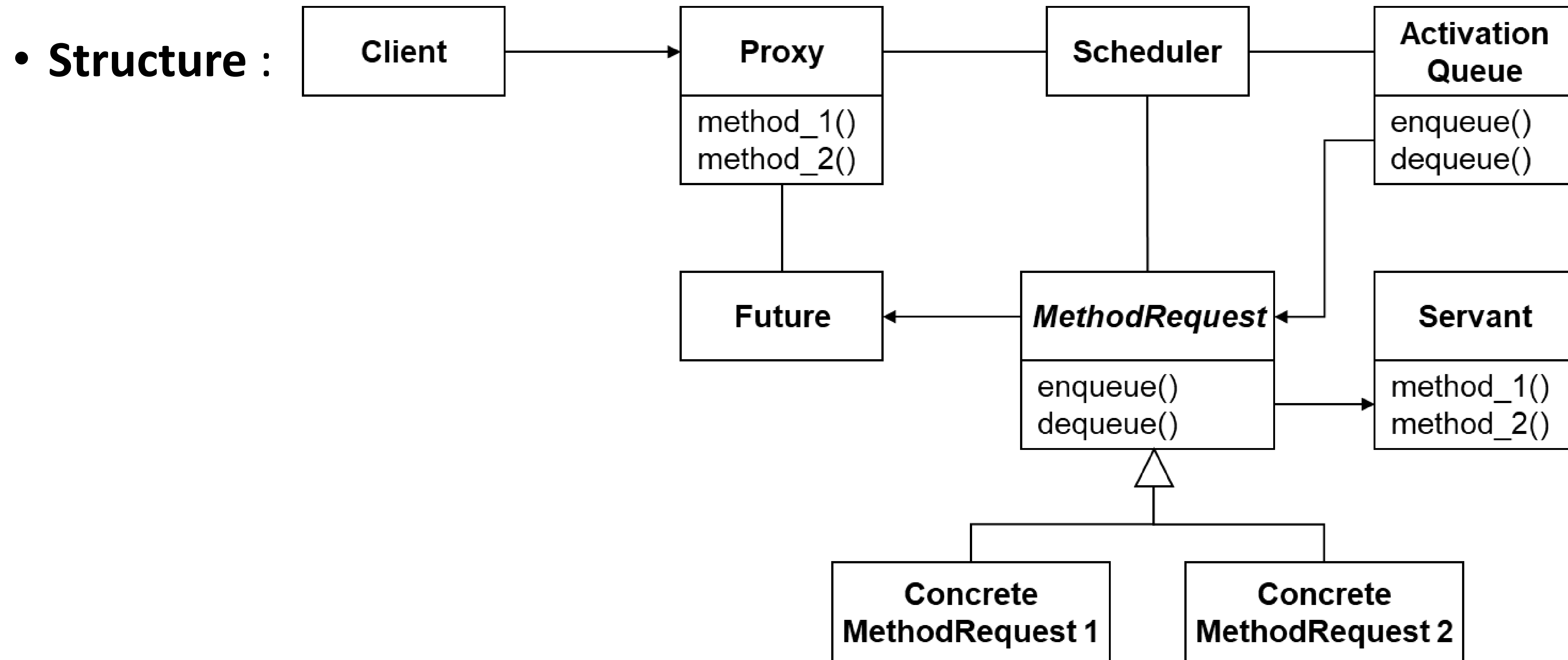
1. Active object
2. Half-sync/half-async
3. Monitor object
4. Leader/followers
5. Thread-specific storage

Active object

- **Objectif** : Découpler l'exécution de méthode de l'invocation de méthode pour améliorer la concurrence et simplifier l'accès simultané aux objets qui résident dans leurs propres fils de contrôle.
- **Application** : Pour améliorer la performance lorsque des clients accèdent à des objets s'exécutant dans des fils de contrôle séparés. Lorsque des clients ne doivent pas être liés à des détails spécifiques de synchronisation, de sérialisation ou de planification de méthodes.

Active object

- **Structure :**

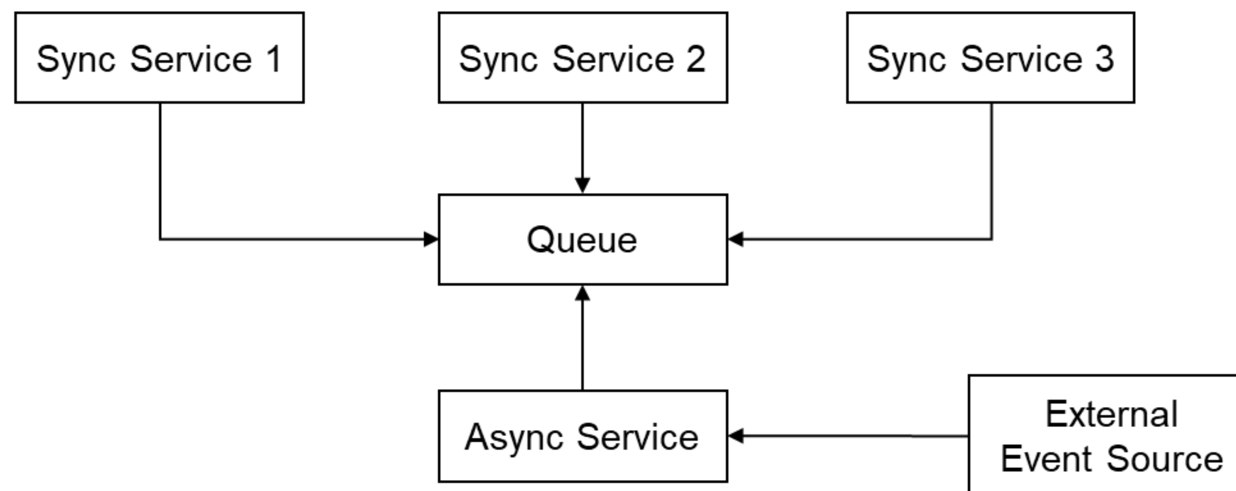


Active object

- **Conséquences**
  - + Améliore la concurrence et simplifie la complexité de la synchronisation
  - + Exploite le parallélisme disponible de manière transparente
  - + L'ordre d'exécution des méthode peut être différent de l'ordre d'invocation des méthode
  - - Surcoût de performance
  - - Complexe à déboguer

Half-sync/half-async

- **Objectif** : Découpler le traitement synchrones et asynchrones des services dans les systèmes concurrents pour simplifier le développement sans réduire la performance.
- **Application** : Lorsque un système concurrent exécute des services synchrones et asynchrones qui doivent communiquer.
- **Structure** :



Half-sync/half-async

- **Conséquences**

  + Simplification et performance

  + Séparation des responsabilités

  + Centralisation de la communication inter-couches

  - Possibilité de pénalité en traversant les frontières entre les couches

  - Les services de haut niveau ne bénéficient pas toujours de l'efficacité de l'I/O asynchrone

Monitor Object

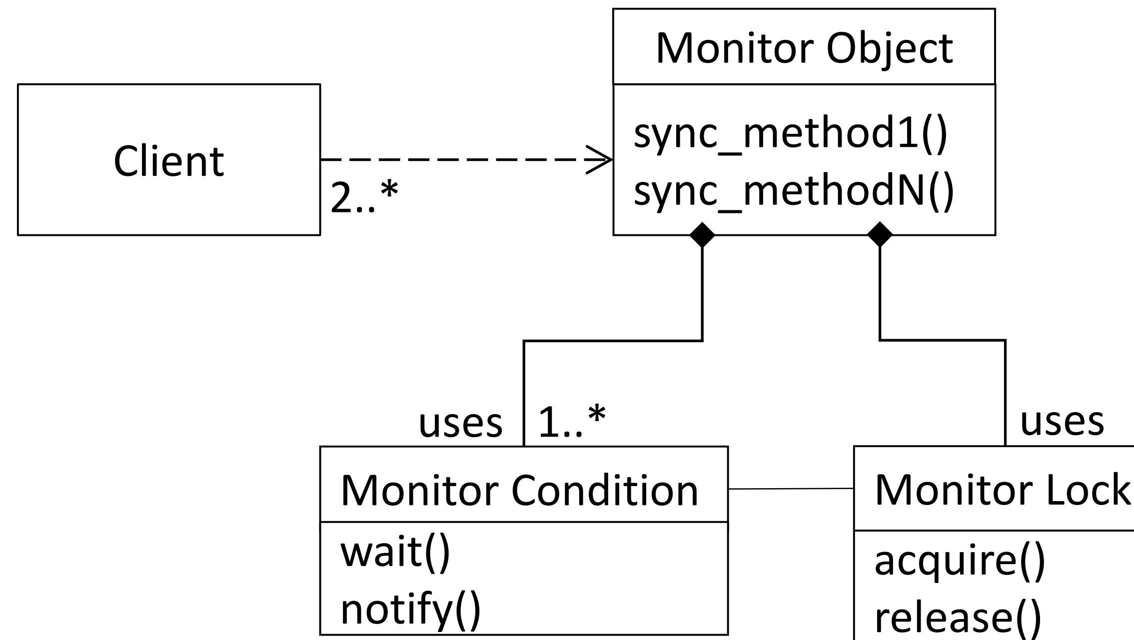
- **Objectif** : Synchroniser l'exécution de méthodes concurrentes pour s'assurer qu'une seule méthode à la fois s'exécute dans un objet. Permettre aussi aux méthodes d'un objet de planifier de façon collaborative la séquence de leur exécution.
- **Application** : Lorsque plusieurs fils d'exécution accèdent au même objet de façon concurrente.
- **Problème** : Plusieurs applications contiennent des objets dont les méthodes sont invoquées concurremment par plusieurs fils d'exécution. Ces méthodes modifient souvent l'état de leurs objets. Pour que de telles applications s'exécutent correctement il est nécessaire de synchroniser et planifier l'accès aux objets.

Monitor Object

- **Solution** : Synchroniser l'accès aux méthodes d'un objet de façon à ce qu'une seule méthode puisse s'exécuter à la fois. Chaque objet qui doit être accédé de façon concurrente par plusieurs fils clients est défini comme un Monitor Object. Les clients ne peuvent accéder aux méthodes définies par le Monitor Object seulement au travers des méthodes synchronisées. Pour éviter les conditions de concurrence sur l'état interne de l'objet seulement une méthode synchronisée à la fois peut s'exécuter dans le Monitor Object. La sérialisation est effectuée à l'aide d'un Monitor Lock. Les méthodes peuvent déterminer les situations dans lesquelles elles suspendent ou reprennent leur exécution en se basant sur des Monitor Conditions.

Monitor Object

- **Structure** :



Monitor Object

Conséquences :

+ Simplification du contrôle de la concurrence
+ Simplification de la planification de l'exécution des méthodes
- Complexité d'extension lié au couplage entre la fonctionnalité de l'objet et les mécanismes de synchronisation des méthodes
- Possibilité de verrouillage dans le cas de Monitor Objects imbriqués

Leader/Followers

- **Objectif** : Patron architectural qui fournit un modèle efficace de concurrence où plusieurs fils d'exécution partagent à tour de rôle un ensemble de sources d'évènements pour détecter, démultiplexer, soumettre et exécuter des requêtes de service qui arrivent d'une source d'évènements.
- **Application** : Dans une application basée sur la programmation événementielle où un grand nombre de requêtes de service arrivent sur un ensemble de sources d'évènements, qui doivent être traitées efficacement par plusieurs fils d'exécution qui partagent les sources d'évènements.
- **Problème** : L'utilisation de plusieurs fils d'exécution est une technique courante pour implémenter des applications qui doivent traiter de nombreux évènements de façon concurrente. Il est cependant difficile d'implémenter des serveurs haute-performance ayant plusieurs fils d'exécution. Ces applications doivent souvent traiter des grands nombres d'évènements de différents types qui arrivent tous simultanément.

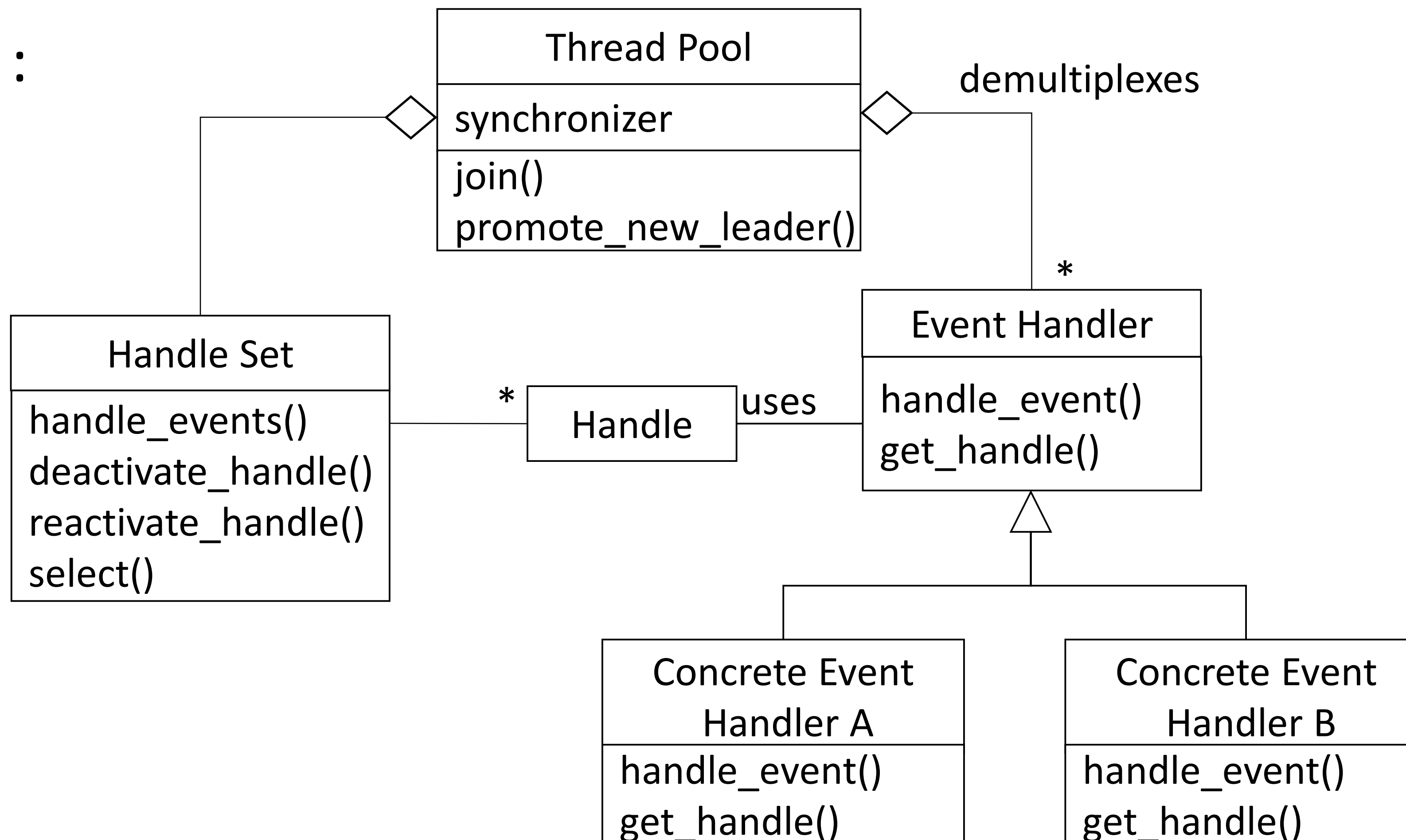
POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Leader/Followers

- **Solution** : Structurer une queue de fils d'exécution pour partager de façon efficace un ensemble de sources d'évènements et démultiplexer à tour de rôle les évènements qui arrivent de ces sources, puis envoyer de façon synchrone les évènements aux services applicatifs pour qu'ils soient traités.

Leader/Followers

- **Structure :**



POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Leader/Followers

Conséquences :

+ Amélioration de la performance:
  - Améliore l'affinité des CPUs avec la cache et élimine l'allocation dynamique et le partage de tampons entre les fils d'exécution.
  - Minimise le surcoût lié aux verrous en n'échangeant pas de données entre les fils d'exécution.
  - Permet de minimiser les inversions de priorité en éliminant les queues supplémentaires d'évènements.
  - Ne nécessite pas de changement de contexte pour traiter chaque évènement.

+ Simplification du modèle de programmation du traitement d'évènements concurrents.

- Complexité d'implémentation: la promotion des fils de leader à follower et l'inverse doit être implémenté avec des opérations atomiques.

- Manque de flexibilité: difficulté de prioriser les évènements en l'absence de queue.

Thread-Specific Storage

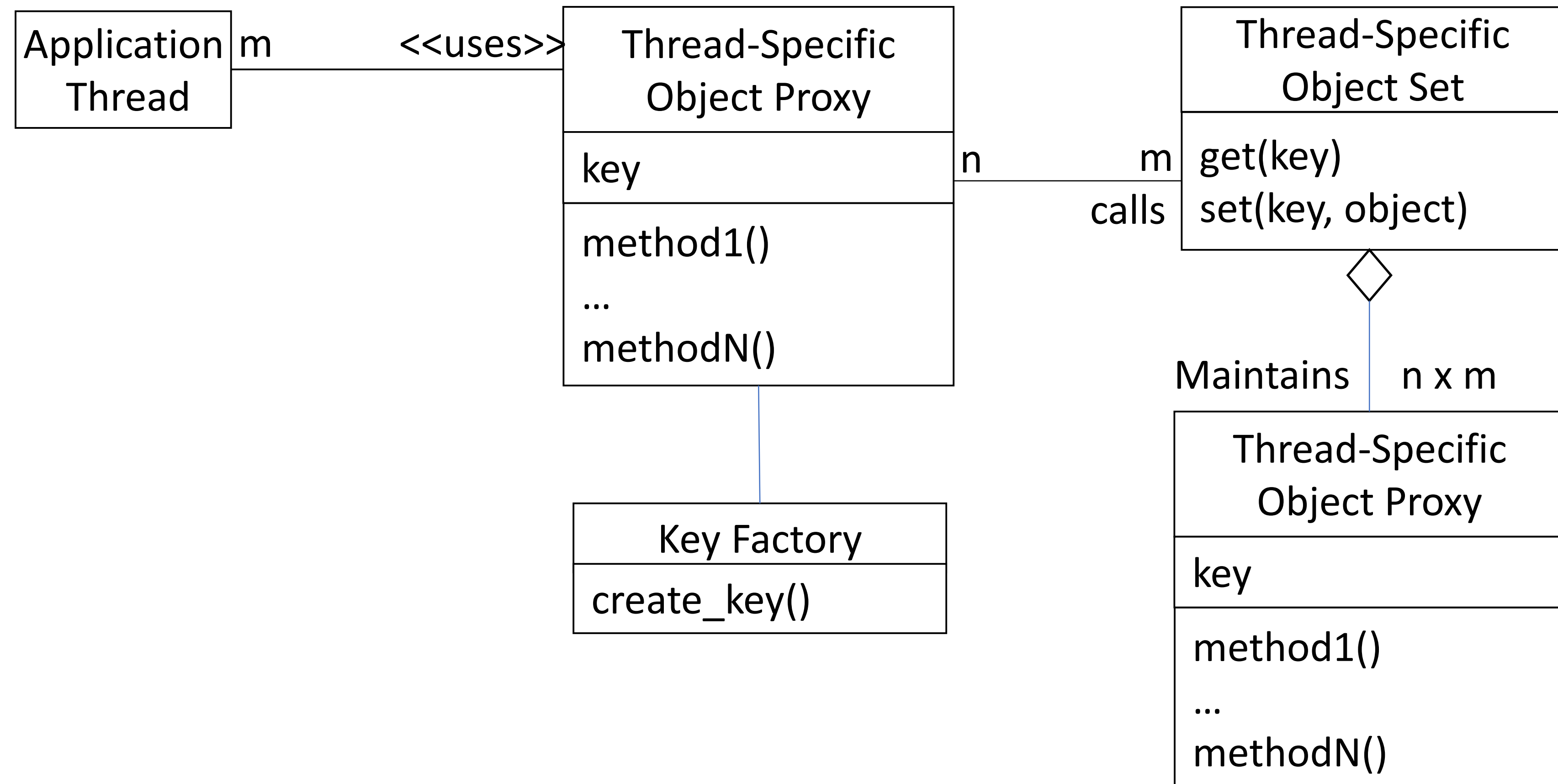
- **Objectif** : Patron de conception qui permet à plusieurs fils d'exécution d'utiliser un point d'accès « logiquement global » pour récupérer un objet local à un fil d'exécution, sans encourir un surcoût de verrouillage à chaque accès à l'objet.
- **Application** : Utilisé dans les applications ayant plusieurs fils d'exécution qui ont besoin d'accéder fréquemment à des données ou des objets qui sont stockés logiquement de façon globale mais dont l'état devrait être physiquement local à chaque fil d'exécution.
- **Problème** : À cause du surcoût associé aux verrous, la performance des applications comportant plusieurs fils d'exécution n'est souvent pas meilleure que celles des applications utilisant un seul fil.

Thread-Specific Storage

- **Solution** : Introduire un point d'accès global pour chaque objet spécifique à un fil d'exécution mais maintenir le vrai objet dans un espace de stockage local de chaque fil. S'assurer que les applications manipulent les objets spécifiques aux fils d'exécution uniquement en utilisant les points d'accès globaux.

Thread-Specific Storage

- **Structure :**



Thread-Specific Storage

Conséquences :

+ Efficacité: peut être implémenté de façon à ce que des verrous ne soient pas nécessaires pour accéder aux données spécifiques à un fil d'exécution

+ Réutilisabilité: Ce patron fournit du code qui peut être réutilisé en collaboration avec d'autres patrons comme Wrapper Façade.

+ Facilité d'utilisation: une fois encapsulé dans un Wrapper Façade, le Thread-Specific Storage est relativement facile à utiliser pour les programmeurs d'applications.

+ Portabilité: le stockage spécifique aux fils d'exécution est disponible sur la grande majorité des systèmes d'exploitation.

- Encourage l'utilisation d'objets globaux.
- Camoufle la structure du système.
- Restreint les options d'implémentation.

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Event Handling patterns

1. Reactor
2. Proactor
3. Asynchronous Completion Token
4. Acceptor-Connector

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Reactor

- **Objectif** : Le patron architectural Reactor permet à des applications événementielles démultiplexer et d'envoyer des requêtes de service qui sont soumises à une application par un ou plusieurs clients.
- **Application** : Une application événementielle qui reçoit de nombreuses requêtes de service simultanément, mais qui les traite de façon synchrone et ne série.
- **Problème** : Les applications événementielles dans un système distribué, et particulièrement les serveurs, doivent être prêtes à traiter de nombreuses requêtes simultanément, même si c'est requêtes sont ultimement traitées séquentiellement par l'application. L'arrivée de chaque requête est identifiée par un évènement spécifique d'*indication*. Avant d'exécuter séquentiellement un service spécifique, l'application doit démultiplexer et envoyer les évènements d'indication arrivant de façon concurrente aux implémentations appropriées des services.

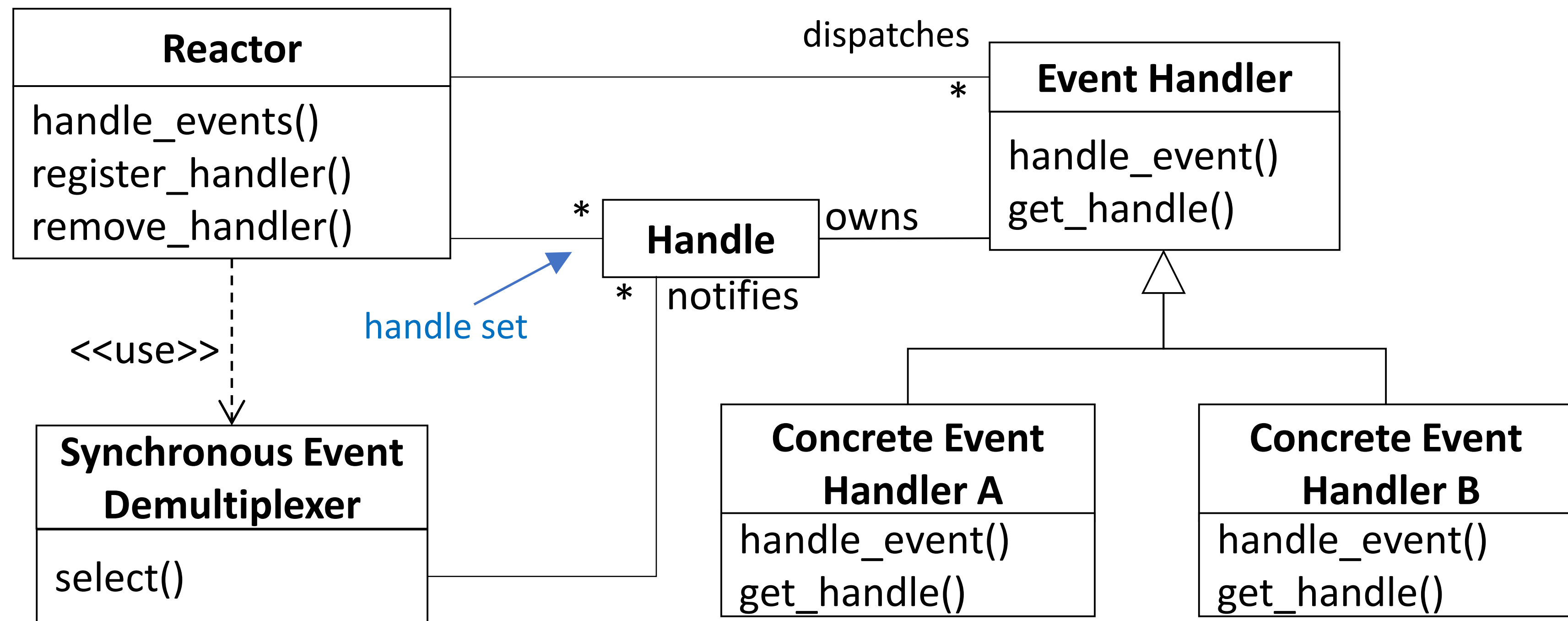
Reactor

- **Solution** : Attendre de façon synchrone l'arrivée d'évènement d'indication venant d'une ou plusieurs sources, comme p.ex. d'identificateurs de connexion réseau (*socket handle*) . Intégrer un mécanisme qui démultiplexe et envoie les évènements aux services qui doivent les traiter. Découpler ces mécanismes de démultiplixage et d'envoie des mécanismes de traitement des évènements d'indication dans les services, qui sont spécifique à une application.

  Pour chaque service qu'offre une application, introduire un gestionnaire d'évènement (*event handler*) distinct qui traite certains types d'évènements provenant de certaines sources. Les gestionnaires d'évènement s'enregistrent auprès du Reactor, qui utilise un démultiplexeur synchrone (*synchronous demultiplexer*) pour attendre qu'arrivent des évènements d'indication d'une ou de plusieurs sources. Lorsque des évènements arrivent, le démultiplexeur synchrone avertit le Reactor, qui déclenche de façon synchrone le gestionnaire d'évènement associé au service afin qu'il puisse répondre à la requête.

Reactor

- **Structure :**



Reactor

Conséquences :

+ Séparation des responsabilités:
+ Modularité, réutilisabilité et configurabilité:.
+ Portabilité:.
+ Contrôle de la concurrence à gros grain:.
- Applicabilité limitée.
- Absence de préemption.
- Complexe à débugger et tester.

Proactor

- **Objectif** : Le patron architectural Proactor permet à des applications évènementielles de démultiplexer et d'envoyer efficacement des requêtes de service déclenchées par la complétion d'opérations asynchrones, afin de bénéficier des bénéfices en performance de la concurrence sans encourir certaines de ses handicaps.
- **Application** : Une application évènementielle qui reçoit et traite de façon asynchrone de nombreuses requêtes de service.
- **Problème** : La performance des applications évènementielles, particulièrement des serveurs, dans un système distribué peut souvent être améliorée en traitant les requêtes de service de façon asynchrone. Lorsque le traitement asynchrone est complété, les applications doivent gérer les évènements de terminaison transmis par le système d'exploitation pour indiquer la fin des calculs asynchrones.

Proactor

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

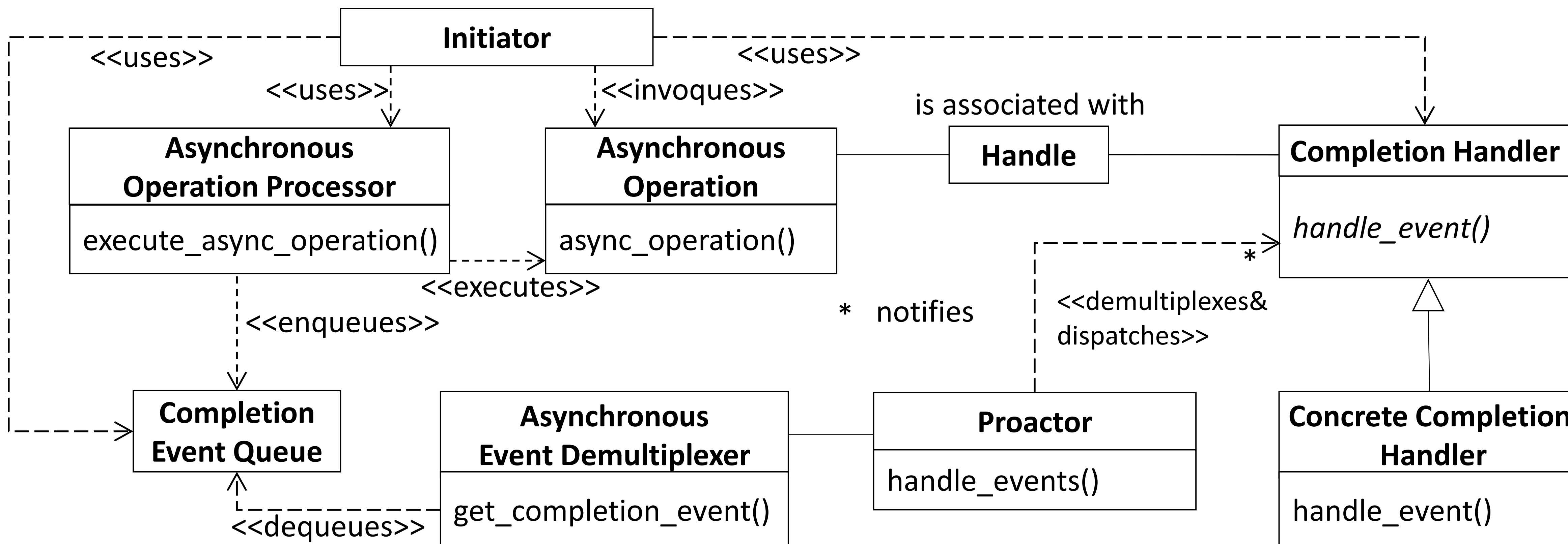
- **Solution** : Séparer les services applicatifs en deux parties: 1) les opérations longues qui s'exécutent de façon asynchrone et 2) les gestionnaire de terminaison qui traitent les résultats de ces opérations lorsqu'elles sont terminées. Intégrer le démultiplexage des évènements de terminaison, qui sont envoyés lorsque l'opération est terminée, avec le déclenchement du gestionnaire qui traite les résultats. Découpler les mécanismes de démultiplexage et de déclenchement des gestionnaires des traitement des résultats, liés à des applications spécifiques.

  Pour chaque service offert par une application, introduire des opérations asynchrones qui initient le traitement des requêtes de service de façon proactive, via un identificateur (*handle*), conjointement avec un gestionnaire de terminaison qui traite les évènements de terminaison contenant les résultats de ces opérations asynchrones. Une opération asynchrone est déclenchée, dans une application, par un initiateur pour, par exemple, accepter des requêtes de connexion entrante provenant d'applications distantes. L'opération est exécutée par un processeur d'opération asynchrone. Lorsqu'une opération se termine, le processeur d'opération asynchrone insère un évènement de terminaison contenant les résultats du traitement dans une queue d'évènements de terminaison.

  Cette queue est monitorée par une démultiplexeur asynchrone d'évènements, appelé par un Proactor. Lorsque le démultiplexeur retire une évènement de la queue, le Proactor démultiplexe et envoie l'évènement au gestionnaire de terminaison spécifique pour l'application. Ce gestionnaire traite les résultats et peut déclencher d'autres opérations asynchrones selon le même schéma.

Proactor

- **Structure :**



POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Proactor

Conséquences :

+ Séparation des responsabilités:
+ Portabilité:.
+ Encapsulation des mécanismes de concurrence:.
+ Découplage des fils d'exécution et de la concurrence:.
+ Performance:.
+ Simplification de la synchronisation des applications:.
- Applicabilité limitée.
- Complexe à débugger et tester.
- Planification, contrôle et annulation des opérations s'exécutant de façon asynchrone.

Asynchronous Completion Token (ACT)

- **Objectif** : Le patron de conception Asynchronous Completion Token (ACT) permet à une application de démultiplexer et traiter efficacement les réponses retournées par les opérations asynchrones qu'elle invoque sur des services.
- **Application** : Un système événementiel dans lequel les applications invoquent des opérations asynchrones sur des services et doivent ensuite traiter les évènements de terminaison associés.
- **Problème** : Lorsqu'une application client invoque une requête applicative sur un ou plusieurs services de façon asynchrone, chaque service retourne sa réponse à l'application par un évènement de terminaison. L'application doit alors démultiplexer les événements vers les gestionnaires appropriés (fonction ou objet) qu'elle utilise pour traiter le résultat de l'opération contenu dans l'évènement de terminaison.

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

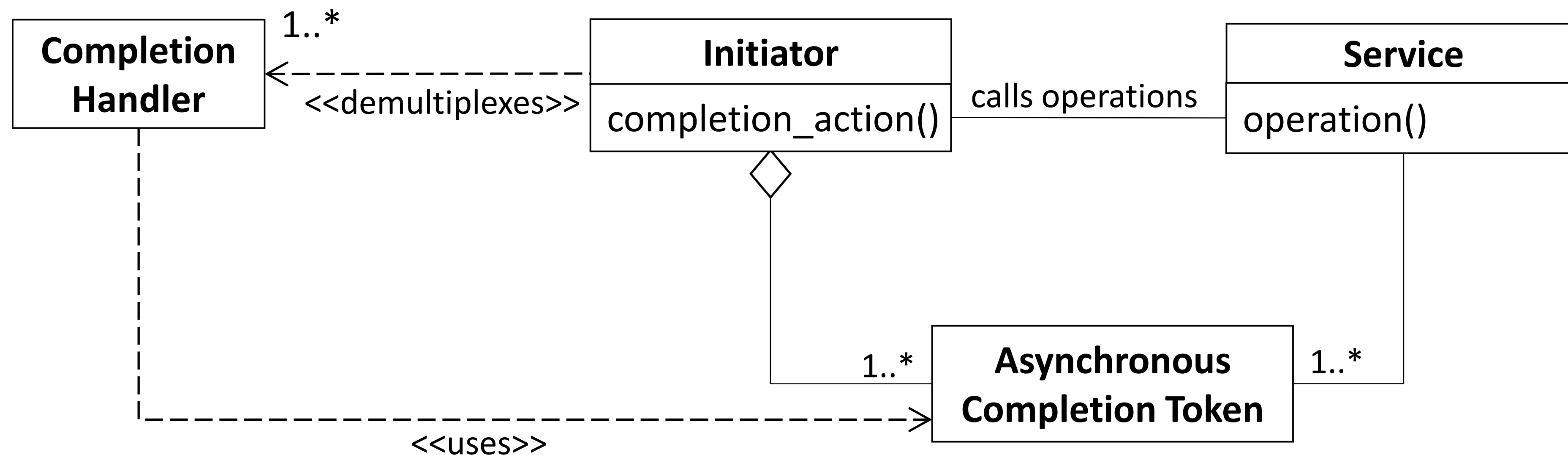
Asynchronous Completion Token (ACT)

- **Solution** : En association avec chaque opération asynchrone qu'un client initiateur invoque sur un service, transmettre de l'information qui identifie comment l'initiateur devrait traiter la réponse du service. Retourner cette information à l'initiateur lorsque l'opération est complétée afin qu'elle soit utilisée pour démultiplexer la réponse efficacement, permettant ainsi à l'initiateur de la traiter.

  Pour chaque opération asynchrone qu'un client initiateur invoque sur un service, créer un ACT. Cet ACT contient de l'information qui identifie de façon unique le gestionnaire de terminaison, qui est la fonction ou l'objet responsable de traiter la réponse de l'opération. Passer l'ACT au service avec l'opération, qui conserve mais ne modifie pas l'ACT. Lorsque le service répond à l'initiateur, la réponse inclut l'ACT. L'initiateur peut alors utiliser l'ACT pour identifier le gestionnaire de terminaison qui doit traiter la réponse.

Asynchronous Completion Token (ACT)

- **Structure :**



Asynchronous Completion Token (ACT)

Conséquences :

+ Simplification des structures de données de l'initiateur
+ Efficacité dans l'acquisition d'état
+ Efficacité en espace.
+ Flexibilité.
+ Politiques de concurrence non dictatoriales
- Possibilité de fuite de mémoire.
- Authentification.
- Invalidation suite à des déplacements en mémoire.

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Acceptor-Connector

- **Objectif** : Le patron de conception Acceptor-Connector découple la partie **connexion et initialisation** de la partie de **traitement** dans un système distribué de services pairs à pairs qui collaborent.
- **Application** : Une application ou un système distribué dans lequel des protocoles orientés-connexion sont utilisés pour communiquer entre des services pairs connectés par des points terminaux de transport.
- **Problème** : Les applications dans un système distribué orienté-connexion contiennent souvent une quantité significative de code de configuration qui établit les connexions et initialise les services. Ce code de configuration est largement indépendant du traitement effectué par les services sur les données échangées entre les points terminaux de transport. Coupler étroitement le code de configuration avec le code de traitement n'est donc pas souhaitable.

Acceptor-Connector

Solution : Découpler la partie connexion et initialisation des services pairs de la partie de traitement que ces services effectuent une fois qu'ils sont connectés.

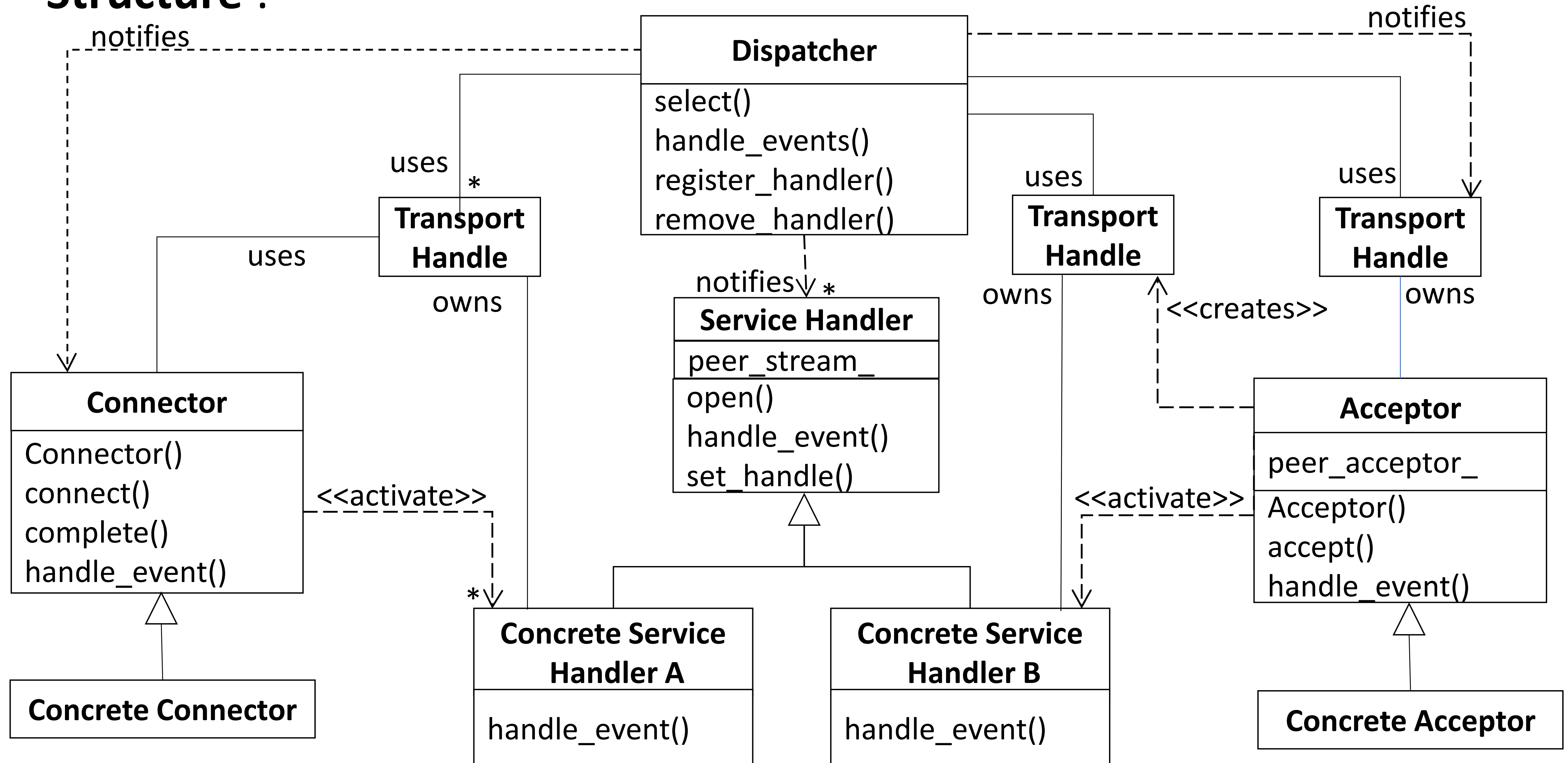
Encapsuler les services applicatifs dans des gestionnaires de services pairs. Chaque gestionnaire de service implémente la moitié d'un service point-à-point dans une application distribuée. Connecter et initialiser les gestionnaires de services pairs à l'aide de deux usines: Acceptor et Connector. Les deux usines collaborent pour créer l'association complète entre deux gestionnaires de services et les deux points terminaux de transport par lesquels ils sont connectés, chacun encapsulé dans un gestionnaire de transport.

L'usine Acceptor établit des connexions de façon passive au nom du gestionnaire de service auquel elle est associée lorsqu'arrive une requête de connexion émise par un gestionnaire de service distant. De la même façon, l'usine Connector établit une connexion de façon active vers un service distant désigné au nom du gestionnaire de service auquel elle est associée.

Une fois la connexion établie, l'Acceptor et le Connector initialise les gestionnaires de service qui leur sont associés et leur passent les identificateurs de transport. Les gestionnaires de service effectuent alors les traitements applicatifs, en utilisant les identificateurs de transport pour échanger des données au travers de la connexion. En général, les gestionnaires de service n'interagissent plus avec les usines Acceptor et Connector une fois qu'ils sont connectés et initialisés.

Acceptor-Connector

- **Structure :**



Acceptor-Connector

Conséquences :

+ Efficacité.
+ Réutilisabilité, portabilité, extensibilité.
+ Robustesse.
- Indirections supplémentaires.
- Complexité additionnelle.