

LOG2410: Conception logiciel

Travail Personnel

Rédigé par : Georges Louis : XXXXXXX

École Polytechnique de Montréal

Remarque : la référence est parfois mise à coter du paragraphe à cause du manque d’espace. Cette référence dit d’où proviennent les idées dans le paragraphe.

Tâches asynchrones et patrons de conception.

De nos jours, l’évolution rapide de nos ordinateurs qui sont insérés de plus en plus par un nombre d’unités toujours croissant à traiter et qui utilisent une architecture distribuée qui est « L’architecture d'un environnement informatique ou d'un réseau est dite distribuée quand toutes les ressources ne se trouvent pas au même endroit ou sur la même machine » (Wikipédia, s.d.) va en faveur de changer le type de processus que ces ordinateurs utilisent par des processus asynchrones. La programmation asynchrone n’est pas assez maitrisée de nos jours et cause plusieurs problèmes, entre autres l’ordre d’exécution de certaines lignes de code ne sera plus garanti, sauf dans quelques modes d’exécution, et le résultat n’est pas obtenu immédiatement. Afin de faciliter le travail des programmeurs, les langages et les logiciels intègrent de plus en plus les mécanismes qui sont responsables de gérer les taches asynchrones et la concurrence. L’asynchronisme sera expliqué plus tard, mais pour la concurrente, c’est quoi?

La concurrence est tout simplement le travail de plusieurs entités indépendantes, mais qui communiquent entre eux. Comme Sam & Max la décrivent, c’est « Si vous avez un serveur et un client, c’est de la programmation concurrente. Si vous avez un module qui s’occupe des I/O utilisateurs, un qui s’occupe de la base de données et un qui surveille le comportement de l’OS, dans des processus séparés, et qui communiquent entre eux, c’est de la programmation concurrente ». Un exemple qui illustre bien l’idée de la concurrente est présenté dans la figure 3 de l’annexe 1.

La complexité d’introduire ce nouveau concept, la programmation asynchrone, et de régler tous les problèmes qui y sont reliés, peut être facilité par ce qui est connu sous le nom de patron de conception, qui est « […] un arrangement caractéristique de modules, reconnus comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels » (Wikipédia, s.d.). Donc, il est important d’élargir un peu sur les sujets des processus asynchrones et des patrons de conception.

En premier lieu, avant de commencer à parler d’asynchronisme dans l’informatique, il est important qu’on parle d’IO parce qu’on peut dire qu’on parle d’asynchronisme si on parle de IO. Ce dernier qui est l’abréviation de Input/Output, est toute action qui comporte de faire entrer ou sortir des données dans notre programme : par exemple les IO que l’utilisateur saisi, qu’il écrive sur un terminal, qui lit d’un fichier ou bien qu’il écrive sur un disque, etc. Les IO ont les caractéristiques suivantes :

* Le temps que ça prend pour effectuer une opération ne dépend pas de la vitesse du processus, mais plutôt de plusieurs autres facteurs comme : la vitesse du réseau (si on télécharge une donnée de l’internet), la vitesse du disque, etc.
* À cause de ces plusieurs facteurs qu’on vient de mentionner, le temps que le programme prend pour effectuer une opération est inconnu.
* Parfois, c’est le IO qui prend beaucoup de temps dans l’exécution du programme. Cela est présent quand on télécharge de grandes données de l’internet.

(Sam & Max, 2013)

Cela est un inconvénient que les informaticiens doivent régler, par ce que dans le cas d’un code qui demande de télécharger le contenu d’une page et de l’afficher, la plupart des programmes figent. Un exemple qui illustre bien ce cas est représenté dans la figure 1 de l’annexe 1. Ce code, qui est écrit en python, permet de télécharger toutes les informations contenues dans tous les liens que « mille\_urls » contient et de les sauvegarder dans le tableau « contenu ». La plupart du temps de notre programme est passé en attendant l’internet de répondre, ce qui est normal parce que notre CPU est beaucoup plus rapide que l’internet.

Une façon de régler ce problème est la programmation asynchrone : ce qui consiste à laisser le programme fonctionner au lieu d’attendre les IO de finir les autres opérations, ce qui peut parfois prendre beaucoup du temps. En effet, une partie du programme qui fonctionne dans l’arrière-plan attend dans une boucle infinie toutes les opérations de IO jusqu’ils finissent. La plupart du temps c’est juste un callback, qui est « […] une fonction passée en paramètre, qui va être appelée à une condition. La condition est la plus souvent “quand ceci arrive” et “ceci” est le plus souvent “quand le traitement est terminé” » (Sam & Max, 2012). Un exemple de code écrit en Javascript illustre bien ce cas, est représenté dans la figure 2 de l’annexe 1. Le code écrit dans la figure 2 optimise le temps d’attente de téléchargement de données, car à chaque fois qu’on reçoit la contenue d’un lien, ceci est tout de suite appelé par le callback, donc on attend plusieurs choses en même temps.

La programmation asynchrone est utilisée dans plusieurs endroits, les deux plus populaires sont quand on utilise les E/S (entrées-sorties) de manière intensive, ce qui est le cas quand on télécharge des données de l’internet, et le deuxième est quand on utilise le processeur de manière intensive, ce qui est le cas quand on effectue un calcul immense dans un jeu. Dans ces deux cas, les mots clés « async » et « await » seront utilisés d’une façon abondamment, il est donc important de les comprends. Voici la définition de ces deux mots clés :

* « Le mot clé async définit une méthode comme asynchrone, ce qui vous permet d’utiliser le mot clé await dans le corps de la méthode.
* Quand le mot clé await est utilisé, il suspend la méthode d’appel et cède le contrôle à son appelant jusqu’à ce que la tâche awaited soit terminée. » (Microsoft, 2016).

La programmation asynchrone est fondée sur « Task » et « Task<T> » qui sont des objets ou des classes dans le code (comme java) servant à une modélisation du travail qui est effectué dans l’arrière-plan, qui eux-mêmes sont utilisés pour implémenter la logique du travail qui sera fait dans le plan-arrière du « thread » (Oracle, 2014). Le thread « […] ou fil d'exécution […] est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur» (Wikipédia, s.d.). Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle.Pour le cas où le code utilise les E/S intensivement, on utilise les opérations « await » dont la valeur de retour est un « Task » ou « Task<T> » dans une méthode « async ». Un exemple du code (en C#) qui utilise les opérations de « async » est représenté dans la figure 4 de l’annexe 1. Pour l’autre cas, où le code utilise le processeur intensivement, on utilise les opérations « await » qui fonctionnent dans les « threads » d’arrière-plan avec la méthode « Task.Run ». La figure 5 de l’annexe 1 nous montre un exemple du code (en C#) où on utilise le processeur intensivement. Il est donc important de savoir si le code utilise les E/S ou bien le processus d’une manière intensive, pour ne pas coder de manière incorrecte et la trainer avec nous durant tout le code. On peut se demander deux questions avant de commencer à coder qui nous permet de savoir si on utilise les E/S ou le processeur de manière intensive. Une des deux questions est la suivante : est-ce que le code doit attendre une entrée ? Si la réponse est oui, alors nous sommes dans le cas où on utilise les E/S intensivement. La deuxième question est : est-ce que le code doit faire un calcul complexe? Si la réponse est oui, alors nous sommes dans le cas où on utilise le processeur intensivement.

(Microsoft, 2016)

En second lieu, un patron de conception, qui est fait pour répondre à des problèmes de conception des logiciels, est le résultat de l’expérience des développeurs logiciels et des concepteurs logiciels. Ces derniers font face toujours à des problèmes similaires, c’est pour cela il était utile pour eux de créer ces patrons de conceptions qui décrivent en détail la procédure à suivre pour faciliter la conception d’un logiciel. Cette procédure n’est pas toujours exacte et résout le problème au complet, mais c’est plutôt une description des grandes lignes de la solution pour ensuite être modifié et adapté selon notre besoin (Wikipédia, s.d.).

Comme mentionné dans les paragraphes précédents, l’adaptation des processus qui fonctionnent de manière asynchrone dans les logiciels est toujours croissante, à cause de l’évolution de nos ordinateurs que l’on insère toujours une quantité d’informations croissante à traiter. Comme le concept de la programmation asynchrone est nouveau pour la plupart des concepteurs, développeurs et programmeurs, il est donc important que l’on aille des patrons de conceptions qui décrivent le concept de la programmation asynchrone pour faciliter la conception des classes implémentant les méthodes asynchrones.

Tous les patrons de conception ont un formalisme qui est unique et fixe et qui est le suivant :

* Le nom du patron
* La description du problème à résoudre
* La description de la solution : qui a le nom du « patron de conception » et qui a tous les éléments de la solution avec leurs relations.
* Les conséquences : les résultats issus de la solution.

(Wikipédia, s.d.)

Ce formalisme aide les utilisateurs de ces patrons à mieux comprendre leur utilisation et leur logique. Une meilleure façon de construction est l’orthogonalité où chaque patron est unique : dans son approche, qui ne contient pas les stratégies ou les idées des autres patrons (Wikipédia, s.d.).

Pour les patrons, qui sont conçus pour faciliter la conception des classes implémentant les méthodes asynchrones et qui mettent en œuvre les processus asynchrones, voici une liste d’eux :

1. Active object
2. Proactor pattern (patron proactif)

Premièrement, le patron « Active object » est un patron qui sépare l’exécution de la méthode de son invocation pour simplifier la synchronisation d’accès à un objet qui se trouve dans son propre fil de contrôle (Lavender & Schmidt, s.d.). Ce patron est connu aussi sous le nom de « Concurrent Object and Actor » et son intention est bien expliquée dans sa définition. Pour illustrer le concept du patron « Active object », on prend l’exemple de la figure 6 qui est l’exemple de la « Communication Gateway ». Dans cet exemple la communication est faite avec le TCP qui est « a set of rules that governs the delivery of data over the Internet or other network that uses the Internet Protocol, and sets up a connection between the sending and receiving computers » (Google Dictionary, s.d.). Le TCP utilise le contrôle de flux pour s’assurer que les fournisseurs rapides ou Gateway ne produisent pas de données plus rapidement que les consommateurs lents ou les réseaux encombrés. Alors pour résoudre ce problème, le Gateway ne doit pas être bloqué en attente du contrôle du flux qui va diminuer la connexion d’un autre consommateur, mais plutôt le Gateway doit être capable d'évoluer efficacement comme le nombre de fournisseurs et les consommateurs augmentent. Une façon pour éviter le blocage et d’évoluer le Gateway est d’introduire le concept de la concurrence, que nous avons expliquée avant.

(Lavender & Schmidt, s.d.)

L’« Active object » utilise le concept de concurrence où les objets sont exécutés dans leur thread de contrôle, contrairement à la version de « single threaded » qui exécute les méthodes dans le thread de contrôle du client qui les a invoqués. Mais, pour utiliser le concept de concurrence dans les objets, nous devons synchroniser l’accès à leurs méthodes et données si ces objets sont partagés par plusieurs utilisateurs. Trois grands problèmes émergents de ce fait et qui sont les suivants :

1. Les méthodes invoquées dans l’« Active object » qui utilise la concurrence ne doivent pas bloquer tout le processus afin d'éviter de dégrader la qualité de service des autres méthodes.
2. La synchronisation d’accès de tous les objets doit être simple. En général, les méthodes qui sont soumises à des contraintes de synchronisation doivent être sérialisés de manière transparente lorsqu'un objet est accédé par plusieurs clients threads, ce qui est difficile à programmer.
3. Les applications doivent être conçues de façon transparente à favoriser le parallélisme disponible sur un « hardware or software platform ».

(Lavender & Schmidt, s.d.)

Tous ces problèmes mentionner dans l’« Active object » peuvent être réglé simplement par : pour tous les objets qui utilise le concept de la concurrence, il faut séparer l’exécution de la méthode de son invocation.  
Ce séparage est conçu pour que le thread du client apparaisse qui invoque une méthode ordinaire. Mais, cette méthode est convertie en une méthode de demande d’objet, qui est transmise à un autre thread de contrôle, où il est reconverti en une méthode et exécuté sur l'implémentation de l'objet (Lavender & Schmidt, s.d.).

L’« Active object » fournit plusieurs avantages qui sont les suivants :

* Améliorer la concurrence des applications et simplifier la complexité de synchronisation. La concurrence est améliorée en exécutant d’une façon simultanée les threads du client et les méthodes asynchrones. La simplification de la complexité de synchronisation est faite par le « Scheduler », qui « […] runs in a different thread than its clients, managing an Activation Queue of Method Requests that are pending execution. » (Lavender & Schmidt, s.d.).
* Exploiter de manière transparente le parallélisme disponible : Ce patron permet l’exécution de plusieurs objets actifs en parallèle en respectant les contraintes de synchronisation, si le « hardware or software platform » peut soutenir plusieurs processeurs efficacement.
* L'ordre d'exécution de la méthode peut différer de l’ordre de l'invocation de la méthode: Les méthodes invoquées de manière asynchrone sont exécutées en fonction de leurs contraintes de synchronisation, qui peuvent différer de leur ordre d'invocation.

(Lavender & Schmidt, s.d.)

Ce patron a aussi des limites qui sont les suivantes :

* Frais généraux de performance : En général, le modèle d'objet actif est le plus applicable sur des objets relativement grands. Mais pour les objets petits, les frais de performance peuvent être excessifs.
* Débogage compliqué : il peut être difficile de déboguer des programmes contenant des objets actifs en raison de la concurrence et non déterminisme du « Scheduler ».

(Lavender & Schmidt, s.d.)

Deuxièmement, le patron « Proactor » est un patron qui décrit comment structurer des applications et des systèmes qui utilisent efficacement les mécanismes asynchrones qui sont pris en charge par les systèmes d’exploitation (Pyarali & Harrison & Schmidt & Jordan, s.d.). L’intention de ce patron est de prendre en charge le démultiplexage et l’envoi de gestionnaires d’événements multiples, qui sont déclenchés par l’achèvement d’événements asynchrones. Ce modèle simplifie développement d'applications asynchrones en intégrant le démultiplexage des événements d'achèvement et la répartition de leurs gestionnaires d'événements correspondants (Pyarali & Harrison & Schmidt & Jordan, s.d.). Pour illustrer le concept du patron « Proactive » et savoir où l’utiliser pour profiter de ses avantages on prend l’exemple de la figure 8 qui est l’exemple d’un « Web server communication software architecture ». Cet exemple illustre bien les avantages du patron « Proactive » parce qu’on utilise une application qui utilise un réseau et qui effectue plusieurs opérations simultanément. Dans notre exemple le « Web server » doit procéder, d’une manière concurrente, les demandes de « HTTP » qui sont faits par plusieurs utilisateurs. La figure 8 illustre bien l’interaction entre le « Web browser » et « Web server ». Quand l’utilisateur veut ouvrir une « URL » sur un navigateur, ce dernier envoie une demande « HTTP » de type « GET » au « Web server », et quand cette demande est reçue, le serveur analyse et valide la demande et renvoie les fichiers spécifiés au navigateur.

(Pyarali & Harrison & Schmidt & Jordan, s.d.)

L’utilisation du patron « Proactive » a des avantages comme des inconvénients. Les avantages de l’utilisation de ce patron sont nombreux, parmi ces avantages on verra les plus importantes qui sont les suivants:

* Amélioration de la portabilité de la logique applicative : En permettant à son interface d’être réutilisée indépendamment des appels de système d’exploitation sous-jacents qui effectuent un événement démultiplexage.
* Augmentation de la séparation des préoccupations : En découplant les mécanismes d’asynchronisme qui sont indépendants de l’application, de la fonctionnalité spécifique à l’application. Ce qui rend les mécanismes indépendants réutilisables qui sachent comment démultiplexer les événements d’achèvement associés avec des opérations asynchrones. L’encapsulation des mécanismes de concurrence : Un avantage du découplage « Completion Dispatcher » de l’« Asynchronous Operation Processor » est que les applications peuvent configurer « Completion Dispatchers » avec plusieurs accès simultanés stratégiques sans affecter les autres participants.
* Simplification de la synchronisation des applications : Tant que le « Completion Handlers » n’invoque pas des threads de contrôle, la logique d’application peut être écrite avec peu de problèmes de synchronisation.

(Pyarali & Harrison & Schmidt & Jordan, s.d.)

Ce patron a aussi des limites qui sont les suivantes :

* La difficulté de débogage : Les applications écrites avec le patron « Proactor » peuvent être difficile à déboguer parce que le flux de contrôle oscille entre l’infrastructure du « framework » et les méthodes « callbacks » mentionner en détail dans les paragraphes précédents.
* L’ordre d’exécution des opérations n’est plus respecté : Le « Proactive Initiators », qui est « […] any entity in the application that initiates an Asynchronous Operation » (Pyarali & Harrison & Schmidt & Jordan, s.d.), n’a plus le contrôle sur l’ordre d’exécution des opérations asynchrones.

(Pyarali & Harrison & Schmidt & Jordan, s.d.)

Enfin, la programmation asynchrone est un concept assez populaire de nos jours et c’est un concept qui vit une adaptation croissante par nos programmeurs et concepteurs logiciels afin de faciliter le traitement des données dans nos ordinateurs qui sont toujours croissants. C’est un type de programmation qui fait plusieurs choses en même temps et qui évite le blocage de nos processeurs par plusieurs facteurs comme l’internet. Ce fait n’est pas pareil, mais il a plusieurs formes dépendamment du contexte, si c’est IO alors c’est asynchrone, si le travail est réparti en plusieurs bouts alors c’est parallèle et s’il y a plusieurs entités indépendantes alors c’est concurrent.   
Je pense, après avoir vu le concept de la programmation asynchrone et où nous nous dirigeons dans le futur vers une société de AI, qui est l’abréviation d’artificial intelligence et qui veut dire « the theory and development of computer systems able to perform tasks that normally require human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages » (Google Dictionary, s.d.) que la programmation asynchrone doit être adopté de plus en plus par nos programmeurs et concepteurs pour s’évoluer plus rapidement vers notre cible en évitant tout genre de problèmes, plus précisément le temps d’exécution des opérations qui est plus rapide dans la programmation asynchrone « multithreaded » que la programmation synchrone.  
Pendant des décennies, l'humanité a imaginé un avenir avec des machines intelligentes: véhicules autonomes, assistants personnels, usines intelligentes, robots qui rendront nos vies plus faciles, plus sûres et meilleures, etc. Les développements récents transforment ce fantasme en réalité. L'IA (intelligence artificielle) commence à transformer plusieurs industries telles que la finance, la santé, le commerce de détail, le transport, les médias, etc... (Future of AI, 2018), est-ce qu’on peut atteindre ce but et le potentiel maximum de l’IA en utilisant la programmation asynchrone? Est-ce que les contraintes de la programmation asynchrone vont affecter l’atténuation du potentiel maximale du IA? Tout cela reste à vérifier.

**Annexe 1 : Code**

Figure 1 : télécharge et sauvegarde dans une liste le contenu de chacune des 1000 urls (Sam & Max, 2013).

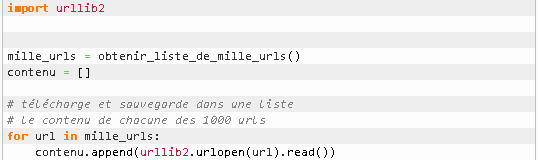
****

Figure 2 : télécharge et sauvegarde dans une liste le contenu de chacune des 1000 urls d’une façon asynchrone (Sam & Max, 2013).

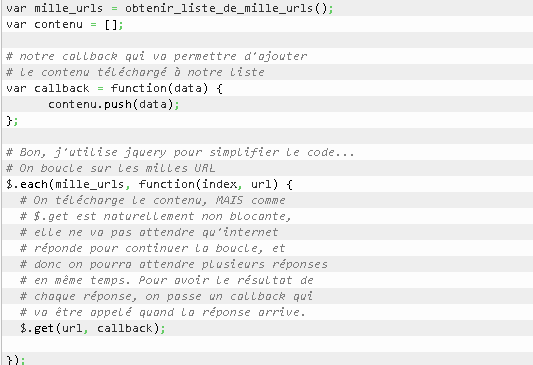
****

Figure 3 : Regarde les nouveaux courriels et les sauvegardes. En cas de suspect un autre processus entre en ligne qui est l’antivirus qui désinfecte le mail en plus de surveiller l’ordinateur (Sam & Max, 2013).

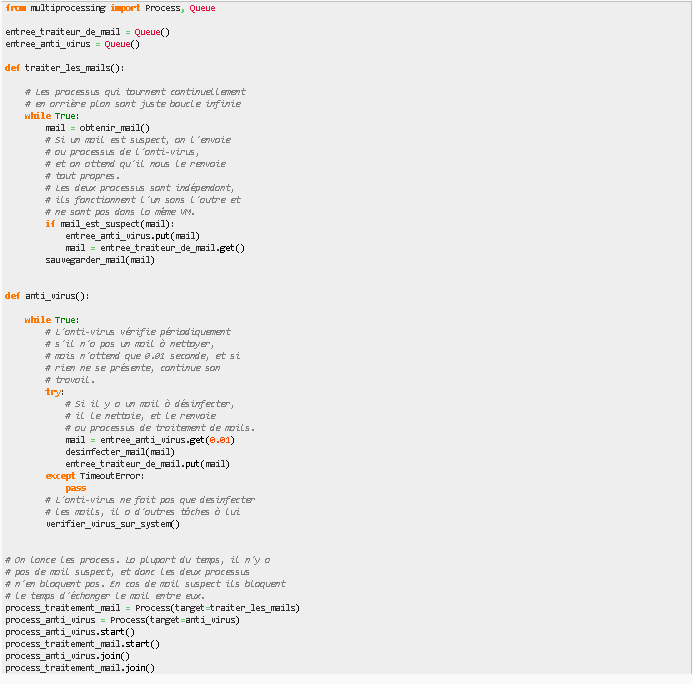


Figure 4 :Télécharge les données d’un site internet (Microsoft, 2016).

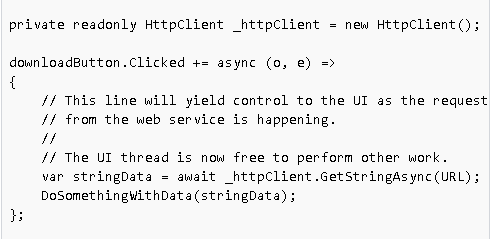
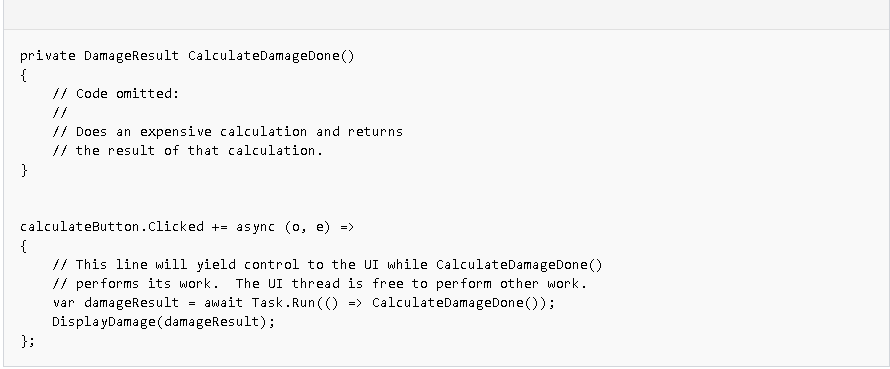
****

Figure 5 : Calcul effectuer dans un jeu. (Microsoft, 2016)



**Annexe 2 : UML & Figure**

Figure 6 : Exemple qui illustre le concept du patron « Active Object » (Lavender & Schmidt).

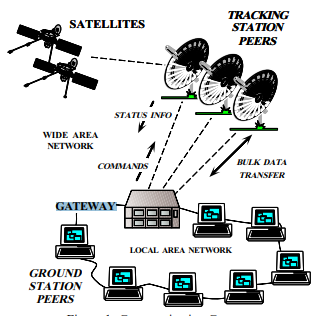
****

Figure 7 : Diagramme de class du patron « Active Object » (Oak, 2014).

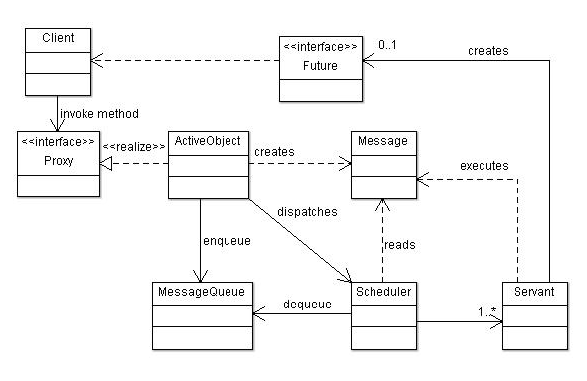
****

Figure 8 : Exemple qui illustre le concept du patron « Proactive » (Pyarali & Harrison & Schmidt & Jordan, s.d.).

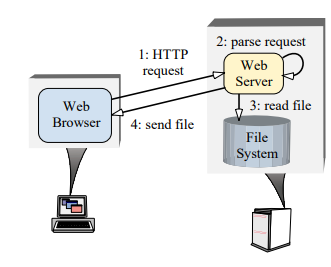
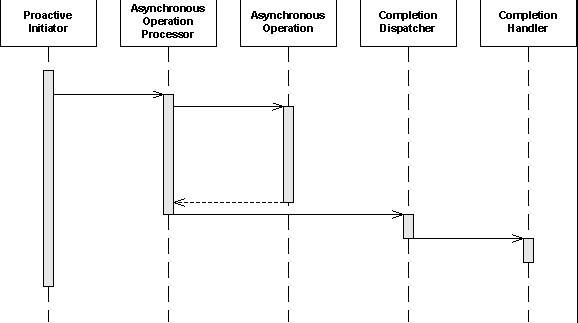


Figure 9 : Diagramme de séquence du patron « Proactive » (Wikipédia, s.d.).



**Références :**

1. Greg Lavender, R., Schmidt, D.C. (s.d.). Active Object. Tiré de <http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf>
2. Pyarali, I., Harrison, T., Schmidt, D.C., Jordan, T.D. Proactor. Tiré de <http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf>
3. Software design pattern. Consulté le 22 mars 2018. Tiré de <https://en.wikipedia.org/wiki/Software_design_pattern>
4. Architecture distribuée. Consulté le 22 mars 2018. Tiré de <https://fr.wikipedia.org/wiki/Architecture_distribu%C3%A9e>
5. Sam & Max. 9 octobre 2013. La différence entre la programmation asynchrone, parallèle et concurrente. Tiré de <http://sametmax.com/la-difference-entre-la-programmation-asynchrone-parallele-et-concurrente/>
6. Carter, P., olprod, OpenLocalizationService, Microsoft. 20 juin 2016. Programmation asynchrone. Tiré de <https://docs.microsoft.com/fr-fr/dotnet/csharp/async>
7. Entrées-sorties. Consulté le 23 mars 2018. Tiré de <https://fr.wikipedia.org/wiki/Entr%C3%A9es-sorties>
8. ORACLE. 2014. Concurrency in JavaFX. Tiré de <https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/concurrency.htm>
9. Future of AI. 2018. ISRAEL’S LARGEST CONFERENCE ON AI .Tiré de <https://www.futureofai.com/>
10. Patron de conception. Consulté le 22 mars 2018. Tiré de <https://fr.wikipedia.org/wiki/Patron_de_conception>
11. Active object. Consulté le 28 mars 2018. Tiré de <https://en.wikipedia.org/wiki/Active_object%20https://repozytorium.ukw.edu.pl/bitstream/handle/item/3523/Active%20object%20design%20pattern.pdf?sequence=1%20UML%20https://madhuraoakblog.wordpress.com/2014/05/10/active-object-pattern/>
12. Oak, M. 10 mai 2014. Active Object Pattern. Tiré de <https://madhuraoakblog.wordpress.com/2014/05/10/active-object-pattern/>
13. Proactor pattern. Consulté le 29 mars 2018. Tiré de <https://en.wikipedia.org/wiki/Proactor_pattern>
14. Thread. Consulté le 24 mars 2018. Tiré de   
    <https://fr.wikipedia.org/wiki/Thread>
15. Schedule. Consulté le 25 mars 2018. Tiré de <https://en.wikipedia.org/wiki/Schedule>