LOG 6306: Patrons pour la compréhension de programme

Foutse Khomh

Patrons pour la conception d'applications distribuées

ÉCOLE POLYTECHNIQUE MONTRÉAL

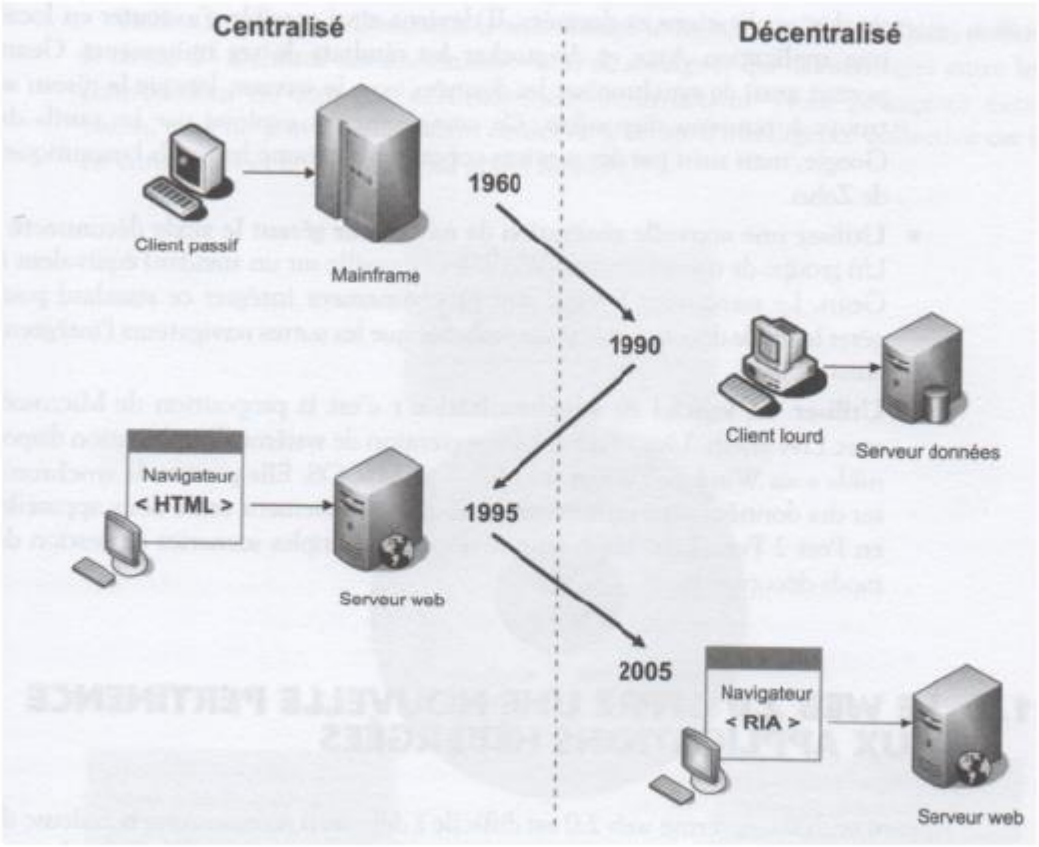
Département de génie informatique et de génie logiciel
École Polytechnique de Montréal

Historique des Architectures

- Les architectures informatiques suivent un cycle régulier de centralisation / décentralisation

- 1960 : mainframes
- 1990 : architectures client/serveur
  - protocoles de communication type CORBA
- 1995 : explosion du web
- 2005 : Rich Internet Application (RIA)

Historique des Architectures



source : Cloud computing et SaaS - Dunod

Rappel sur les patrons de conception

- Définition
  - Patron de conception = *design pattern*
  - Un patron traite un problème de conception récurrent
  - Il apporte une solution générale, indépendante du contexte

- En clair
  - Description de l'organisation de classes et d‘objets en interaction pour résoudre un problème de conception

- Solution générique de conception
  - Est « élégante » et réutilisable
  - A été testée et validée dans l'industrie logicielle
  - Vise un gain en terme des caractéristiques du logiciel
  - Est indépendante du contexte

Rappel sur les patrons de conception

- Quatre éléments principaux définissent un patron
  - Objectif
    - Description de son utilité
  - Problème / Motivation
    - Quand appliquer le patron de conception
    - Relations problématiques entre les classes
  - Solution proposée
    - Éléments impliqués (classes, méthodes, objets)
    - Relations entre les éléments
    - Schémas conceptuels (*e.g.*, diagrammes UML)
  - Conséquences
    - Compromis éventuels
    - Qualité de la solution

Conception d'applications distribuées

- Patrons d'accès aux services et de configuration
  - *Wrapper Facade*
  - *Component Configurator*
  - *Interceptor*
  - *Extension Interface*
- Patrons pour la gestion d'évènements
  - *Reactor*
  - *Proactor*
  - *Asynchronous Completion Token*
  - *Acceptor–Connector*

Patrons d’accès aux services et de configuration

- Façade d'adaptation / *Wrapper Facade*
  - Fournir une interface objet pour une API non-objet existante
- Configurateur (de composant) / *(Component) Configurator*
  - Lier dynamiquement des implémentations (sans recompilation)
- Intercepteur / *Interceptor*
  - Ajouter de manière transparente des services à un cadriciel
- Interface d'extension / *Extension Interface*
  - Permettre à un composant d'exporter plusieurs interfaces

Façade d'adaptation / *Wrapper Facade*

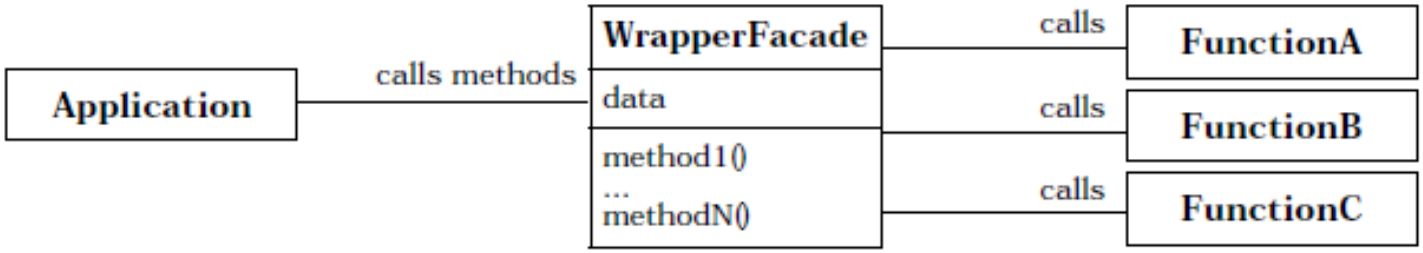
- Objectif
  - Fournir une interface objet pour une API non-objet existante
- Problème
  - Plusieurs applications accèdent à des services offerts par des fonctions de bas niveaux, des structures de données et des bibliothèques non-objets, comme des bibliothèques d'interfaces usagers
  - Les développeurs programmant directement avec ces fonctions de bas niveaux produisent généralement du code
    - Répétitif et peu lisible avec une prolifération de IF et ELSE
    - Non portable, peu robuste, peu cohésif avec une absence de modules encapsulant les différentes fonctions
    - Difficile à maintenir ➔ plus de fautes

Façade d'adaptation / *Wrapper Facade*

- Solution
  - Éviter d'accéder directement aux fonctions et structures de données de bas niveaux
  - À la place
    - Regrouper-les en ensembles cohésifs
    - Créer une ou plusieurs classes *Wrapper Facade*
    - Encapsuler ces fonctions et structures de données dans des méthodes plus concises, robustes, portables et maintenables offertes par l'interface de la *Wrapper Facade*

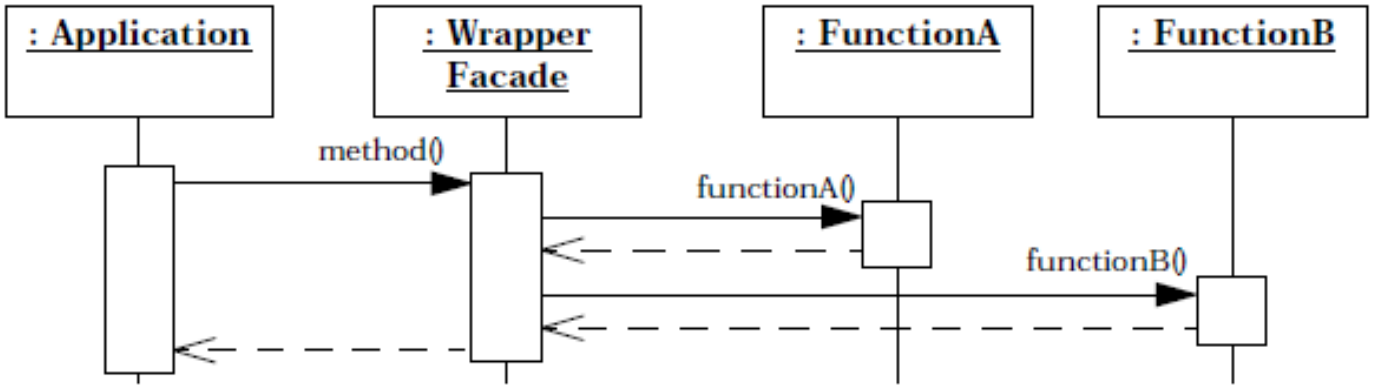
Façade d'adaptation / *Wrapper Facade*

- Deux acteurs : *Wrapper Facade* et classes de fonctions



Façade d'adaptation / *Wrapper Facade*

- Les collaborations sont directes
  - “The application code invokes a method via an instance of the wrapper facade.”
  - “The wrapper facade method forwards the request to one or more of the underlying functions that it encapsulates, passing along any internal data structures needed by the function(s).”



Façade d'adaptation / *Wrapper Facade*

- Utilisations connues
  - Microsoft Foundation Classes (MFC) offre un ensemble de *Wrapper Facade* encapsulant la plus part des fonctions de l'API C Win32
  - Autres
    - Le cadriciel Adaptive Communication Environment (ACE)
    - La Java Virtual Machine et les bibliothèques Java telles que AWT, Swing…

Façade d'adaptation / *Wrapper Facade*

- Conséquences
  - Bénéfices
    - Interfaces de programmation concises, cohésives et robustes
    - Portabilité et maintenabilité
    - Modularité, reutilisabilité et configurabilité
  - Points négatifs
    - Indirection supplémentaires
      - Cependant possibilité de *inline* dans le cas des langages comme C++

Configurateur/ *Configurator*

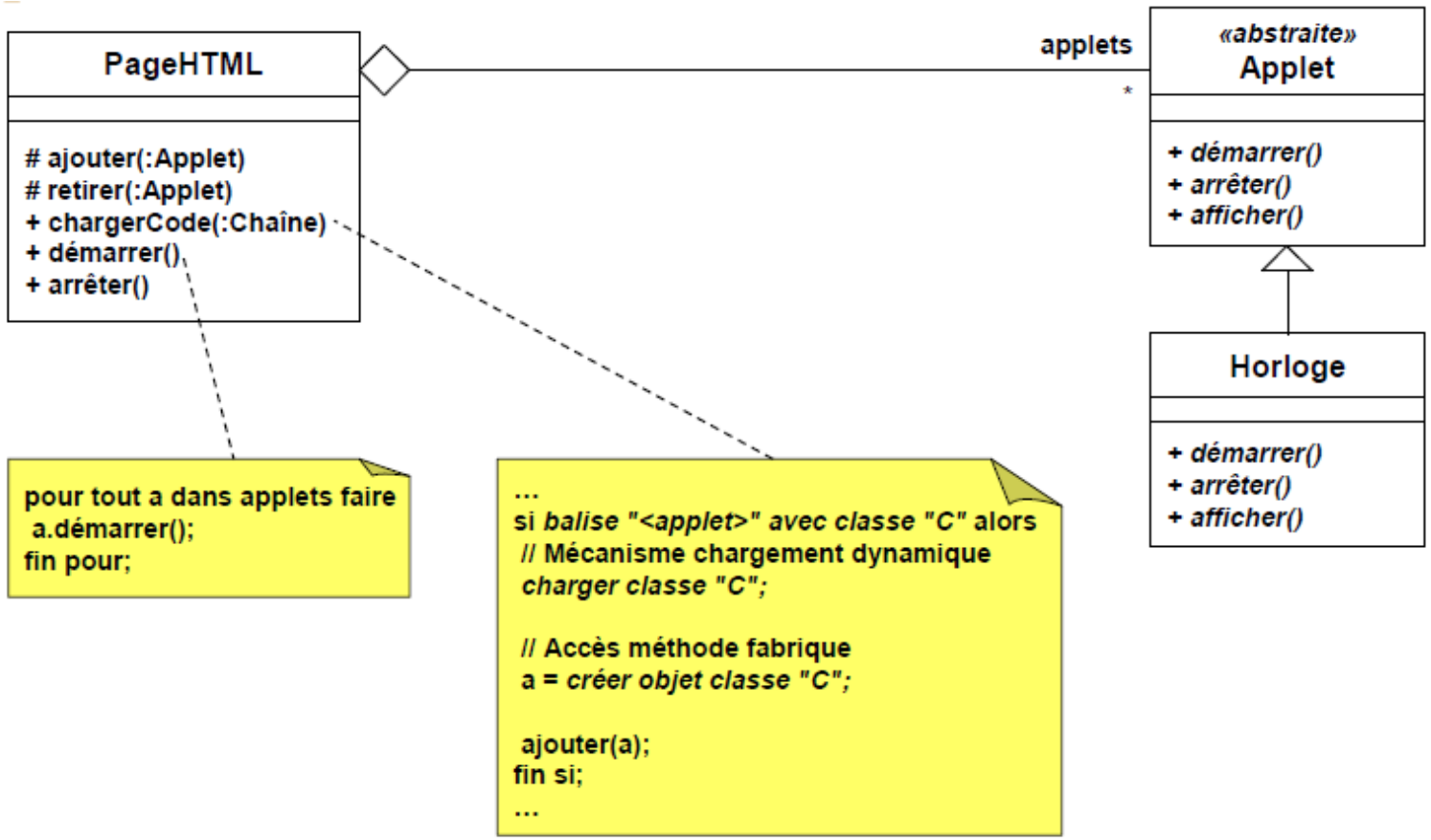
- Objectif
  - Lier et délier l'implémentation de composants à l'exécution
  - Sans recompiler ni éditer les liens
- Problème
  - Un système ou ses composants doivent être démarrer, arrêter, redémarrer ou échanger de manière flexible et en toute transparence pendant l’exécution
  - Le système doit offrir des mécanismes pour permettre cette reconfiguration de ses composants à tout moment de son exécution

Configurateur/ *Configurator*

- Solution
  - Une classe abstraite représente les composants
    - Séparation de l'interface des composants et de leurs implémentations
      - Nouvelle implémentation = définition d'une sous-classe
      - Sous-classe stockée dans une unité chargeable dynamiquement
        - » Exemples : bibliothèques dynamiques (DLL), classes Java
    - Création d'interfaces de configuration des composants
      - Ces interfaces sont utilisées pour modifier et configurer les composants
  - Un « configurateur » manipule les composants
    - S'occupe de la création, du démarrage et de l'arrêt des composants
    - Exemple : *link* / *unlink* de DLLs

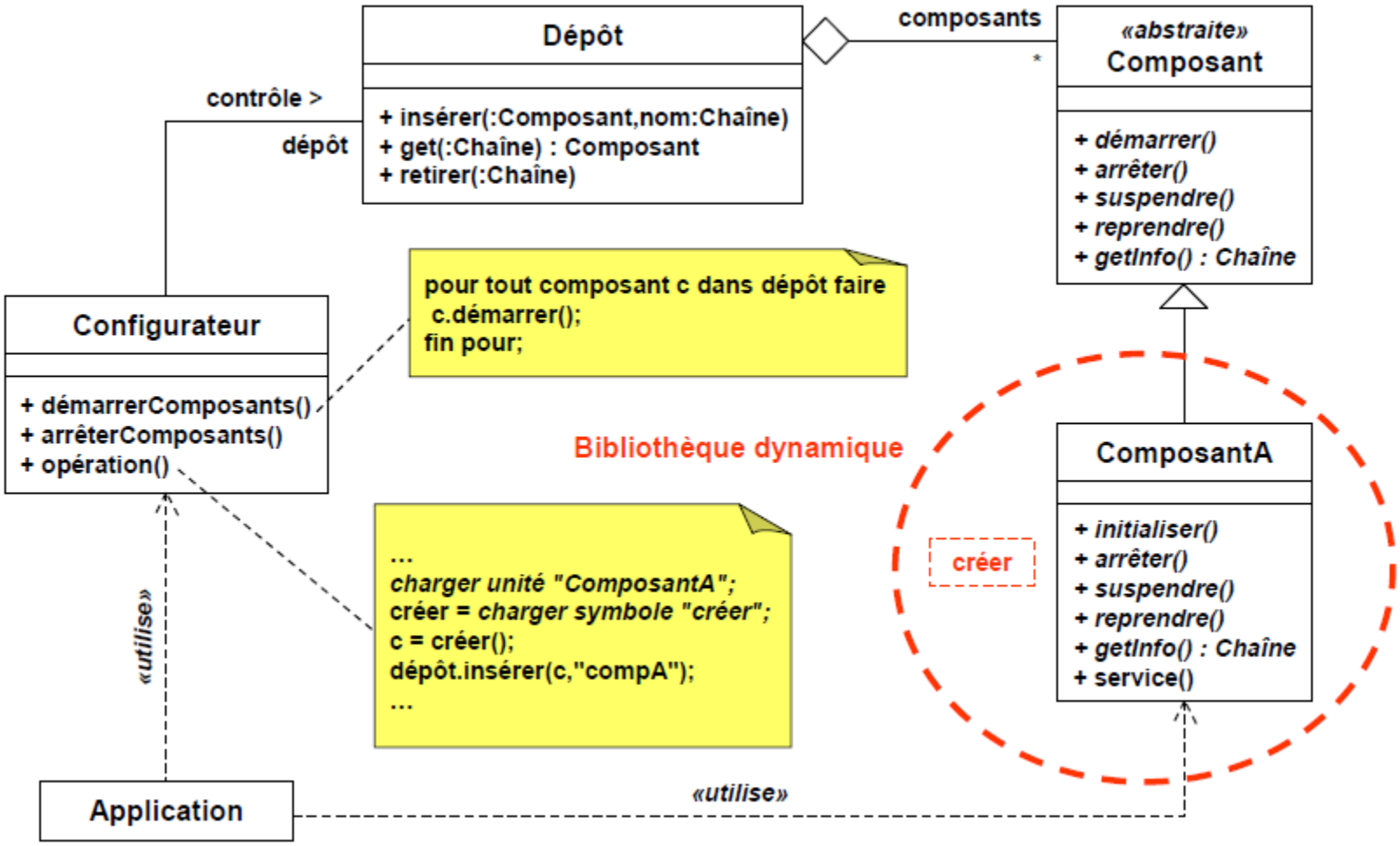
Configurateur/ *Configurator*

Exemple : page HTML où les appliquettes sont chargées dynamiquement



Exemple code HTML: `<applet code="Horloge.class">...</applet>`

Configurateur/ *Configurator*

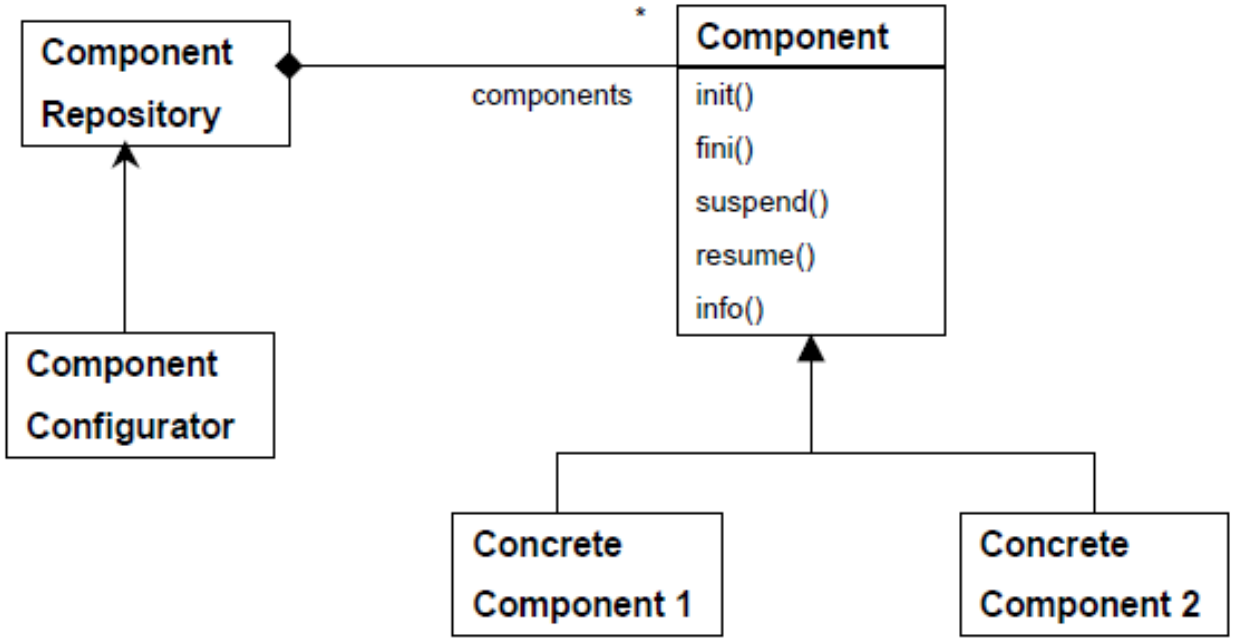


Configurateur/ *Configurator*

- Composant
  - Défini une interface uniforme permettant de configurer et contrôler les services ou fonctionnalités offertes par une implémentation de composants (démarrage, arrêt, affichage)
- Composant concret : ComposantA
  - Implémente l'interface de contrôle « Composant » pour offrir un type spécifique de composant
  - Implémente les méthodes pour offrir des services à l'application
  - Peut être *linked* / *unlinked* à l'exécution (DLL)
- Répertoire de Composants : Dépôt
  - Responsable de la gestion de tous les composants dans le système
  - Permet la gestion du système et le contrôle des composants par l'intermédiaire d'un mécanisme d'administration centrale

Configurateur/ *Configurator*

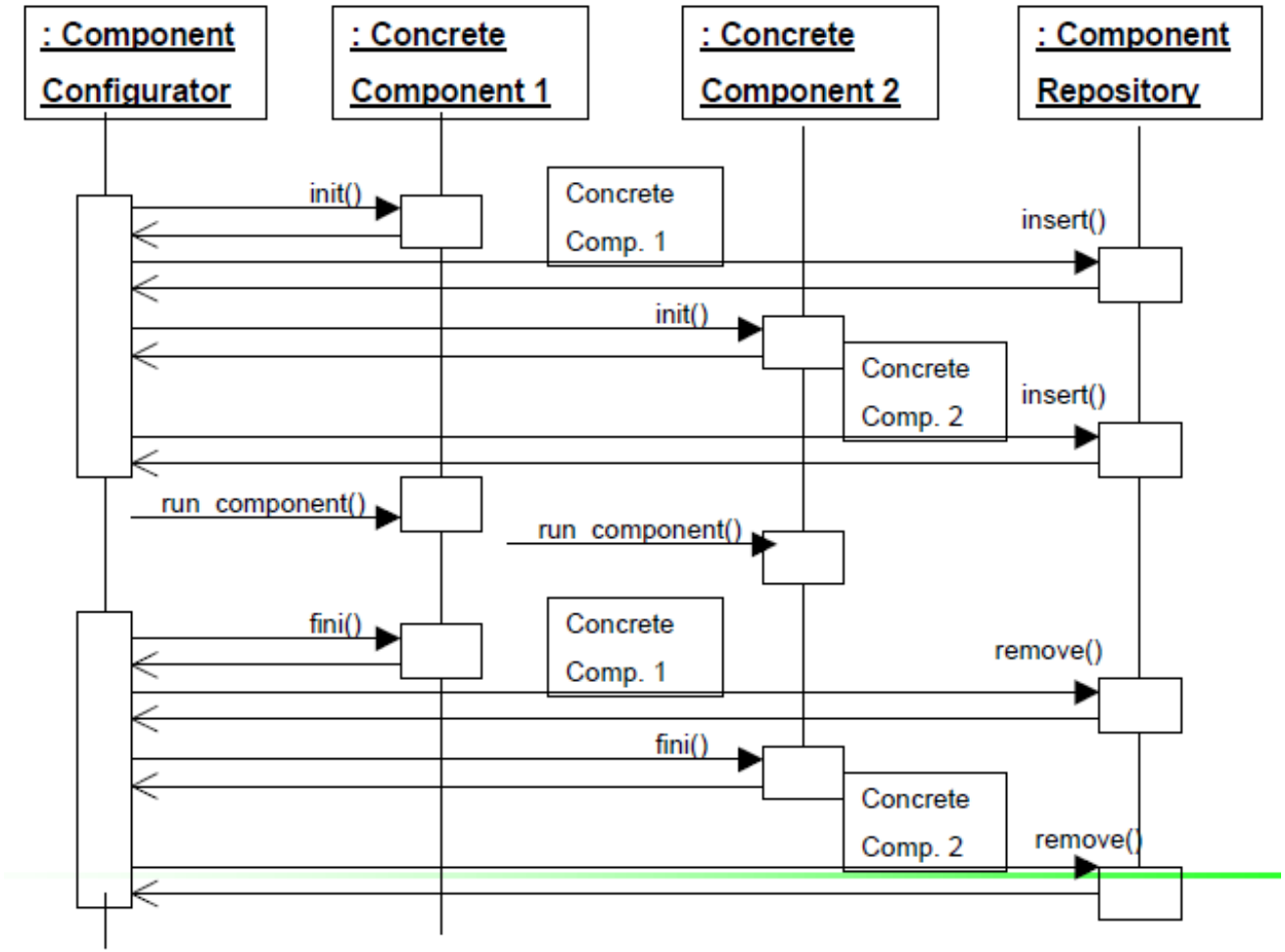
- Configurateur
  - Grâce au répertoire de composants, coordonne la (re)configuration des composants concrets
  - Interprète et exécute un script de (re)configuration des composants dans le système (via *dynamic linking* et *unlinking* à partir des DLLs)



Configurateur/ *Configurator*

- Les collaborations entre acteurs sont
  - *Component initialization*
    - *Dynamically links a component to a system*
    - *Initializes it*
    - *If component has been successfully initialized, adds it to the component repository that manages all configured components at run-time*
  - *Component processing*
    - *Exchanging messages with peer components*
    - *Performing service requests*
    - *Component configurator can suspend and resume existing components temporarily*
  - *Component termination*
    - *Before terminating a component, configurator allows them to clean up their resources*
    - *Removes it from the component repository*
    - *Unlink it from the system address space*

Configurateur/ *Configurator*



Configurateur/ *Configurator*

- Appelé aussi « service configurator »
- Utilisations connues
  - Windows Service Control Management (SCM), OSGi, drivers de Linux, appliquettes…
- Conséquences
  - Bénéfices
    - Uniformité des composants
      - Tous les composants respectent la même interface
    - Administration centralisée
      - Facilite le démarrage/arrêt de tous les composants
    - Modularité, testabilité et reutilisabilité

Configurateur/ *Configurator*

- Bénéfices (suite)
  - Remplacement de composants « à chaud »
    - Chargement / déchargement dynamique de composants
  - Permet une adaptation dynamique
    - Mécanisme d'analyse / apprentissage
    - Réglage des paramètres du composant
    - Chargement d'un composant mieux adapté
- Points négatifs
  - Absence de déterminisme et d’ordre dans les dépendances
  - Risques liées à la sécurité et fiabilité
  - Augmentation de la complexité
  - Augmentation de *run-time overhead*

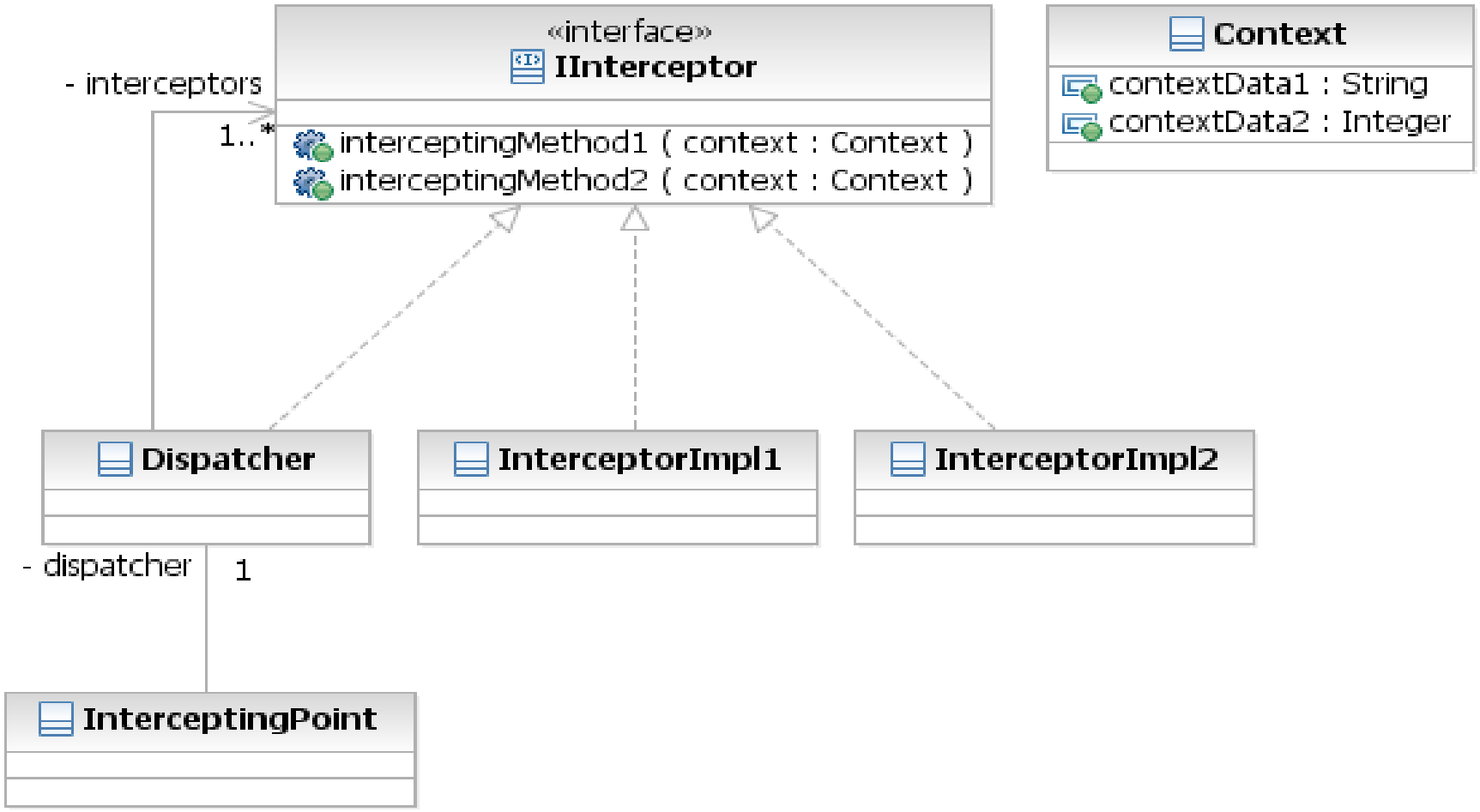
Intercepteur / *Interceptor*

- Objectif
  - Ajouter de manière transparente des services à un cadriciel
  - Les activer automatiquement suite à certains événements
- Problème
  - Sur un système on souhaite pouvoir
    - Contrôler son fonctionnement
    - Modifier une partie de son comportement
    - Étendre le système avec de nouvelles fonctionnalités sans
      - Connaître le reste du système
      - Changer le code existant
  - Les nouvelles extensions ne doivent pas affecter le système

Intercepteur / *Interceptor*

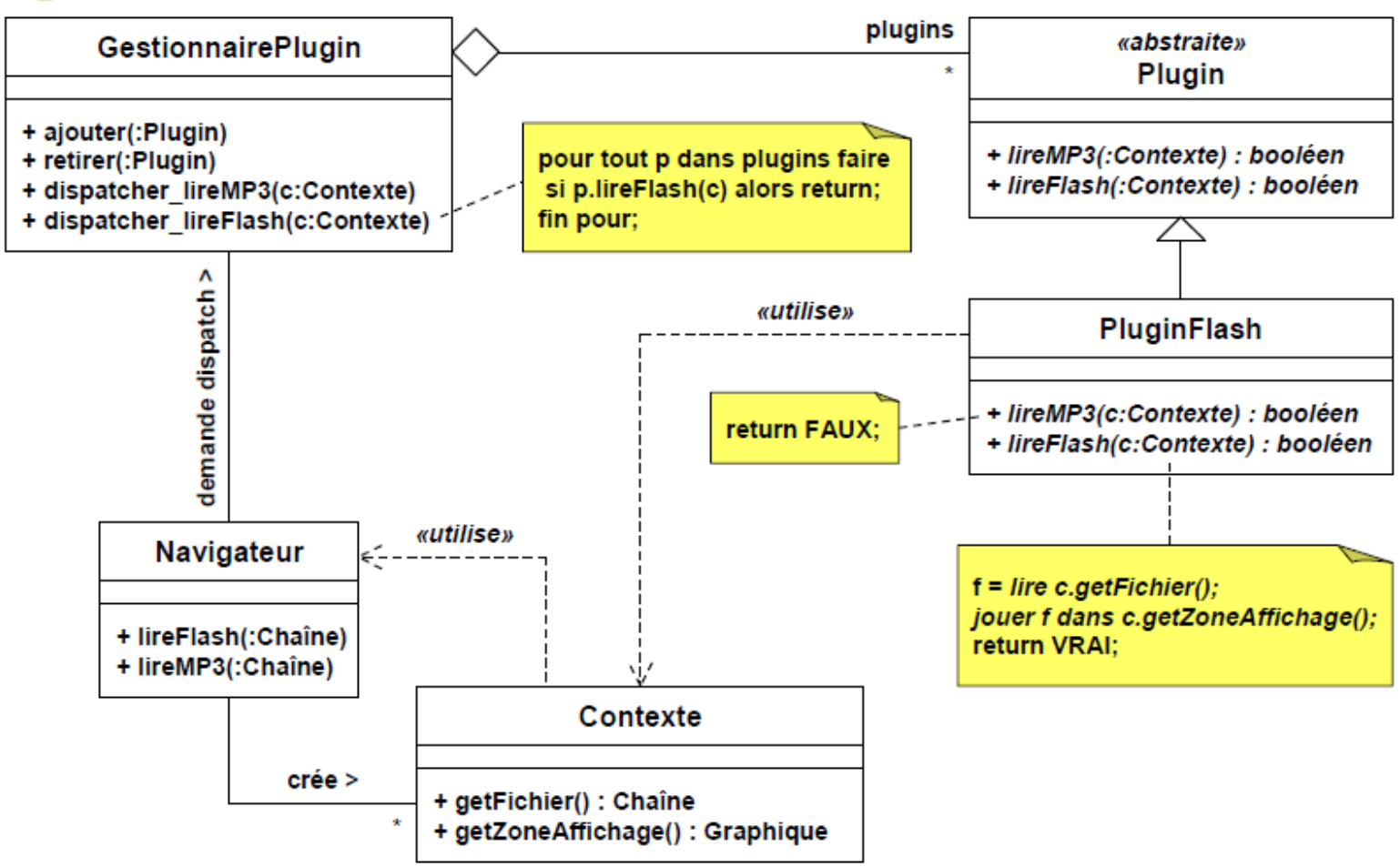
- Solution
  - Un « dispatcheur » est chargé de diffuser les événements
  - Des « intercepteurs » sont capables de recevoir les événements
    - Classe abstraite possédant une méthode par événement
    - Héritage ➔ Proposition de nouveaux services
  - Les intercepteurs n'ont pas un accès direct au cadriciel
    - Communication par l'intermédiaire d'un « contexte »
    - Le contexte accède aux données et services du cadriciel

Intercepteur / *Interceptor*

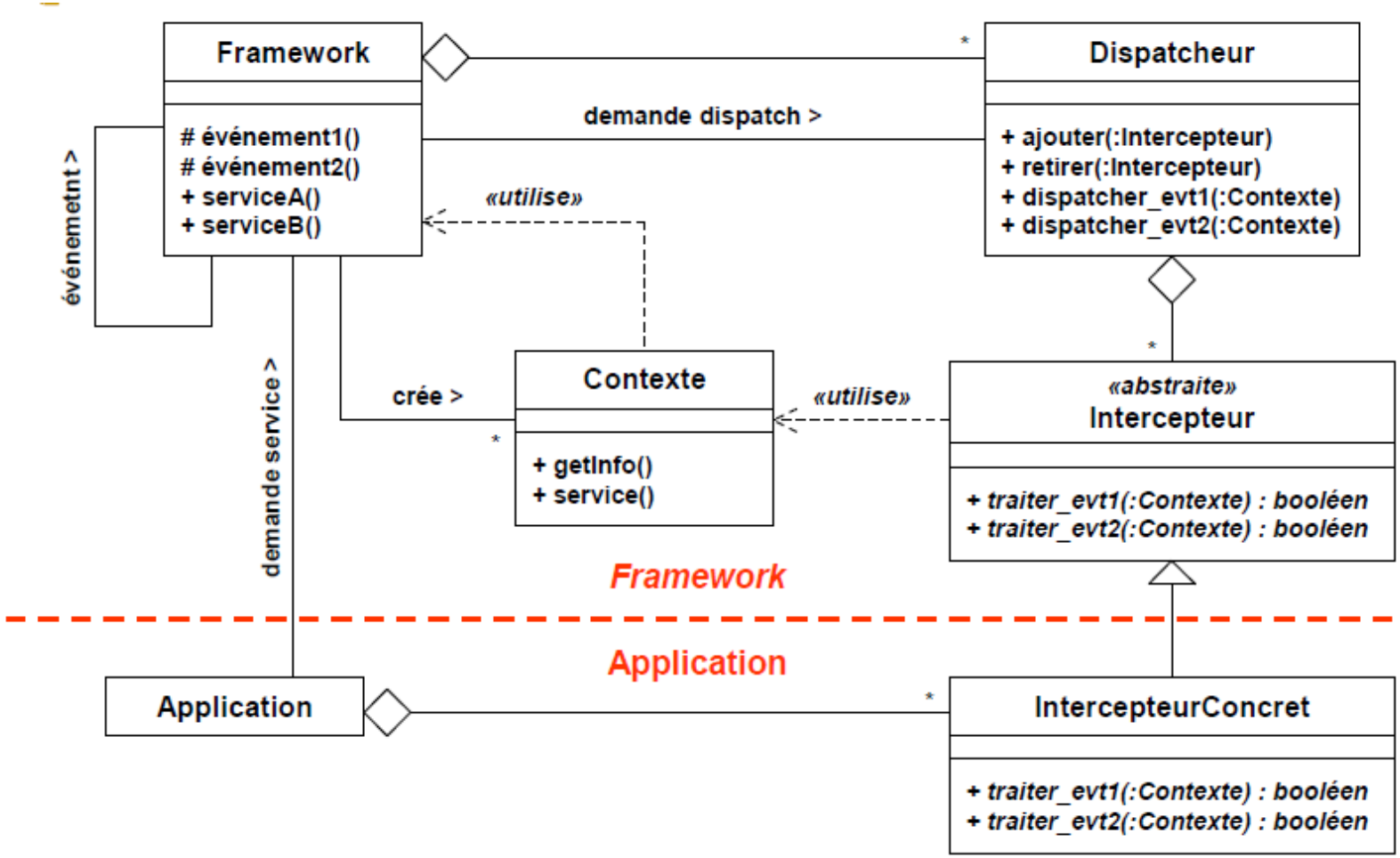


Intercepteur / *Interceptor*

Exemple : mécanisme de plugins dans un navigateur Web



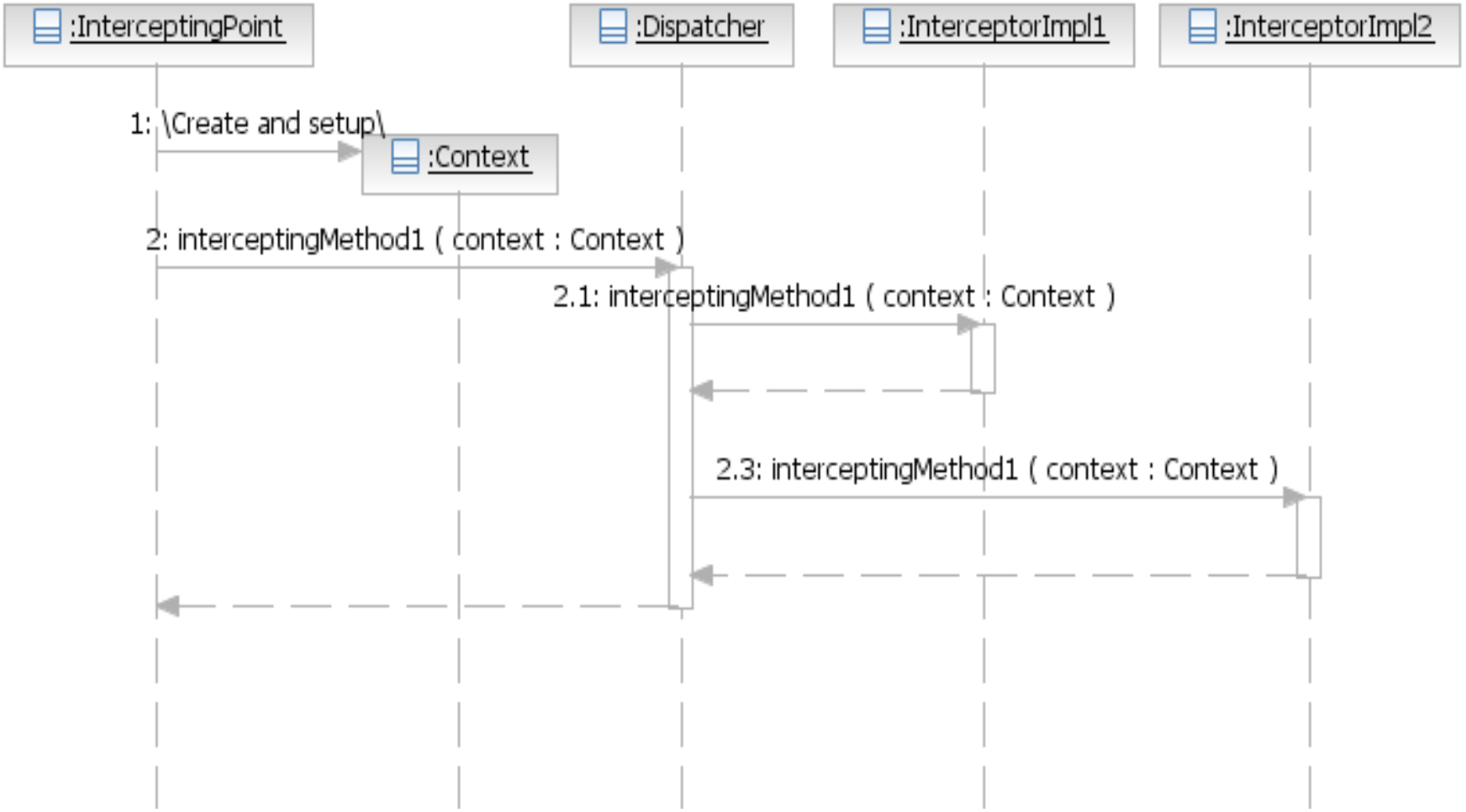
Intercepteur / *Interceptor*



Intercepteur / *Interceptor*

- Les collaborations sont
  - *Intercepteur* – L'interface IIntercepteur définit les méthodes exécutées par le Dispatcheur dans des circonstances spécifiques (en début de requête, en fin de requête, …). Cette interface est implémentée par des « *intercepteurs* » concret (*logging*, *authorization*, *transaction*, ...).
  - *Dispatcheur* – La classe Dispatcheur implémente l'interface IIntercepteur. Le Dispatcheur est la classe invoquée à partir d'un « intercepting point » pour effectuer l'interception
  - *Contexte* – La classe Contexte contient typiquement des données utilisées par les Intercepteurs ou les résultats des interceptions (exemple : *transaction id generated by TransactionInterceptor*)

Intercepteur / *Interceptor*



Intercepteur / *Interceptor*

- Conséquences
  - Bénéfices
    - Extensibilité du cadriciel
      - Services liés aux intercepteurs intégrés au cadriciel
      - Nouveaux services ➔ Héritage de « Intercepteur »
        - » Aucun impact sur le cadriciel
    - Séparation des intérêts
      - Infrastructure du cadriciel d'un côté, les services de l'autre
        - » Inutile de connaître toute l'infrastructure pour coder un service
    - Mécanisme de contrôle et de surveillance
      - Intercepteur + Contexte ➔ Moyen de tracer le système

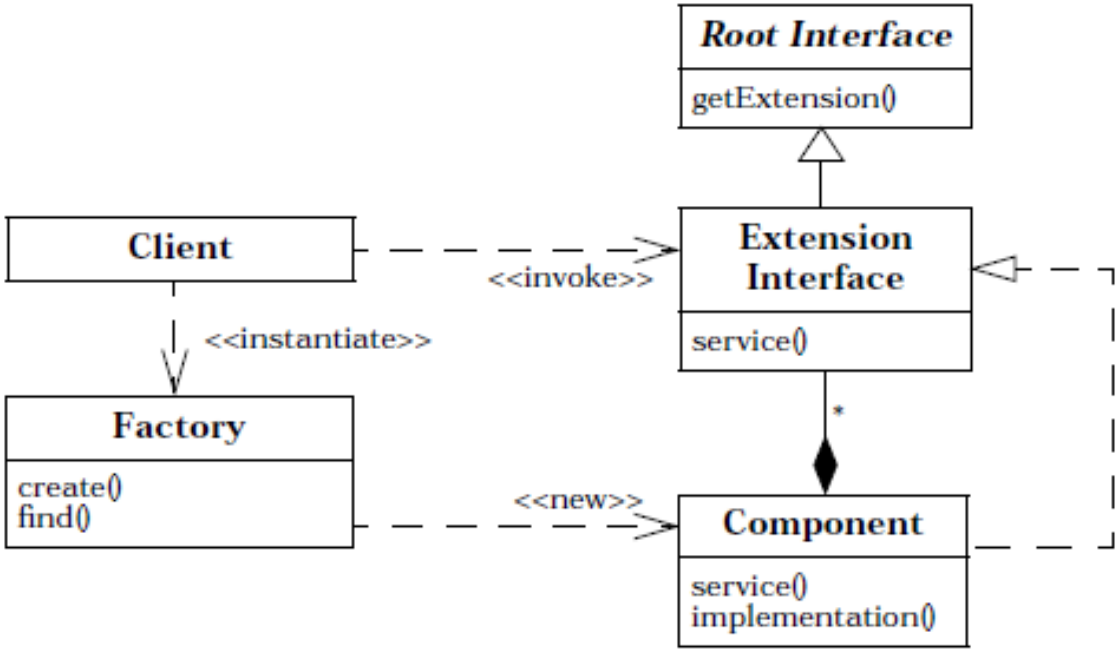
Interface d'extension / *Extension Interface*

- Objectif
  - Permettre à un composant d'exporter plusieurs interfaces
  - Éviter le « gonflement » de l'interface lors de l'ajout de nouvelles fonctionnalités
- Problème
  - Il est difficile d'anticiper tous les besoins des clients ainsi que leurs utilisations possible des composants
  - Les utilisateurs souhaitent en permanence des modifications et extensions des fonctionnalités des composants
  - Ces changements pourraient déstabiliser l'architecture du composant, compliquer son déploiement et sa réutilisation avec au passage un risque élevé d'introduction de fautes

Interface d'extension / *Extension Interface*

- Solution
  - Décomposer l'interface par intérêt
    - Une interface mère + des interfaces filles (une par intérêt)
  - Un composant agrège plusieurs interfaces
    - Plusieurs possibilités
      - ➔Implémentation multiple des interfaces
      - ➔ Agrégation d'interfaces et délégation
- Motivation
  - Proposer de nombreuses fonctionnalités sur des composants
  - Éviter d'alourdir l'interface de tous les composants

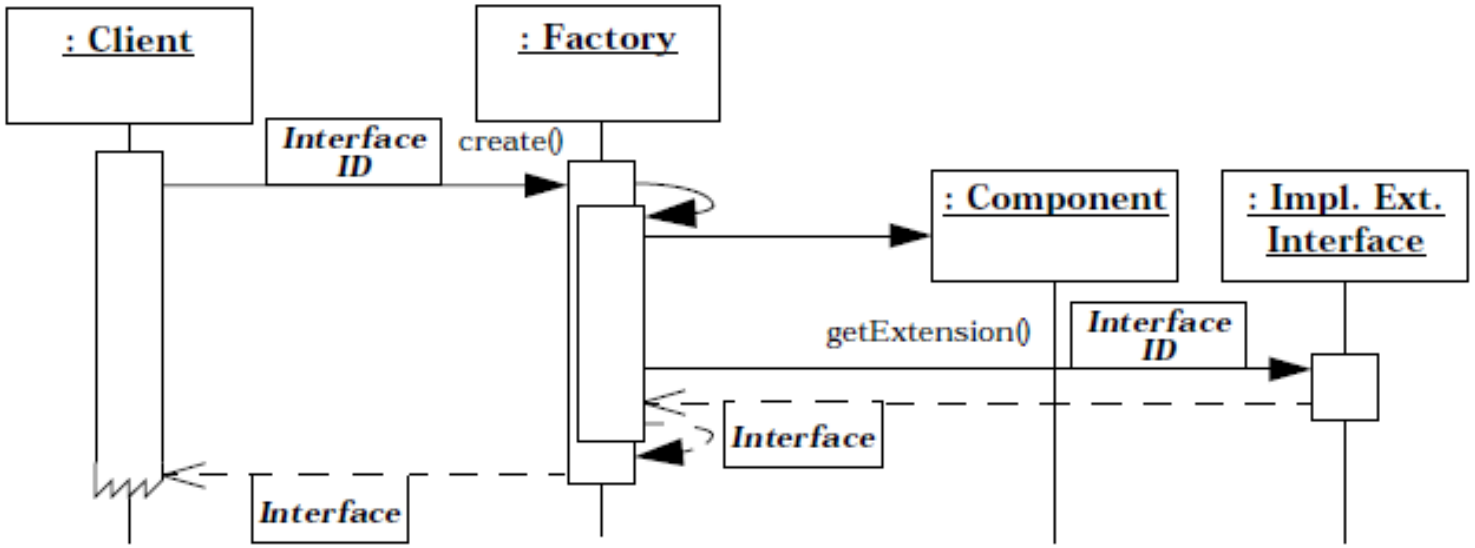
Interface d'extension / *Extension Interface*



Interface d'extension / *Extension Interface*

- Les collaborations sont
  - Scénario 1 : le client crée de nouveaux composants et obtient une interface d’extension
    - *The client asks the factory to create a new component and to return a particular extension interface*
    - *The factory creates a new component and retrieves an extension interface as a result*
    - *The factory asks the root interface for the requested extension interface and then returns the extension interface to the client*

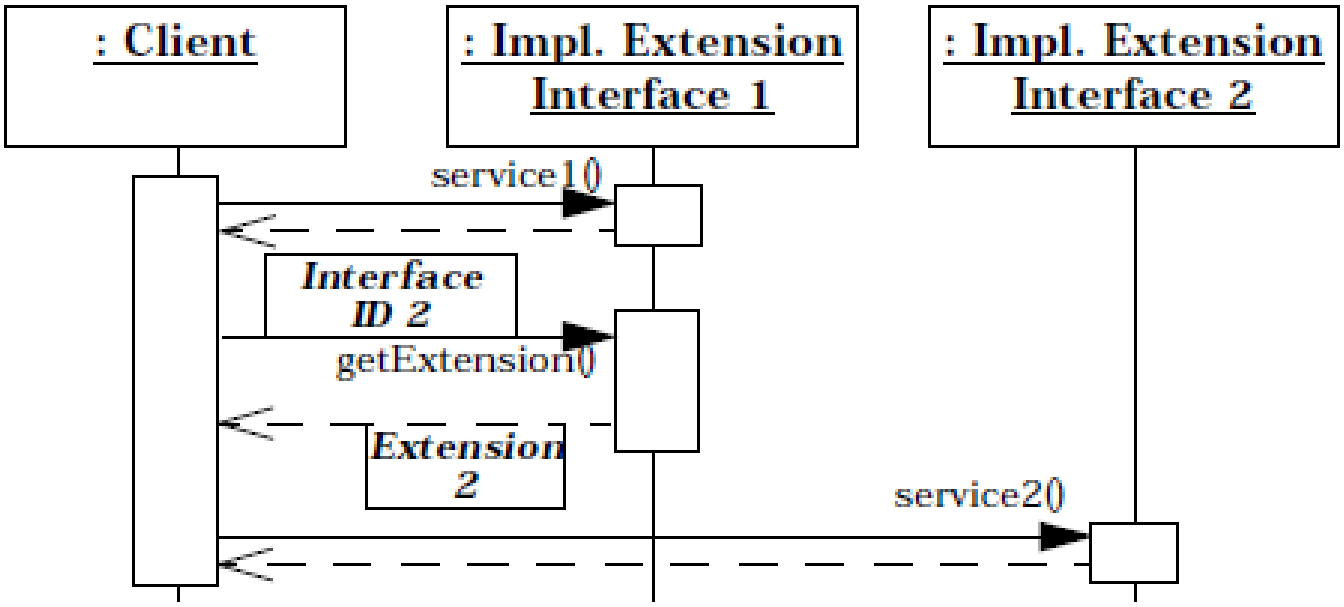
Interface d'extension / *Extension Interface*



Interface d'extension / *Extension Interface*

- Scénario 2 : collaboration entre les clients et les interfaces d'extension
  - *The client calls a method on Extension Interface 1*
    - *Being called by the client, the implementation of Extension Interface 1 within the component executes the requested method and returns results, if any, back to the client*
  - *The client calls the getExtension() method of Extension Interface 1. It passes a parameter specifying which extension interface it is interested in. The getExtension() denotes a generic method derived from the root interface, therefore it is implemented by all extension interfaces. The implementation of Extension Interface 1 within the component locates the requested Extension Interface 2 and returns it to the client*
  - *The client calls a method on Extension Interface 2 that is then executed*

Interface d'extension / *Extension Interface*

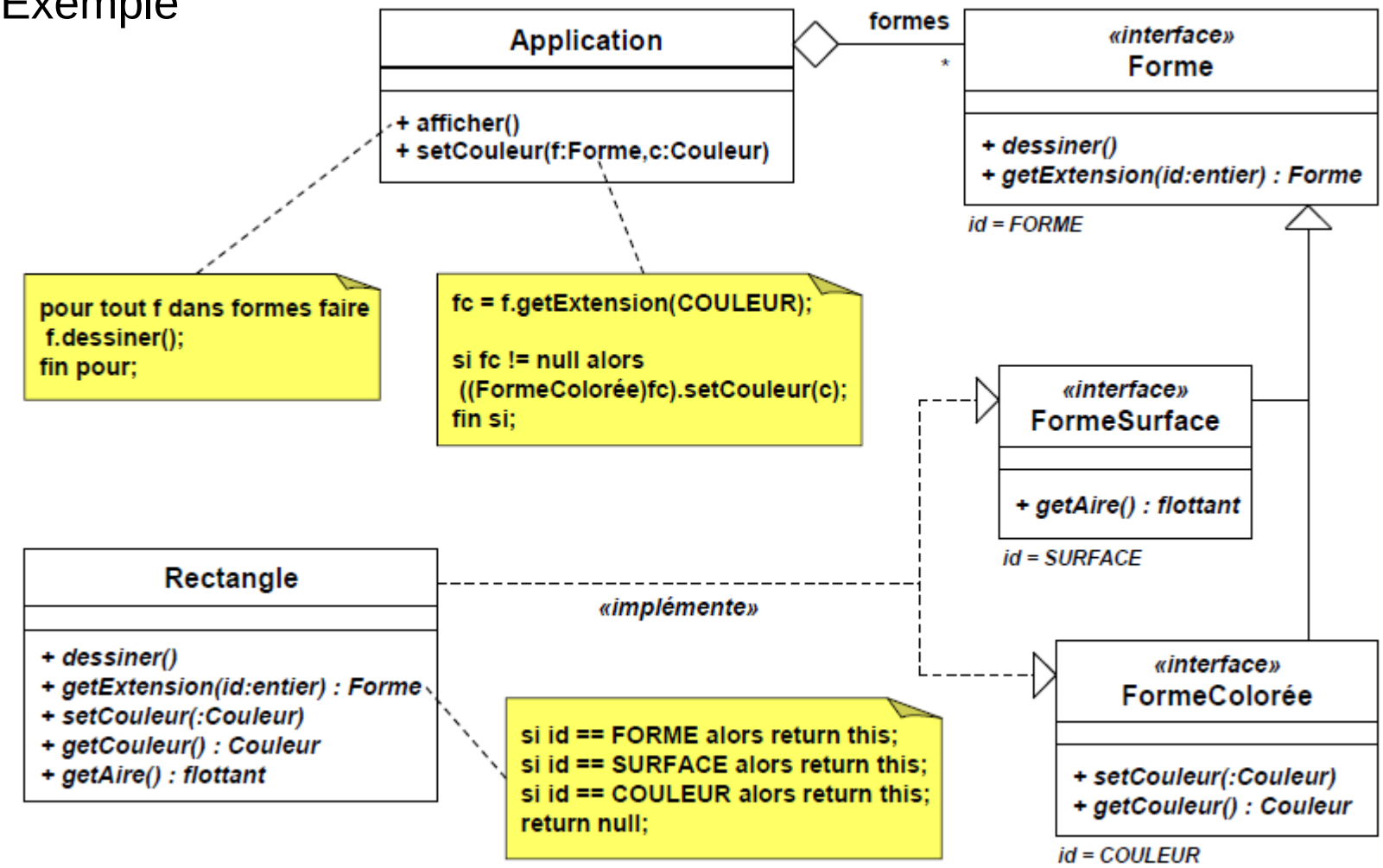


Interface d'extension / *Extension Interface*

- Utilisations connues
  - Microsoft COM/COM+ s'appuie sur des extension d'interfaces
  - Autres
    - CORBA3 : les composants peuvent offrir plus d'une interface
    - OpenDoc, l'ajout des fonctionnalités se fait par extensions d'interfaces
- Conséquences
  - Bénéfices
    - Extensibilité : ajout de nouvelles fonctionnalités ➔ Nouvelle interface
    - Séparation des intérêts : une interface par intérêt
  - Points négatifs
    - Attention au surcoût : accès indirect au composant
    - Plus de complexité
- Relations avec d'autres patrons
  - Pont
    - Implémentation de la composition d'interfaces
  - Fabrique abstraite
    - Peut être utilisée pour la création de composants

Interface d'extension / *Extension Interface*

Exemple



Patrons pour la gestion d'évènements

- *Reactor*
  - *Demultiplexing and Dispatching Handles for Synchronous Events*
- *Proactor*
  - *Demultiplexing and Dispatching Handlers for Asynchronous Events*
- *Asynchronous Completion Token*
  - *Dispatches processing actions within a client in response to the completion of asynchronous operations invoked by the client*
- *Acceptor-Connector*
  - *Connecting and Initializing Communication Services*

Reactor

- Objectif
  - Permettre la gestion de requêtes de services concurrentes par une application
  - Chaque service étant constitué d'une ou plusieurs méthodes et représenté par un « Event handler » différent, celui-ci est responsable du « dispatching » des requêtes de services spécifiques
- Problème
  - Le serveur d'une application distribuée reçoit des requêtes concurrentes d'un ou plusieurs clients
  - Avant d'invoquer un service spécifique, ce serveur doit demultiplexer et dispatcher ces requêtes aux fournisseurs de services correspondants
  - Le serveur doit toujours être disponible pour traiter de nouvelles requêtes même si d'autres requêtes sont en exécution

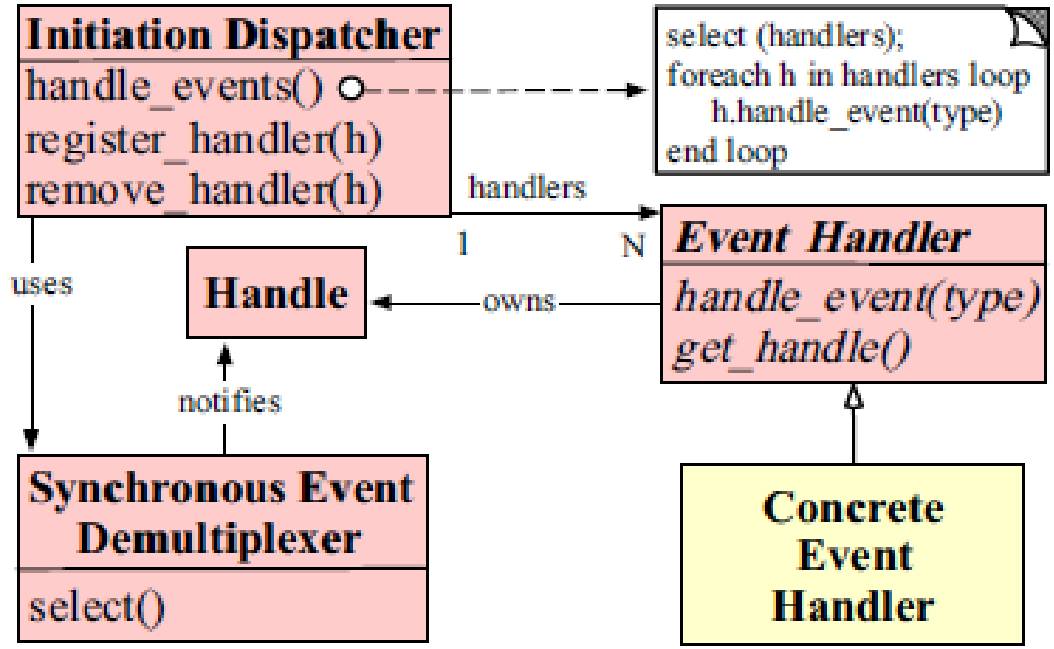
Reactor

- Problème (suite)
  - Le temps de latence doit être minimal et le débit maximal
  - La conception du serveur doit être simple et adaptable aux nouveaux services tel que le changement du format des messages ou l'ajout de cache par exemple
  - Toute implémentation de nouveaux services doit se faire sans modification des mécanismes génériques de demultiplexage et dispatching
  - Le serveur doit être portable

Reactor

- Solution
  - Pour chaque service offert par l'application, introduire un *Event Handler* différent charger de traiter un type d'événement
  - Tous les *Event Handlers* implémentent une interface commune
  - Implémenter un *Initiation Dispatcher* pour enregistrer, retirer et dispatcher les *Event Handlers*
  - Implémenter un *Synchronous Event Demultiplexer* charger de repérer les événements et d'informer le *Initiation Dispatcher*. Celui-ci fera ensuite appel aux *Event Handler* des services invoqués

Reactor


Initiation Dispatcher
handle_events()
register_handler(h)
remove_handler(h)
select (handlers);
foreach h in handlers loop
h.handle_event(type)
end loop
handlers
1
N
Event Handler
handle_event(type)
get_handle()
uses
Handle
owns
notifies
Synchronous Event Demultiplexer
select()
Concrete Event Handler

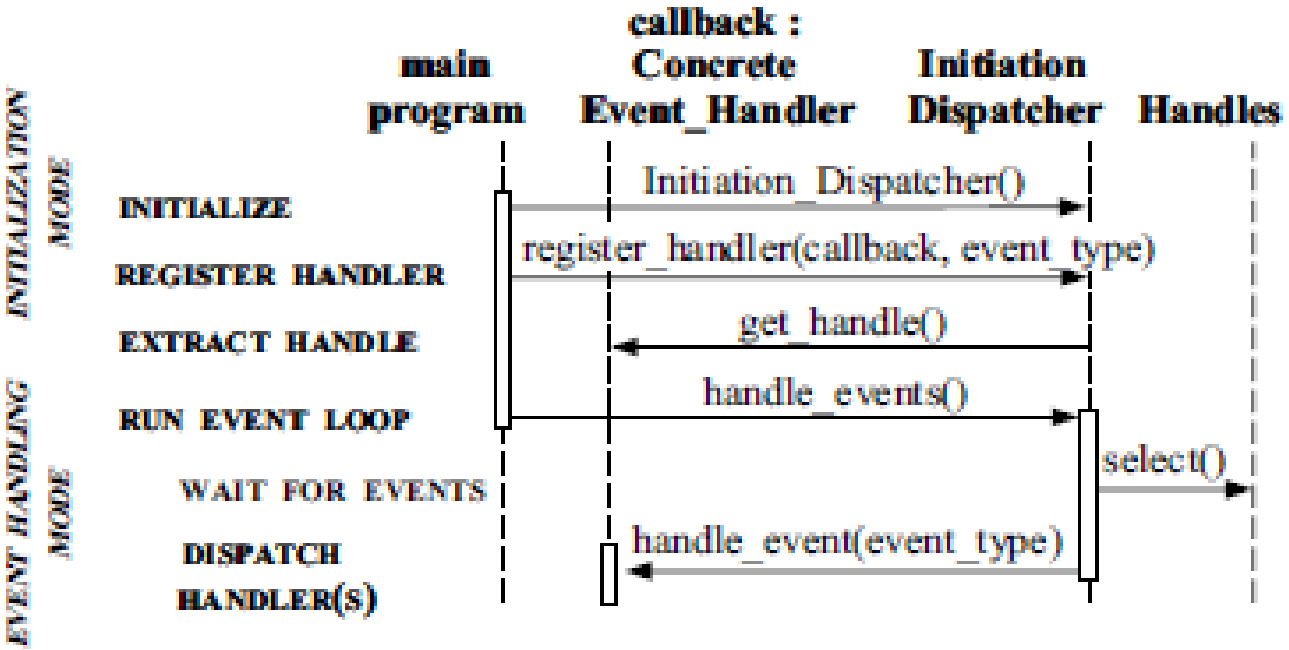
Reactor

- Les collaborations sont
  - *When an application registers a Concrete Event Handler with the Initiation Dispatcher the application indicates the type of event(s) this Event Handler wants the Initiation Dispatcher to notify it about when the event(s) occur on the associated Handle*
  - *The Initiation Dispatcher requests each Event Handler to pass back its internal Handle.This Handle identifies the Event Handler to the OS*
  - *After all Event Handlers are registered, an application calls handle events to start the Initiation Dispatcher's event loop. At this point, the Initiation Dispatcher combines the Handle from each registered Event Handler and uses the Synchronous Event Demultiplexer to wait for events to occur on these Handles*

Reactor

- *The Synchronous Event Demultiplexer notifies the Initiation Dispatcher when a Handle corresponding to an event source becomes “ready,”*
- *The Initiation Dispatcher triggers Event Handler hook method in response to events on the ready Handles. When events occur, the Initiation Dispatcher uses the Handles activated by the event sources as “keys” to locate and dispatch the appropriate Event Handler’s hook method.*
- *The Initiation Dispatcher calls back to the handle event hook method of the Event Handler to perform application-specific functionality in response to an event.*

Reactor


main program
callback : Concrete Event_Handler
Initiation Dispatcher
Handles
INITIALIZATION MODE
INITIALIZE
Initiation_Dispatcher()
REGISTER HANDLER
register_handler(callback, event_type)
EXTRACT HANDLE
get_handle()
EVENT HANDLING MODE
RUN EVENT LOOP
handle_events()
WAIT FOR EVENTS
select()
DISPATCH HANDLER(S)
handle_event(event_type)

Reactor

- Utilisations connues
  - ACE Framework utilise ce patron pour demultiplexer et dispatcher les évènements
  - Autres
    - CORBA ORBs
    - Ericsson EOS Call CenterManagement System
- Conséquences
  - Bénéfices
    - Séparation des intérêts
      - Une interface par intérêt
    - Augmente la modularité, reutilisabilité, portabilité et configurabilité
  - Points négatifs
    - Ne peut être appliqué que si l'OS supporte les Handles
    - Difficile à déboguer

Proactor

- Objectif
  - Permettre la gestion de requêtes de services concurrentes
  - Chaque service est constitué d'une ou plusieurs méthodes et représenté par un « Event handler » différent, celui-ci étant responsable du « dispatching » des requêtes de services
  - À la différence du patron *Reactor*, ici on souhaite augmenter la performance en tirant avantage du parallélisme
- Problème
  - Une application doit exécuter une ou plusieurs opérations asynchrones sans blocage
  - L'application doit être notifiée lorsque les opérations asynchrones sont terminées
  - Le temps de latence doit être minimal et le débit maximal

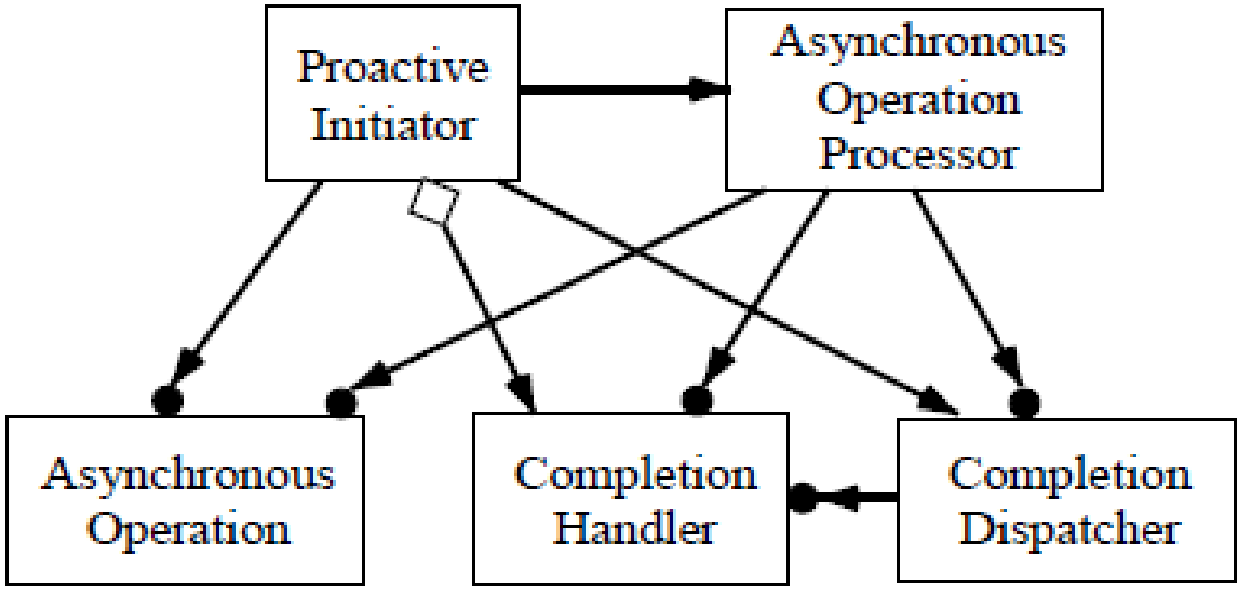
Proactor

- Problème (suite)
  - La conception du serveur doit être simple et adaptable aux nouveaux services tel que le changement du format des messages ou l'ajout de cache par exemple
  - Toute implémentation de nouveaux services doit se faire sans modification des mécanismes générique de demultiplexage et dispatching
  - Le serveur doit être portable
  - On souhaite gérer la planification des tâches (chose qui est confiée à l'OS dans le cas du patron *Reactor* et qui résulte en des problèmes d'équité entre les clients de la plateforme, car il y a un seul fil d'exécution à partager)
  - L'application doit être plus performante que dans le cas d'utilisation du patron *Reactor*

Proactor

- Solution
  - Implémenter un *Proactive Initiator* pour le lancements des opérations asynchrones de *Asynchronous Operations*
  - Implémenter un *Completion Handler* pour la notification et l'exécution des *Asynchronous Operations*
  - L' OS doit implémenter un *Asynchronous Operation Processor* pour la gestion des exécutions de *Asynchronous Operation*. Lorsque les opérations asynchrones (*Asynchronous Operation*) sont terminées, celui-ci se charge de prévenir le *Completion Dispatcher*
  - Le *Completion Dispatcher* est chargé de faire appel aux *Handlers*

Proactor


Proactive Initiator
Asynchronous Operation Processor
Asynchronous Operation
Completion Handler
Completion Dispatcher

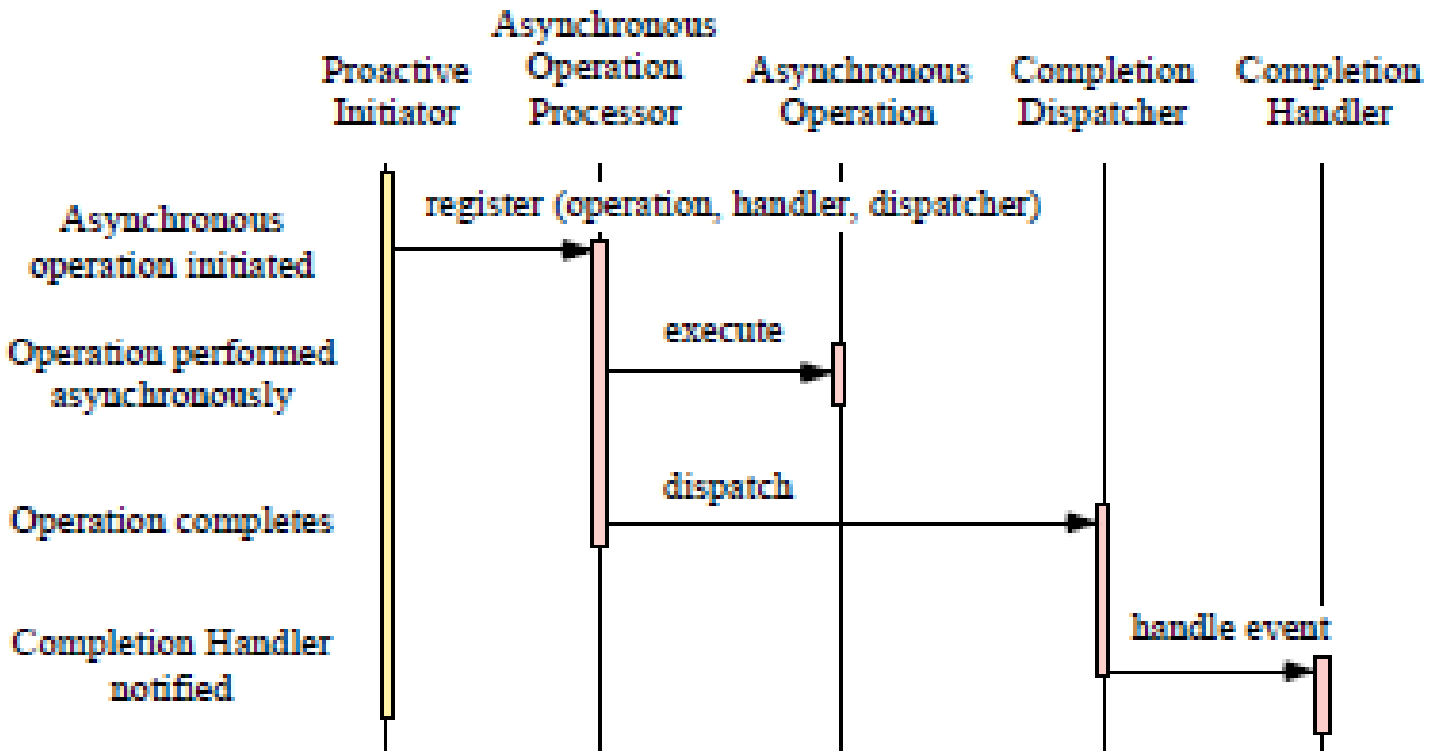
Proactor

- Les collaborations sont
  - *Proactive Initiators initiates operation: To perform asynchronous operations, the application initiates the operation on the Asynchronous Operation Processor*
  - *Asynchronous Operation Processor performs operation: When the application invokes operations on the Asynchronous Operation Processor it runs them asynchronously with respect to other application operations*

Proactor

- *The Asynchronous Operation Processor notifies the Completion Dispatcher: When operations complete, the Asynchronous Operation Processor retrieves the Completion Handler and Completion Dispatcher that were specified when the operation was initiated. The Asynchronous Operation Processor then passes the Completion Dispatcher the result of the Asynchronous Operation and the Completion Handler to call back*
- *Completion Dispatcher notifies the application: The Completion Dispatcher calls the completion hook on the Completion Handler, passing it any completion data specified by the application*

Proactor


Proactive Initiator
Asynchronous Operation Processor
Asynchronous Operation
Completion Dispatcher
Completion Handler
register (operation, handler, dispatcher)
Asynchronous operation initiated
execute
Operation performed asynchronously
dispatch
Operation completes
handle event
Completion Handler notified

Proactor

- Utilisations connues
  - *I/O Completion Ports* et *Asynchronous Procedure Calls* dans Windows NT
  - La famille UNIX AIO de *Asynchronous I/O Operations*
  - ACE Proactor
  - …

Proactor

- Conséquences
  - Bénéfices
    - Séparation des intérêts
      - Une interface par intérêt
    - Augmente la modularité, reutilisabilité, portabilité et configurabilité
    - Séparation entre exécution et gestion de la concurrence
    - Augmentation de la performance
    - Simplification de la synchronisation
  - Points négatifs
    - Difficile à déboguer
    - Planification et contrôle des tâches difficile et délicat

Asynchronous Completion Token

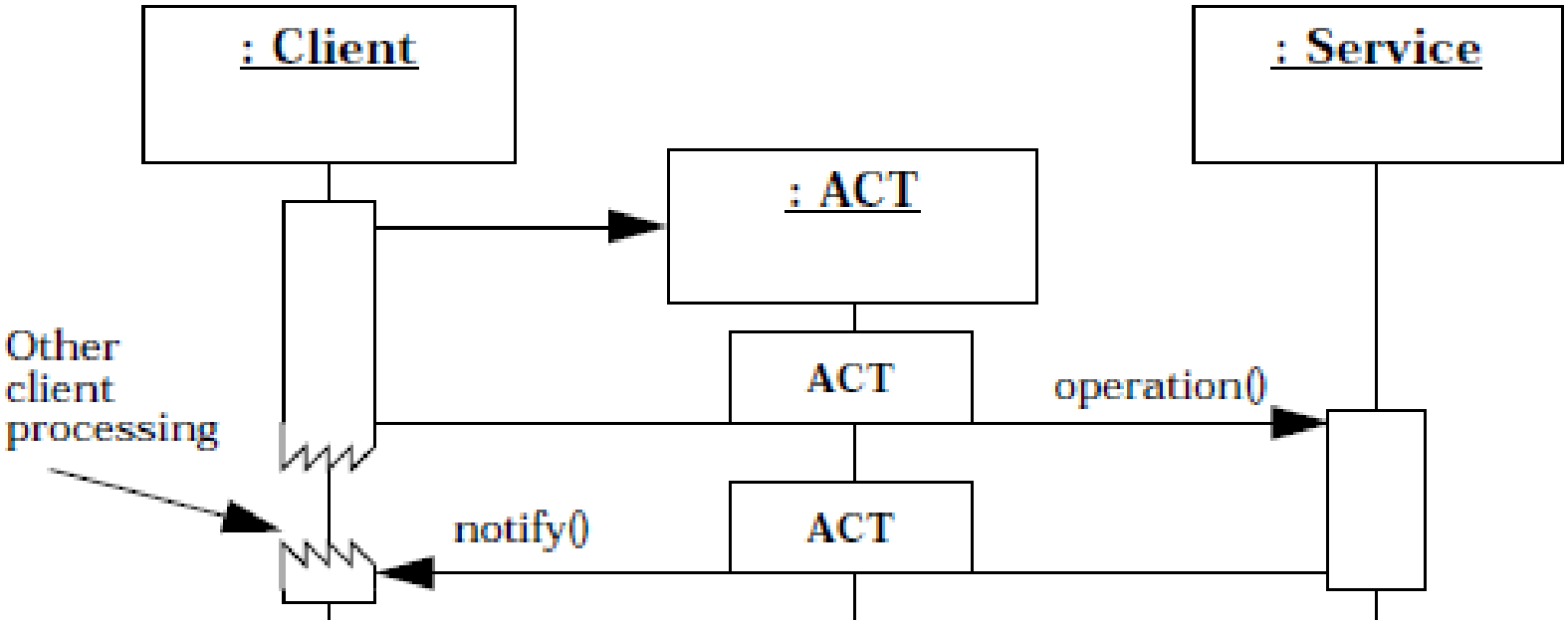
- Objectif
  - Permettre l'invocation asynchrone des requêtes de services
  - Le services signalent leur fin d'exécution au client
  - Le client doit pouvoir réaliser des traitements suite à ces réponses
- Problème
  - Un client doit pouvoir invoquer une ou plusieurs opérations asynchrones
  - Le client doit être averti par les services lorsque les opérations asynchrones sont terminées
  - Un mécanisme de demultiplexage doit être implémenté pour associer les réponses des services aux actions à réaliser par le client
  - Le temps de latence doit être minimal et le débit maximal

Asynchronous Completion Token

- Solution
  - Pour chaque opération asynchrone invoquée par le client sur un service, créer un *asynchronous completion token (ACT)* qui identifie de façon unique les actions et états nécessaire à la poursuite de l'exécution de l'opération
  - Fournir cet ACT en même temps que la requête d'opération au service
  - Cet ACT doit être inclus dans la réponse du service
  - Le client pourra donc grâce à cet ACT identifier les conditions de l'exécution de son opération pour terminer l'exécution de celle-ci

Asynchronous Completion Token

- Les collaborations sont



Asynchronous Completion Token

- Utilisations connues
  - Utiliser dans la majorité des OS
  - Autres utilisations
    - CORBA demultiplexing
    - POSIX
    - EMIS network management
    - FedEx inventory tracking…

Asynchronous Completion Token

- Conséquences
  - Bénéfices
    - Simplifie les structures de données pour le client
      - Le ACT retourne au client toute l'information nécessaire pour son exécution
    - Augmente la performance, la flexibilité, et l'efficacité
  - Points négatifs
    - Risques de fuites de mémoire
      - Si le client utilise les ACT comme des pointeurs, en cas d'échec des services, la mémoire n'est pas désallouée

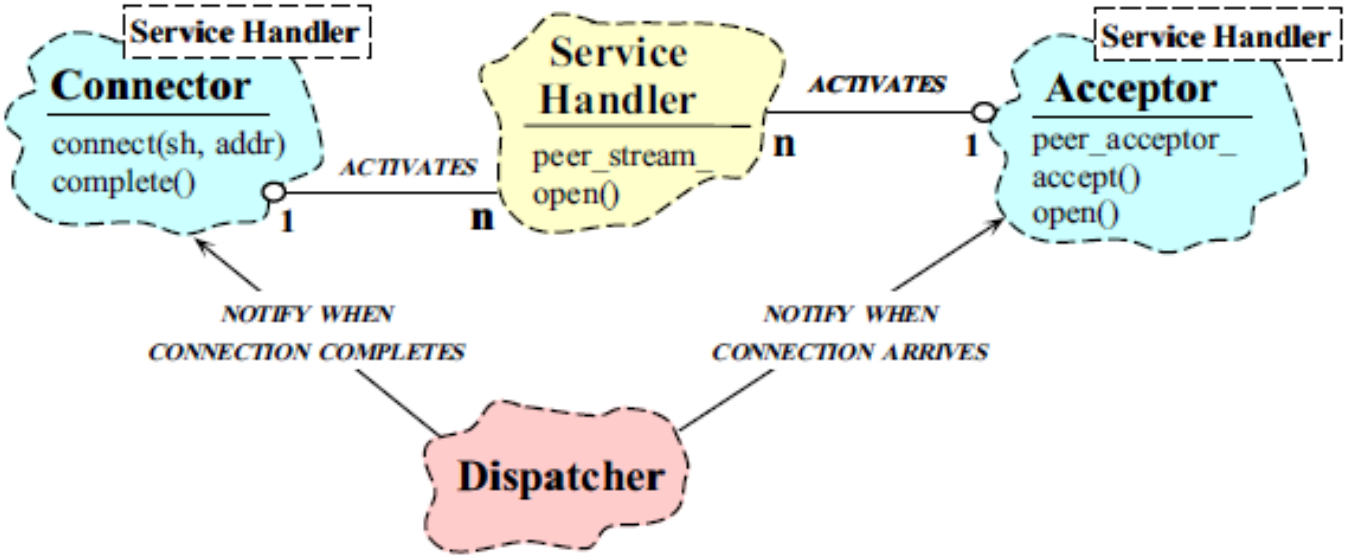
Acceptor-Connector

- Objectif
  - Permettre la séparation connexion et initialisation de services et l'exécution de ces services dans une application distribuée
- Problème
  - Les stratégies de connexion/initialisations changent très peu comparées aux implémentations de services
  - L'ajout de nouveaux types de services et de nouvelles implémentations ne doit pas affecter les connexions déjà établies ni les services déjà initialisés

Acceptor-Connector

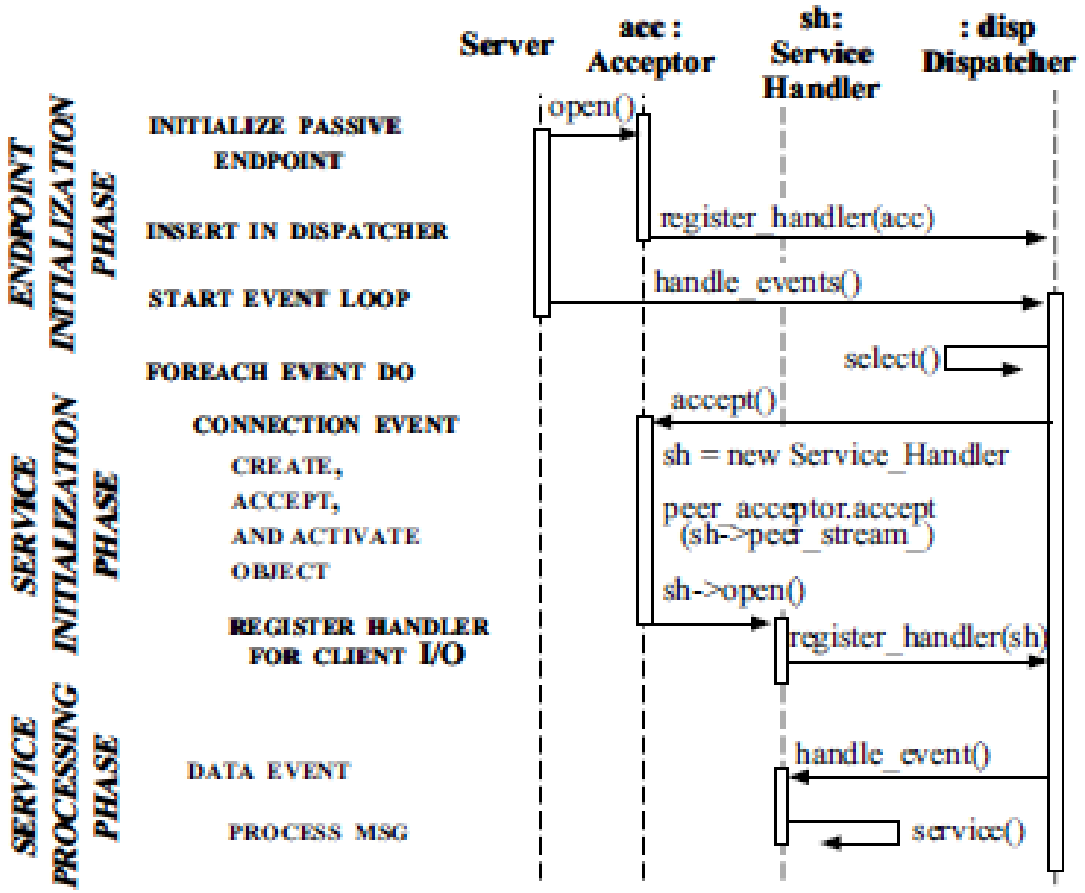
- Solution
  - Implémenter un *Service Handler* implémentant un service de l'application et offrant une méthode utiliser par les *Acceptor* / *Connector* pour activer le service une fois la connexion établie
  - Implémenter une usine *Acceptor* pour établir les connexions et initialiser les services associes (*ServiceHandler*) passivement
    - Lorsqu'une requête de connexion arrive l'*Acceptor* crée un *Service Handler*

Acceptor-Connector



Acceptor-Connector

- Les collaborations sont



Acceptor-Connector

- Utilisations connues
  - UNIX network superservers, CORBA ORBs, ACE Framework
- Conséquences
  - Bénéfices
    - Augmente la reutilisabilité, l'extensibilité, la robustesse
    - Permet une utilisation efficace du parallélisme
  - Points Négatifs
    - Indirections supplémentaires
    - Complexité supplémentaire

Bibliographie

- Douglas C. Schmidt, Wrapper Facade A Structural Pattern for Encapsulating Functions within Classes
- Douglas C. Schmidt, Extension Interface
- Bruno Bachelet, Patrons de conception Design Patterns, http://www.nawouak.net
- Haydar Aydin, Component Configurator Design Pattern

Bibliographie

- Douglas C. Schmidt, Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events
- Irfan Pyarali, Tim Harrison, and Douglas C. Schmidt, Thomas D. Jordan, Proactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events
- Douglas C. Schmidt, Asynchronous Completion Token
- Douglas C. Schmidt, Acceptor-Connector, An Object Creational Pattern for Connecting and Initializing Communication Services
- http://bosy.dailydev.org/2007/04/interceptor-design-pattern.html