Enterprise Integration Patterns

Les EIP (*Enterprise Integration Patterns*) sont issus de l'excellent livre éponyme de G. Hohpe et B. Woolf chez Addisson Wesley.

EIP? Pour quoi faire?

La notion de couplage faible, que ce soit au niveau applicatif ou entre applications, a toujours été primordiale pour, entre autre, permettre une meilleure évolutivité (et tous ce qui va avec) qui est une des bêtes noires pour tous DSI et même pour tous développeurs.

- utilisation d'un répertoire tampon sur le système de fichiers,
- base de données partagées,
- RPC (*Remote Procedure Call*),
- envoie de messages via un MOM (*Middleware Oriented Message*),
- …

Les EIP sont des patterns qui permettent de normaliser les échanges de messages dans un système asynchrone. Comme beaucoup de patterns, ces notions sont généralement connues et les EIP ne font que récapituler, normaliser et nommer de nombreux concepts comme ce que doit contenir les messages qui transitent ou ce qu'il faut faire pour router un message. En outre, il formalise graphiquement la représentation de ses concepts, ce qui permet avec un schéma de se faire comprendre par tous.

Ses applications sont multiples : cela peut aussi bien être dans un middleware que dans un contrôleur (au sens MVC du terme), et ils sont d'ailleurs souvent présent dans les EAI (généralement au travers d'un outil graphique). Aussi, les EIP trouvent parfaitement leurs places dans une infrastructure SOA comme un ESB que ce soit de manière transparente ou apparente.

Concepts des EIP

Les EIP s'appuient sur une architecture Pipes and Filters : un pipe et un Filter pouvant être vu comme un composant qui fait quelque chose (dans certaines implémentations, ce Filter est aussi appelé Processor).

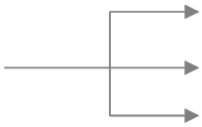
En outre, les EIP décrivent :

- ce qui transite dans le Pipe entre les Filter : il s'agit des **Messages** et quel peut et doit être son contenu en fonction de son cas d'utilisation
- sur le type de **Pipe** sur lequel les messages peuvent transiter : il s'agit des **Channel** (canaux)
- et introduisent les notions de **Routing**, **Transformation** et de**Message Endpoint** qui permettent de classifier les Filters.

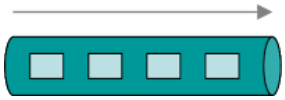
Notion de Channel

Un **Channel** correspond au Pipe dans l'architecture Pipes and Filters et est le canal où transitent tous les messages.

Le **Publish-Subscribe Channel** qui est un canal où un message est consommé par tous les consommateurs qui se sont inscrits comme étant intéressés par les messages du canal. Il est à noter qu'une fois que le message a été consommé par tous les Filter, il n'est plus accessible (même fonctionnement que le Design Pattern Observer du GoF). Ce canal est souvent utilisé pour faire de la notification.

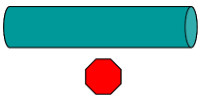


Le **Datatype Channel** qui est un canal où tous les messages qui y transitent sont de même type.

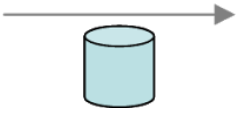


Le **Message Invalid Channel** est un canal qui permet de véhiculer les messages non valides (au sens applicatif du terme : manque d'information, mauvais format, …) et qui ont besoin d'un traitement particulier. Ces messages sont considérés comme invalide par les Filter qui, plutôt que de les ignorer ou de les réémettre sur un canal "normal" ont alors la possibilité d'utiliser ce canal particulier pour, par exemple, pour le logger ou faire un traitement sur erreur.

Le **Dead Letter Channel** est un canal qui permet de rerouter des messages non valides, au sens technique du terme (canal d'émission mal configuré, canal supprimé avant réception du message, message expiré, …). Ces le système de messagerie qui doit prendre en compte ce type de problème et qui doit gérer, s'il existe, ce Dead Letter Channel.



Le **Garanteed Delivery Channel** qui est un canal qui conserve les messages dans un espace de stockage persistant (base de données, système de fichiers, …) tant qu'il n'a pas été transmis ou tant qu'il n'a pas été stocké dans un autre espace de stockage. Ce canal permet de s'assurer de la non perte de données dans le système en cas d'anomalie de ce dernier (arrêt, …).



En outre, les EIP décrivent également dans cette section des notions transverses aux canaux qui permettent leur interaction (liste non exhaustive) :

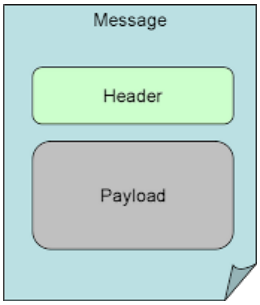
- Le Channel Adapter qui permet à une ou des applications quelconques de recevoir ou d'émettre des messages de ou vers le système de messagerie.
- Le Messaging Bridge qui permet d'interfacer plusieurs systèmes de messagerie.

On peut constater que, généralement, ces différents points ne sont pas nouveaux et qu'ils sont présent dans de nombreux produits, technologies et spécification sous des noms différents (JMS pour les Point-to-Point – *Queue* – ou les Publish-Subscribe –*Topic* -, PABX dans le monde des télécoms pour le Messaging Bridge, …).

Notion de Message

Les EIP décrivent les messages qui peuvent transiter dans les canaux au niveau structurel et fonctionnel.

- un entête
- des propriétés
- un corps (body)
- une pièce jointe



Ils décrivent également ce que peut ou devrait contenir l'entête ou les propriétés du message en fonction de son cas d'utilisation :

- **Return Address** qui permet lorsque le message est utilisé comme requête dans le mécanisme de requête/réponse de connaître à qui (c'est à dire le canal) la réponse doit être renvoyé (s'il connait le canal alors le ou les destinataires sont connus au travers des Filter consommateurs ou en écoute du canal). Cette information est généralement mise dans l'entête du message.
- **Correlation Identifier** qui permet d'associer un identifiant au message afin de, par exemple, l'associer la réponse à sa requête lors d'un échange asynchrone.

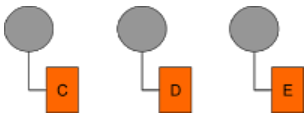
- **Message Sequence** qui permet de reséquencer des messages pouvant être issus d'un même message découpé en raison de sa taille ou en raison de la nécessité d'en émettre des parties à des Filter différents puis de les réagréger. Ce type d'information est généralement mis dans l'entête du message et doit dans la plupart des cas disposer de 3 choses : l'identifiant de la séquence, l'identifiant de position et le nombre de fraction ou un identifiant de fin (flag).
- **Message Expiration** qui permet de spécifier une durée de validité (en temps absolue ou relative) du message. S'il est expiré, il peut, par être exemple, être rerouté vers un

Dead Letter Channel par le système de messagerie ou vers un Invalid Message Channel s'il a été reçu par un Filter. Cette information est généralement mise dans l'entête du message.

- **Format Indicator** qui permet de spécifier le format de la donnée (numéro de version, clé d'identification, format de la donnée comme son schéma xsd, …). Cette information peut être mise soir dans l'entête du message, soir dans le corps de ce dernier.

Enfin, les EIP classifient les messages (qui peuvent être utilisés dans des échanges synchrone ou asynchrone) par type d'utilisation :

- **Command Message** qui sont des messages utilisés pour invoquer une méthode ou une procédure.
- **Document Message** qui sont des messages utilisés pour transmettre des données en laissant aux consommateurs de ces derniers le soin de décider de leur traitement.
- **Event Message** qui sont des messages utilisés pour émettre un événement et qui peut se décliner en deux modèles :
    - **Push Model** : le message est une combinaison du contenu du nouvel élément et de l'événement.
    - **Pull Model** : le message est juste une notification devant être suivie d'une demande du nouvel élément par le consommateur via un Command Message qui reçoit alors un Document Message de l'émetteur.



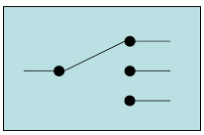
Notion de Message Routing

Un **Message Routing** correspond à une catégorie de Filter dans l'architecture Pipes and Filters. Ce type de Filter permet de gérer les problématiques de routage en offrant des composants de routage, mais aussi de filtrage (qui peut être considérer un peu comme un routeur), de découpage (un routage de 1 vers n), d'agrégation ou de reséquencement. En outre, on peut les classifier en deux catégories : les basiques et les composés qui composent les Message Routing basiques pour offrir un patron plus complet.

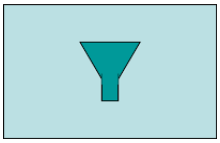
Les EIP décrivent les Message Routing "simple" suivants :

- De 1 vers 1 :

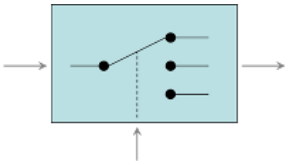
Content-Based Routing qui permet de router les messages en fonction du contenu de ces derniers en fonction de règles prédéfinies.



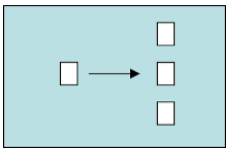
Message Filter qui permet de filtrer les messages en fonction de règles prédéfinies. Il est à noter qu'il peut être sans état ou avec état (par exemple dans le cas où il est nécessaire de supprimer les messages en doublons).



Dynamic Router qui permet de router les messages en fonction de critères fournis par un composant tierce pouvant être renseigné dynamiquement.

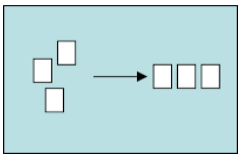


Splitter qui permet de découper un message en fonction de règles pouvant être définies statiquement ou dynamiquement. Il est à noter qu'il peut également adjoindre au message des propriétés comme Message Sequence et Correlation Identifier en prévision d'une éventuelle réagrégation du message découpé.

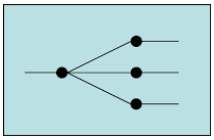


Resequencer qui permet de réordonnancer une série de message. C'est un Filter à état qui nécessite, généralement le Message Sequence et qui peut soit émettre les messages au fils de l'eau (dès qu'il reçoit le message n+1, il l'émet) soit attendre d'avoir reçu tous les messages à réordonner. Il doit pouvoir :

- stocker les messages ne correspondant pas au suivant,
- gérer l'arrivée d'un nouveau message même si un ré-ordonnancement est en cours,
- gérer l'arrivée d'un message ayant déjà été pris en compte dans un processus de ré-ordonnancement,
- gérer son buffer afin d'éviter un dépassement de capacité.



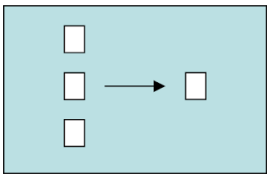
- De 1 vers n :
    - **Recipient List** qui permet de multiplexer un message vers un ensemble de consommateurs en fonction de règles pouvant être basées sur le message ou pouvant être fourni par un composant tierce. Il est à noter que le Recipient List est responsable de l'émission des messages et qu'il doit donc être garant que le message a bien été reçu par tous les destinataires.



- De n vers 1 :
    - **Aggregator** qui permet de ré-agréger des message en un seul. C'est un Filter à état et il peut disposer des stratégies suivantes :
        - Wait for All : attente de tous les messages,
        - Timeout : attente pendant un laps de temps donné,
        - First Best : transmission de la première fraction de message reçu et ignorance des autres,
        - Timeout with Override : attente pendant un laps de temps donné et transmission de la meilleure fraction de message,
        - External Event : attente jusqu'à réception d'un événement extérieur.

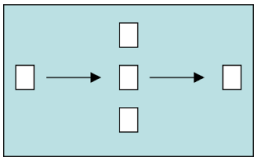
et qu'il doit pouvoir :

- décider du moment où un message est entièrement reçu, et de ce qu'il doit faire sinon,
- décider de la manière dont il doit agréger les messages reçus,
- gérer l'arrivée d'une fraction d'un nouveau message même si une agrégation est en cours,
- gérer l'arrivée d'une fraction d'un message ayant déjà été agrégé et émis.



En outre, les EIP décrivent les Message Routing "composite" suivants :

- **Composed Message Processor** qui permet successivement de recevoir un message, de le découper, de l'émettre à différents composants puis d'agréger le résultat de ces derniers dans un unique message.



- **Scatter-Gather** qui permet successivement de recevoir un message, de l'émettre à différents composants puis d'agréger le résultat de ces derniers dans un unique message.
- **Routing Slip** qui permet d'utiliser l'architecture du Pipes and Filters dans un unique composant, c'est-à-dire chaîner linéairement les appels vers différents filters en masquant les composants appelés et en minimisant les points de contrôle.
- **Process Manager** qui permet d'invoquer, dans un unique composant, un ensemble d'opérations s'exécutant de manière non linéaire en minimisant les points de contrôle. Il s'agit d'un composant à état qui doit connaître l'étape du processus et qui est être capable de gérer plusieurs instances de processus (utilisation d'un identifiant de corrélation). Il est à noter que pour répondre à ces besoins, certains langages ont émergés tels que :
    - XLANG,
    - WSFL (*Web Services Flow Language*),

    - BPEL (*Business Process Execution Language*).
- **Message Broker** qui permet de découpler les destinataires d'un message de l'émetteur en utilisant un unique point de contrôle qui permet de déterminer le consommateur d'un message.

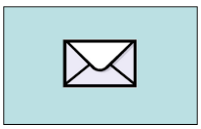
Il est à noter que beaucoup des fonctionnalités des Message Routing décrits précédemment peuvent également être traitées en utilisant différents types de Message Channel, en spécialisant le consommateur ou en combinant d'autres Messages Routing (par exemple, le Content-Based Router peut être remplacé en utilisant un Publish/Subscribe Channel en combinaison de Message Filter) : les critères de choix sont donc à étudier avec attention car cela peut impacter la maintenabilité du système ainsi que ses performances. En outre, donner connaissance de la logique aux composants peut entraîner un couplage plus fort.

Notion de Message Transformer

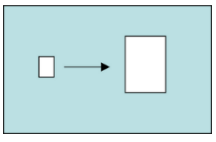
Un **Message Transformer** correspond à une catégorie de Filter dans l'architecture Pipes and Filters. Ce type de Filter permet de gérer les problématiques de transformation et d'encapsulation.

Les EIP décrivent les Message Transformer suivants :

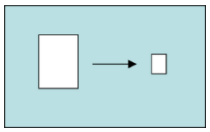
- **Envelope Wrapper** qui permet d'encapsuler un message ou un objet dans un message compatible avec le système de messagerie.



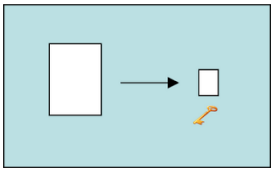
- **Content Enricher** qui permet d'ajouter des informations au message transmis. Ces informations peuvent provenir :
    - du Content Enricher lui-même,
    - de l'environnement système,
    - d'une entité tierce.



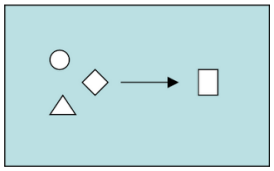
- **Content Filter** qui permet d'élaguer le contenu d'un message (confidentialité, …) ou de simplifier sa structure.



- **Claim Check** qui permet de diminuer temporairement la taille d'un message en stockant les informations (système de fichiers, base de données) qui peuvent alors être récupérées ultérieurement. Il nécessite, généralement, la génération d'un identifiant permettant de récupérer l'information stockée.



- **Normalizer** qui permet de modifier le format d'un message.



Notion de Message Endpoint

Un **Message Endpoint** correspond à une catégorie de Filter dans l'architecture Pipes and Filters. Ce type de Filter permet de gérer les problématiques d'interconnexion avec l'extérieur du système de messagerie.

Les Message Endpoint se décomposent en 2 catégories :

- **Send and Receive Patterns** qui s'appliquent aux émetteurs et récepteurs et qui permet au code de l'application de s'interfacer, via une couche de code permettant également de transformer les données afin qu'elles soient compatibles, avec le système de messagerie. Cette couche peut également gérer le coté transactionnelle.
- **Message Consumer Patterns** qui s'appliquent aux consommateurs de messages et qui peuvent différer sur des critères comme la manière de lire un message dans le canal ou la possibilité de recevoir tous les messages.

Il est à noter que le **Message Producer Patterns** n'est pas défini car la procédure d'émission est simple et ne nécessite pas de mécanisme particulier.

Les EIP décrivent les Message Endpoint suivants :

- **Messaging Gateway** qui permet d'encapsuler l'accès au système de messagerie du reste de l'application en fournissant, généralement une API permettant d'émettre et de recevoir des messages de manière synchrone ou asynchrone et positionnant automatiquement, au besoin, les propriétés utiles aux messages. En outre, généralement, c'est à cette couche de gérer les exceptions pouvant être levées par le système de messagerie.
- **Messaging Mapper** qui permet de lier des objets et des messages compatibles avec le système de messagerie. Cette opération peut s'avérer complexe car, si la donnée est transmise sous forme d'objet, alors les composants du système deviennent dépendants du format (il est à noter que le Messaging Mapper peut être comparé au design pattern Mediator et utiliser les mécanismes de réflexion).
- **Transactional Client** qui permet de gérer les transactions dans le système de messagerie (dans le cas où il existe plusieurs consommateurs, …).
- **Polling Consumer** qui permet, pour un consommateur, de récupérer un message du canal en scrutant ce dernier périodiquement ou lorsqu'il est disponible. Il est aussi dénommé Synchronhous receiver et permet de ne consommer qu'un seul thread.

- **Event-Driven Consumer** qui permet, pour un consommateur, de récupérer un message dès qu'il est disponible sur le canal. Il est aussi dénommé Asynchronous receiver et fonctionne sur le mécanisme de callback.
- **Competing Consumers** qui permet, lorsque le système nécessite une grande réactivité, en dispatchant les messages d'un Point-to-Point Channel, de les consommer de manière concurrente en associant un thread à chacun de consommateurs.
- **Message Dispatcher** qui permet de dispatcher les messages d'un canal pour les consommer par différents consommateurs en leur associant un thread à chacun. Le Message Dispatcher peut utiliser des propriétés pour sélectionner le consommateur adéquat. Généralement, les consommateurs sont dans le même processus que le Message Dispatcher contrairement, au Competing Consumers.
- **Selective Consumer** qui permet de ne consommer que certains messages en provenance d'un canal.
- **Durable Subscriber** qui permet de recevoir tous les messages d'un Publish/Subscribe Channel même lorsqu'il n'est pas connecté (on utilise alors le terme d'inactif puisqu'il nécessite quand même un abonnement).
- **Indempotent Receiver** qui permet de ne recevoir qu'une seule instance d'un même message. Il doit être à état et doit donc gérer la purge de l'historique.
- **Service Activator** qui permet d'invoquer une application prévue pour s'interfacer ou pas avec le système de messagerie. Un Service Activator peut être unidirectionnel (requêtes seules) ou bidirectionnel (requête/réponse) et peut, soit invoquer toujours le même service (application), soit utiliser des mécanismes de réflexion. Il permet d'invoquer, suite à la réception d'un message, l'application comme un client de manière à masquer complètement à l'application le système de messagerie.

Conclusion sur les EIP

Comme il a été vu, les EIP ne réinventent pas la roue mais permettent de donner un nom à des fonctionnalités ainsi qu'une représentation graphique. Cela peut être utile pour être compris de tous si tout le monde partage le même langage (un peu comme les Design Patterns).

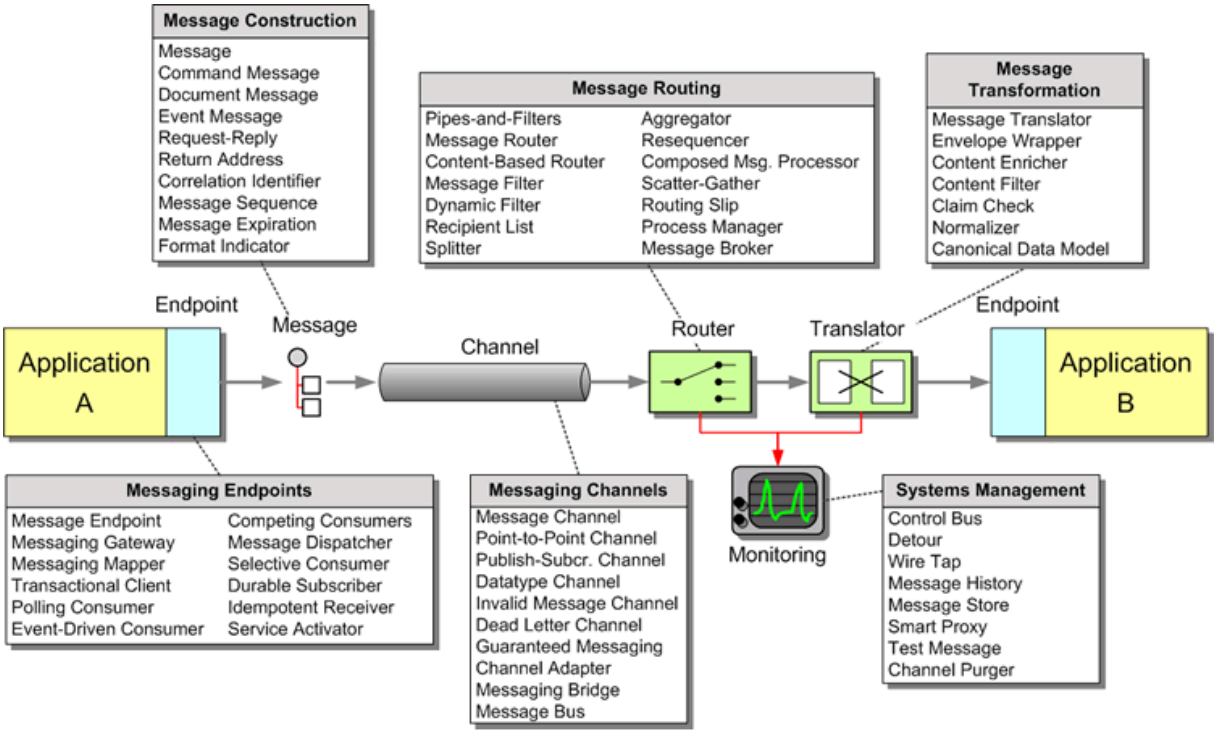
Ils ne proposent pas de solutions clé en main mais mettent en avant des notions qu'il est nécessaire d'avoir dans certains cas d'usage (comme l'identifiant de corrélation pour l'agrégation ou le reséquencement).

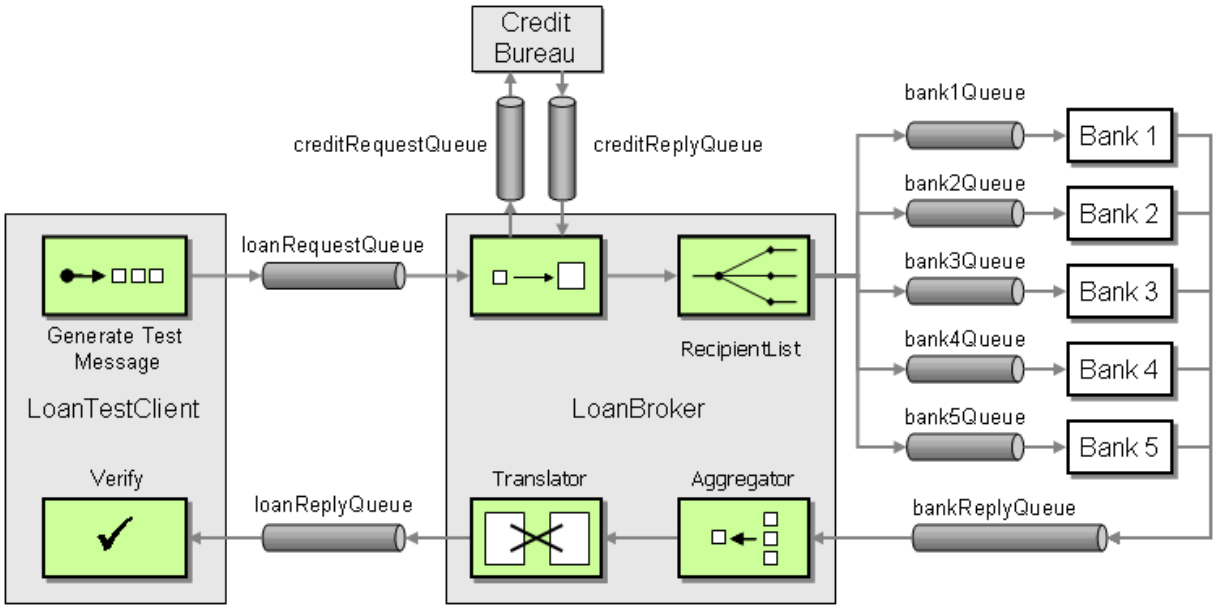
Il est essentiel de connaitre ces concepts lorsque l'on manipule des messages dans un système asynchrone et que des notions de médiation apparaissent.

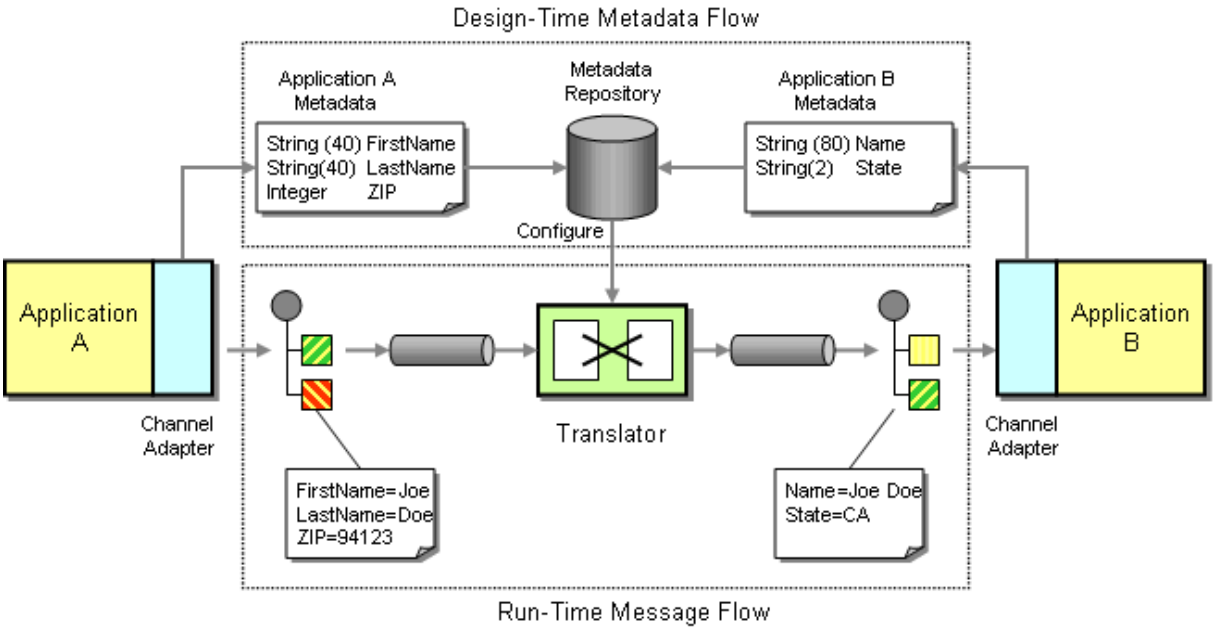
Les EIP trouvent donc parfaitement leurs places dans une architecture SOA et plus précisément dans les ESB.

Au niveau de ses implémentations, on trouve l'excellent Spring Integration ainsi que Apache Camel et l'ESB MuleSoft : une attention toute particulière doit être apportée quand à la mise en œuvre de ces concepts.

Quelques « Use Case »






Design-Time Metadata Flow
Application A Metadata
String (40) FirstName
String(40) LastName
Integer ZIP
Metadata Repository
Application B Metadata
String (80) Name
String(2) State
Configure
Application A
Channel Adapter
Translator
FirstName=Joe
LastName=Doe
ZIP=94123
Name=Joe Doe
State=CA
Channel Adapter
Application B
Run-Time Message Flow