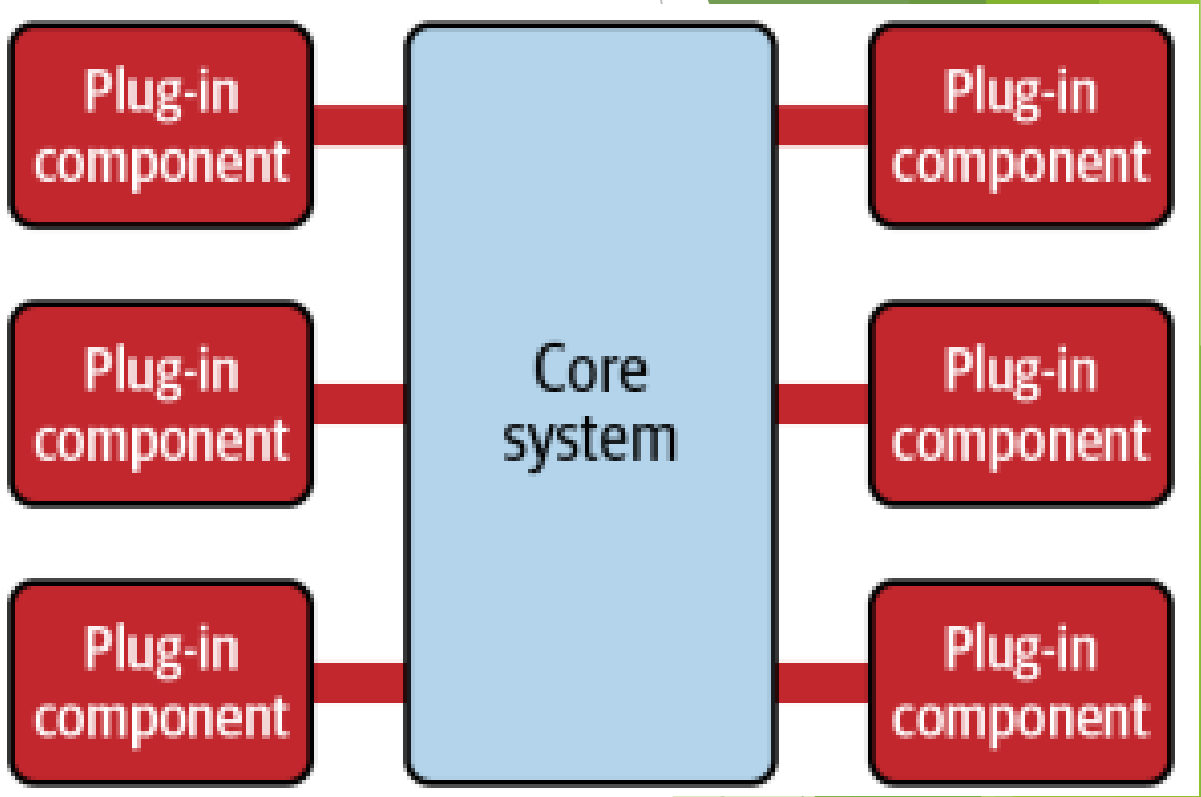
Chapitre 3
Architecture Microkernel (*Plug-in Architecture*)

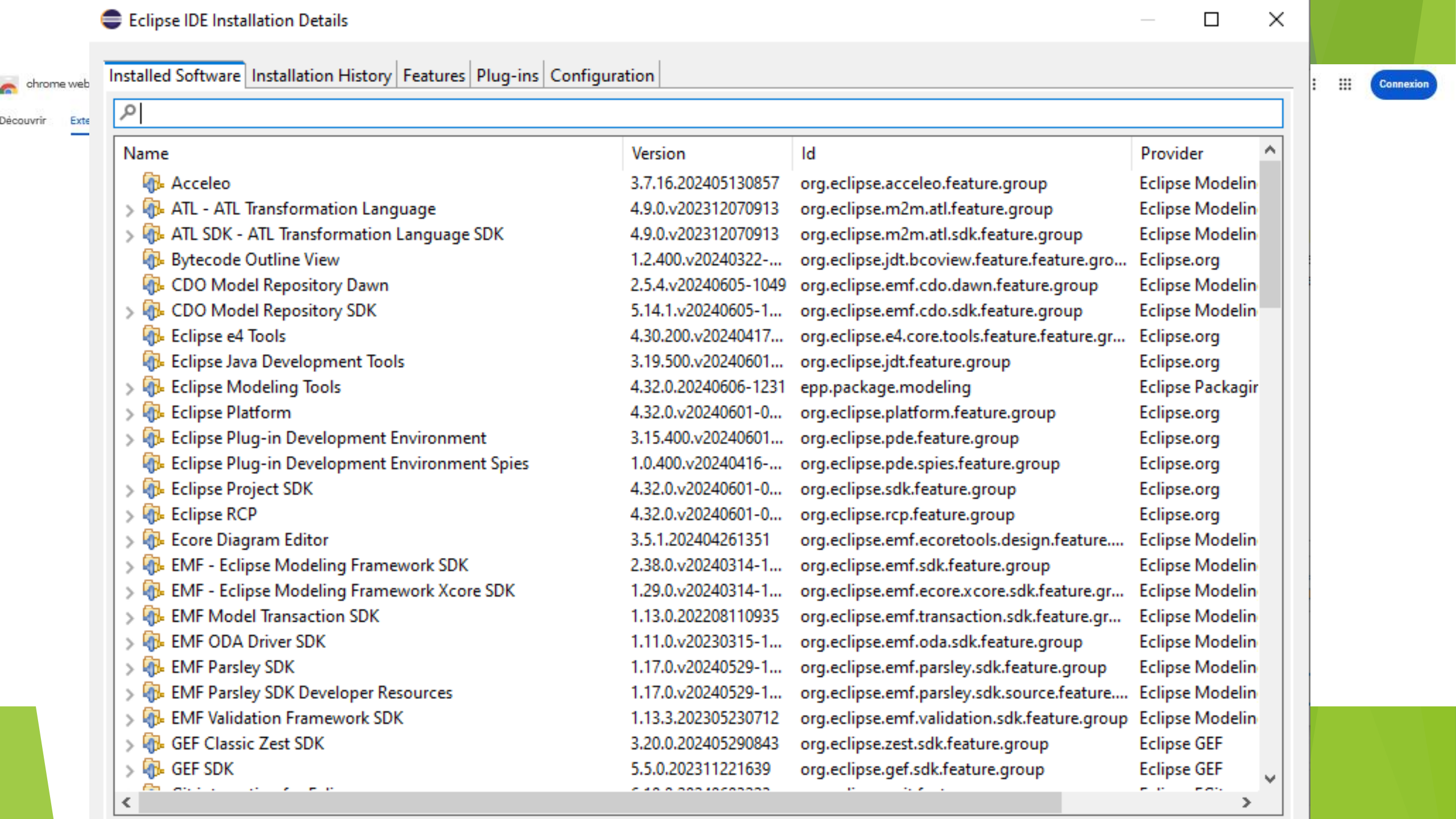
- Le style d’architecture microkernel constitue une architecture à la fois flexible et extensible, permettant à un développeur ou à un utilisateur final d’ajouter aisément des fonctionnalités supplémentaires à une application existante sous forme d’extensions ou de « plug-ins », sans altérer les fonctionnalités fondamentales du système.
- Pour cette raison, l’architecture microkernel est parfois désignée sous l’appellation d’« architecture à plug-ins », autre dénomination couramment employée pour qualifier ce style architectural.

- Ce style d’architecture s’avère particulièrement adapté aux applications orientées produit — c’est-à-dire des applications empaquetées et distribuées sous forme de versions téléchargeables, à l’instar des logiciels proposés par des éditeurs tiers — mais il est également fréquemment adopté dans le cadre d’applications métiers internes développées sur mesure.
- De nombreux systèmes d’exploitation mettent en œuvre le style architectural microkernel, ce qui explique l’origine même de sa dénomination.

Topologie

- Le style d'architecture microkernel se compose de deux catégories de composants architecturaux :
  - un système central (core system) et
  - des modules d'extension (plug-ins).
- La logique applicative est répartie entre des modules d'extension indépendants et le noyau central du système, ce qui confère à l'architecture des propriétés d'extensibilité, de flexibilité et d'isolation des fonctionnalités.




Eclipse IDE Installation Details
Installed Software
Installation History
Features
Plug-ins
Configuration
Name
Version
Id
Provider
Acceleo
3.7.16.202405130857
org.eclipse.acceleo.feature.group
Eclipse Modelin
ATL - ATL Transformation Language
4.9.0.v202312070913
org.eclipse.m2m.atl.feature.group
Eclipse Modelin
ATL SDK - ATL Transformation Language SDK
4.9.0.v202312070913
org.eclipse.m2m.atl.sdk.feature.group
Eclipse Modelin
Bytecode Outline View
1.2.400.v20240322-...
org.eclipse.jdt.bcoview.feature.feature.gro...
Eclipse.org
CDO Model Repository Dawn
2.5.4.v20240605-1049
org.eclipse.emf.cdo.dawn.feature.group
Eclipse Modelin
CDO Model Repository SDK
5.14.1.v20240605-1...
org.eclipse.emf.cdo.sdk.feature.group
Eclipse Modelin
Eclipse e4 Tools
4.30.200.v20240417...
org.eclipse.e4.core.tools.feature.feature.gr...
Eclipse.org
Eclipse Java Development Tools
3.19.500.v20240601...
org.eclipse.jdt.feature.group
Eclipse.org
Eclipse Modeling Tools
4.32.0.20240606-1231
epp.package.modeling
Eclipse Packagir
Eclipse Platform
4.32.0.v20240601-0...
org.eclipse.platform.feature.group
Eclipse.org
Eclipse Plug-in Development Environment
3.15.400.v20240601...
org.eclipse.pde.feature.group
Eclipse.org
Eclipse Plug-in Development Environment Spies
1.0.400.v20240416-...
org.eclipse.pde.spies.feature.group
Eclipse.org
Eclipse Project SDK
4.32.0.v20240601-0...
org.eclipse.sdk.feature.group
Eclipse.org
Eclipse RCP
4.32.0.v20240601-0...
org.eclipse.rcp.feature.group
Eclipse.org
Ecore Diagram Editor
3.5.1.202404261351
org.eclipse.emf.ecoretools.design.feature....
Eclipse Modelin
EMF - Eclipse Modeling Framework SDK
2.38.0.v20240314-1...
org.eclipse.emf.sdk.feature.group
Eclipse Modelin
EMF - Eclipse Modeling Framework Xcore SDK
1.29.0.v20240314-1...
org.eclipse.emf.ecore.xcore.sdk.feature.gr...
Eclipse Modelin
EMF Model Transaction SDK
1.13.0.202208110935
org.eclipse.emf.transaction.sdk.feature.gr...
Eclipse Modelin
EMF ODA Driver SDK
1.11.0.v20230315-1...
org.eclipse.emf.oda.sdk.feature.group
Eclipse Modelin
EMF Parsley SDK
1.17.0.v20240529-1...
org.eclipse.emf.parsley.sdk.feature.group
Eclipse Modelin
EMF Parsley SDK Developer Resources
1.17.0.v20240529-1...
org.eclipse.emf.parsley.sdk.source.feature....
Eclipse Modelin
EMF Validation Framework SDK
1.13.3.202305230712
org.eclipse.emf.validation.sdk.feature.group
Eclipse Modelin
GEF Classic Zest SDK
3.20.0.202405290843
org.eclipse.zest.sdk.feature.group
Eclipse GEF
GEF SDK
5.5.0.202311221639
org.eclipse.gef.sdk.feature.group
Eclipse GEF

- Les modules d'extension (plug-ins) sont des composants autonomes et indépendants qui encapsulent des traitements spécialisés, des fonctionnalités additionnelles, des logiques d'adaptation ou du code spécifique destiné à enrichir ou à étendre le système central afin de fournir des capacités métiers supplémentaires.
- En règle générale, les modules d'extension doivent demeurer indépendants les uns des autres et ne pas dépendre d'autres plug-ins pour assurer leur fonctionnement.
- Il est également essentiel, dans ce style architectural, de limiter au strict minimum les communications entre plug-ins, afin d'éviter des problèmes de dépendances complexes et difficiles à maîtriser.

- Le système central doit être en mesure d'identifier les modules d'extension disponibles ainsi que les modalités d'accès à ceux-ci.
- Une approche couramment adoptée consiste à mettre en œuvre un registre de plug-ins (plug-in registry).
- Ce registre contient des informations relatives à chaque module d'extension, notamment son nom, les détails de son contrat (interface), ainsi que les spécifications du protocole d'accès distant, en fonction du mode de connexion au système central.
- Dans les cas où les contrats et les protocoles d'accès sont standardisés au sein du système, le registre peut se limiter à contenir le nom du module d'extension ainsi que le nom de l'interface permettant son invocation.

Exemple de plugin.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<plugin
    id="com.example.myplugin"
    name="My First Plugin"
    version="1.0.0"
    provider-name="Example Corp">

    <!-- Déclaration d'un point d'extension -->
    <extension-point
        id="sampleExtensionPoint"
        name="Sample Extension Point"
        schema="schema/sample.exsd"/>

    <!-- Utilisation d'un point d'extension existant -->
    <extension point="org.eclipse.ui.views">
        <view
            id="com.example.myplugin.views.SampleView"
            name="Sample View"
            class="com.example.myplugin.views.SampleView"
            category="com.example.myplugin.category"/>
    </extension>

    <!-- Dépendances -->
    <requires>
        <import plugin="org.eclipse.ui"/>
        <import plugin="org.eclipse.core.runtime"/>
    </requires>

</plugin>
```

Informations générales

- Identifie le plugin dans le plugin registry
- Sert à la gestion des versions et conflits

Point d'extension

- Le plugin offre un point d'extension
- D'autres plugins pourront s'y connecter

Extension (utilisation d'un point existant)

- Le plugin utilise un point d'extension fourni par Eclipse
- Ajoute une vue dans l'interface Eclipse

Dépendances

Indique que ce plugin dépend de :

- UI Eclipse
- Runtime Eclipse

- Les modules d'extension peuvent être
central selon diverses modalités.
- Traditionnellement, les plug-in
de bibliothèques ou de modul
JAR ou DLL), connectés au no
par exemple via un appel de m
- Ces modules indépendants p
frameworks (de modules) spécialisés
Service Gateway Initiative), **Project Jigsaw**, **Penrose**, **Prism** ou
encore les environnements **.NET**.

- Charger dynamiquement les modules
- Gérer les dépendances
- Gérer le cycle de vie
- Assurer la communication entre modules (par services, interfaces, …)

OSGi (Open Service Gateway Initiative)

- Définition : C’est une spécification et un cadre pour le développement d’applications Java modulaires.
- Concept clé : Elle permet de créer des modules Java qui peuvent être installés, démarrés, arrêtés ou mis à jour dynamiquement sans redémarrer l’application.
- Usage typique :
  - Applications serveur Java évolutives
  - Plateformes embarquées (IoT, télécommunications)
  - Eclipse IDE est construit sur OSGi pour gérer ses plugins.

- Lorsque les plug-ins sont déployés de cette manière, le modèle global de déploiement correspond à une architecture monolithique (déploiement unique).
- Dans le contexte de l'architecture microkernel utilisant des plug-ins point à point, il est courant d'ajouter un module en déposant un fichier dans un répertoire spécifique, puis en redémarrant l'application.
- Certaines applications reposant sur les frameworks précédemment mentionnés prennent également en charge des mécanismes dynamiques permettant l'ajout ou la modification de plug-ins à l'exécution, sans nécessiter le redémarrage du système central.

- En alternative (aux fichiers séparés (JAR/DLL)), les modules d'extension peuvent également être implémentés au sein d'une base de code unique et consolidée, se matérialisant simplement par une organisation spécifique en espaces de noms (namespaces) ou en packages.
- Dans une application dédiée au recyclage d'équipements électroniques, un plug-in chargé d'évaluer un appareil particulier — tel qu'un iPhone 17 — pourrait être structuré selon l'espace de noms suivant :

  app.plugin.assessment.iphone17

- Le second niveau de cet espace de noms indique explicitement qu'il s'agit d'un module d'extension (*plugin*), consacré en l'occurrence à l'évaluation d'un iPhone 17.
- De cette manière, bien que l'ensemble du code appartienne à une base unifiée, le code du plug-in demeure conceptuellement et structurellement distinct de celui du système central.

En java

```java
// Noyau
package core;

// Plugin 1
package plugins.auth;

// Plugin 2
package plugins.payment;
```

En C# (.NET namespaces)

Le noyau appelle les modules comme du code normal :

```
namespace Core { }

namespace Plugins.Logging { }
namespace Plugins.Security { }
```

- Pas de chargement dynamique : tout est déjà compilé ensemble
- Approche simple (pas de gestion complexe)
- Mais moins puissante que les architectures modulaires dynamiques (comme OSGi)

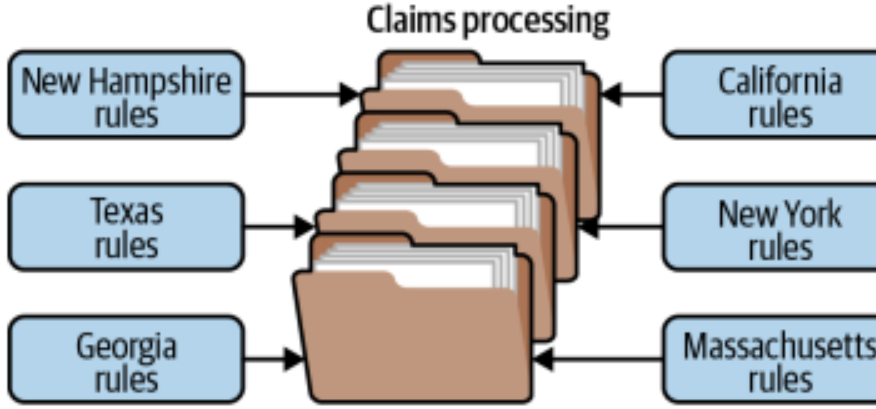
- Les modules d'extension peuvent également être implémentés sous forme de services distants, accessibles depuis le système central au moyen d'interfaces de type REST ou de mécanismes de messagerie.
- Dans une telle configuration, l'architecture microkernel est alors considérée comme une architecture distribuée.
- Dans ce cas, toutes les requêtes transitent toujours par le système central avant d'atteindre les modules d'extension.
- Toutefois, cette configuration:
  - facilite le déploiement dynamique des composants plug-ins
  - offrir une meilleure scalabilité interne (chaque plugin peut être **scalé** indépendamment)
  - une réactivité accrue, notamment lorsque plusieurs modules d'extension doivent être invoqués pour traiter une même requête métier.

Exemples

- Un exemple classique de l'architecture microkernel est **Eclipse IDE**. Le téléchargement de la version de base d'Eclipse ne fournit guère plus qu'un éditeur évolué. Toutefois, à mesure que l'on y ajoute des modules d'extension, l'environnement devient un outil hautement personnalisable et particulièrement performant pour le développement logiciel.
- Les navigateurs Internet constituent un autre exemple courant d'application de l'architecture microkernel : des visionneuses et divers plug-ins viennent enrichir les fonctionnalités du navigateur de base (le système central), en y intégrant des capacités qui ne sont pas disponibles nativement.
- De plus, de nombreux outils et produits destinés au développement et aux chaînes de déploiement — tels que **PMD (**analyseur de code statique), **Jira** (logiciel de gestion de projets) et **Jenkins** (un serveur d'automatisation) — sont également implémentés selon le style architectural microkernel.

Exemple d’un système de gestion des réclamations (compagnie d’assurance)

- Dans une compagnie d’assurance, le traitement des réclamations est complexe car les règles varient selon les juridictions (pays, États, régions). Utiliser un moteur de règles unique peut rendre le système très complexe, où la modification d’une règle peut affecter d’autres règles.
- Pour résoudre ce problème, l’architecture microkernel est utilisée :
  - Le noyau (core system) contient la logique métier de base du traitement des réclamations, qui change rarement.
  - Les règles spécifiques à chaque juridiction sont implémentées sous forme de modules plug-ins séparés.
  - Chaque plug-in peut être ajouté, modifié ou supprimé indépendamment sans affecter le noyau ni les autres modules.
- Idée principale : séparer la logique métier stable du système central des règles spécifiques et variables afin de faciliter la maintenance et l’évolution du système.



Analyse

- Le style d'architecture microkernel est très flexible et peut varier considérablement en termes de granularité.
- Ce style peut décrire l'architecture globale d'un système, ou bien être intégré et utilisé comme partie d'un autre style d'architecture.
- Par exemple, un microservice particulier peut être implémenté en utilisant le style d'architecture microkernel, tandis que d'autres services peuvent être implémentés d'une autre manière.

- Ce style d’architecture offre un excellent support pour la conception évolutive et le développement incrémental.
- Il est possible de produire un système central minimal (core system) qui fournit certaines des fonctionnalités principales du système. Ensuite, à mesure que le système évolue progressivement, de nouvelles fonctionnalités peuvent être ajoutées sans devoir apporter de modifications importantes au système central.

Conception évolutive Vs Développement incrémental Vs Développement itératif

Concept	Définition	Principe	Exemple
Conception évolutive	Approche de conception où l'architecture et le système évoluent progressivement avec les besoins.	On conçoit une architecture flexible qui peut être modifiée ou enrichie facilement.	Un système avec un noyau minimal auquel on ajoute des modules ou plug-ins au fil du temps.
Développement incrémental	Méthode de développement où le logiciel est construit par ajouts successifs de fonctionnalités appelés incréments.	Chaque version ajoute de nouvelles fonctionnalités au système existant.	Version 1 : gestion des tâches. Version 2 : ajout de la recherche. Version 3 : ajout des notifications.
Développement itératif	Méthode où le système est amélioré par répétitions successives (itérations).	On révise et améliore les fonctionnalités déjà existantes à chaque cycle.	Version 1 : interface simple. Version 2 : amélioration de l'interface. Version 3 : optimisation des performances.

- Conception évolutive → concerne surtout l'architecture et la manière de concevoir le système pour qu'il puisse évoluer.
- Développement incrémental → on ajoute progressivement de nouvelles fonctionnalités.
- Développement itératif → on améliore progressivement ce qui existe déjà.

- Selon la manière dont ce style d'architecture est implémenté et utilisé, il peut être considéré soit comme une architecture partitionnée techniquement, soit comme une architecture partitionnée par domaine.
- Par exemple :Utiliser des plug-ins pour fournir des fonctions d'adaptation ou des configurations spécifiques correspond à une architecture partitionnée techniquement.
- Utiliser des plug-ins pour fournir des extensions supplémentaires ou des fonctionnalités additionnelles correspond plutôt à une architecture partitionnée par domaine.

Quand envisager le style d'architecture microkernel ?

- Le style d'architecture microkernel est intéressant à envisager comme point de départ pour une application orientée produit ou une application personnalisée qui prévoit des extensions futures.
- En particulier, c'est un bon choix pour les produits où :
  - de nouvelles fonctionnalités seront ajoutées progressivement au fil du temps,
  - ou lorsque l'on souhaite contrôler quelles fonctionnalités sont disponibles pour quels utilisateurs.

- L’architecture microkernel est également un bon choix pour les applications ou produits qui doivent exister dans plusieurs configurations, selon l’environnement client ou le modèle de déploiement.
- Les modules plug-ins peuvent définir des configurations et des fonctionnalités spécifiques à chaque environnement particulier.
- Exemple concret :
- Une application pouvant être déployée sur n’importe quel cloud (cloud AWS Amazon Web Service, Microsoft Azure) pourrait avoir différents plug-ins servant d’adaptateurs pour s’intégrer aux services spécifiques de chaque fournisseur de cloud.
- Le système central (core system), lui, contient la fonctionnalité principale et reste complètement indépendant de l’environnement cloud.
- Comme pour le style d’architecture en couches, le style microkernel est relativement simple et économique, et constitue un bon choix si vous avez des contraintes de budget et de temps.

Quand ne pas envisager le style d'architecture microkernel ?

- Toutes les requêtes doivent passer par le système central (core system), que les plug-ins soient distants ou appelés point à point.
- En conséquence :
  - Le système central devient le principal goulot d'étranglement de cette architecture.
  - Ce style n'est pas adapté aux systèmes très scalables et élastiques.
  - De même, la tolérance aux pannes globale est limitée, car toute dépendance au noyau comme point d'entrée crée un risque unique de défaillance.

- L’un des objectifs de l’architecture microkernel est de réduire les modifications dans le système central et de déléguer les fonctionnalités étendues et le code volatile aux modules plug-ins, qui sont plus autonomes et plus faciles à tester et à modifier.
- Par conséquent, si vous constatez que la majorité des changements se produisent dans le noyau et que vous n’utilisez pas les plug-ins pour contenir des fonctionnalités supplémentaires, alors ce style d’architecture n’est probablement pas adapté au problème que vous essayez de résoudre.

Characteristic	Star rating
Partitioning type	Technical or domain
Overall cost	$
Agility	★★★
Simplicity	★★★★
Scalability	★
Fault tolerance	★
Performance	★★★
Extensibility	★★★

Architecture en couches

Characteristic	Star rating
Partitioning type	Technical
Overall cost	$
Agility	★
Simplicity	★★★★★
Scalability	★
Fault tolerance	★
Performance	★★★
Extensibility	★