

# **Ingénierie du Logiciel**

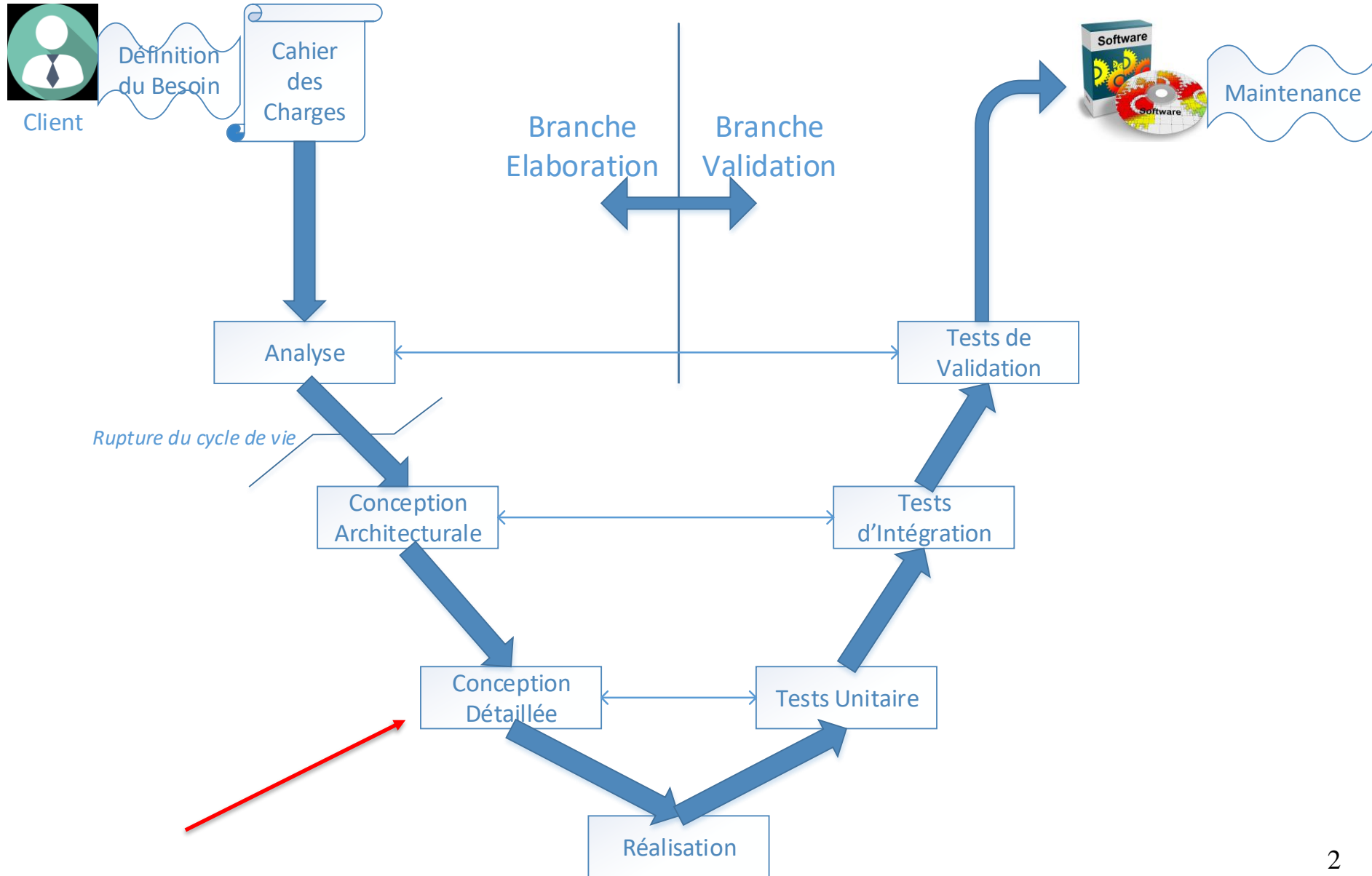
## **Master 1 Informatique – 4I502**

### **Cours 6 :**

### **Conception Détaillée**

**Yann Thierry-Mieg**  
**[Yann.Thierry-Mieg@lip6.fr](mailto:Yann.Thierry-Mieg@lip6.fr)**

# Rappel



# Conception Architecturale vs Détaillée

---

## C.A : Inter-Composants

- Les composants sont des boîtes noires
  - Responsabilités de données + traitements
    - Issus des classes métier découpées + opérations du système
  - Connectés les uns aux autres via des interfaces requises/offertes
  - Séquences => un lifeline par instance de composant

## C.D : Intra-composant

- Réalisation précise d'un composant
  - Comment stocker les données et implanter les traitements
  - Acteurs = les autres composants du système
  - Niveau classes, préparer la realization

Relation de *raffinement* entre ces descriptions.

Cohérence !

# Conception Détaillée : Objectifs

---

- Préparer la réalisation
  - Implémentabilité sur une plateforme fixée
- Choix de conception fins
  - Langage ou technologie support
  - Structure de données, index/map, ...
  - Design Patterns, schéma de BD,...

---

# Plateformes à Composants

---

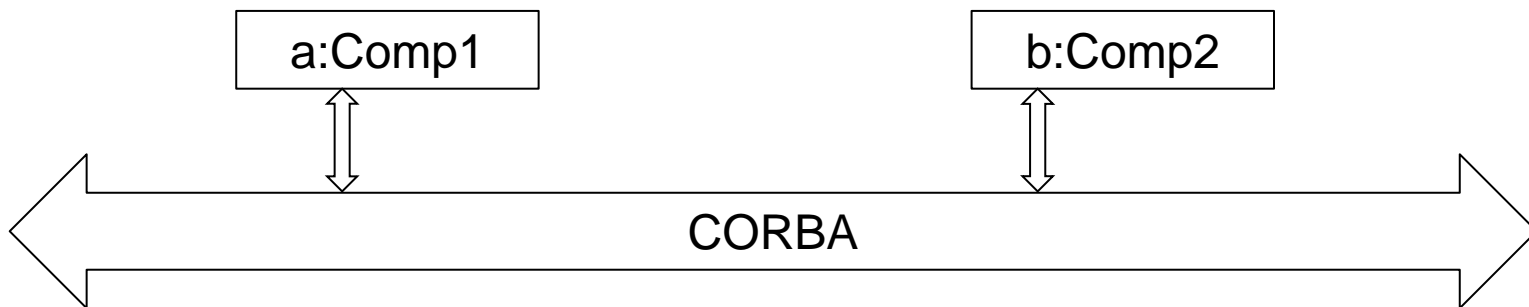
# CORBA ('92)

Common Object Request Broker Architecture (Standard OMG)

Notion de Bus Logiciel

Gestion de défis technologiques de façon générique

- Langages hétérogènes
- Réseau
- Architectures matérielles hétérogènes



a souhaite invoquer : bool foo (String s) sur b

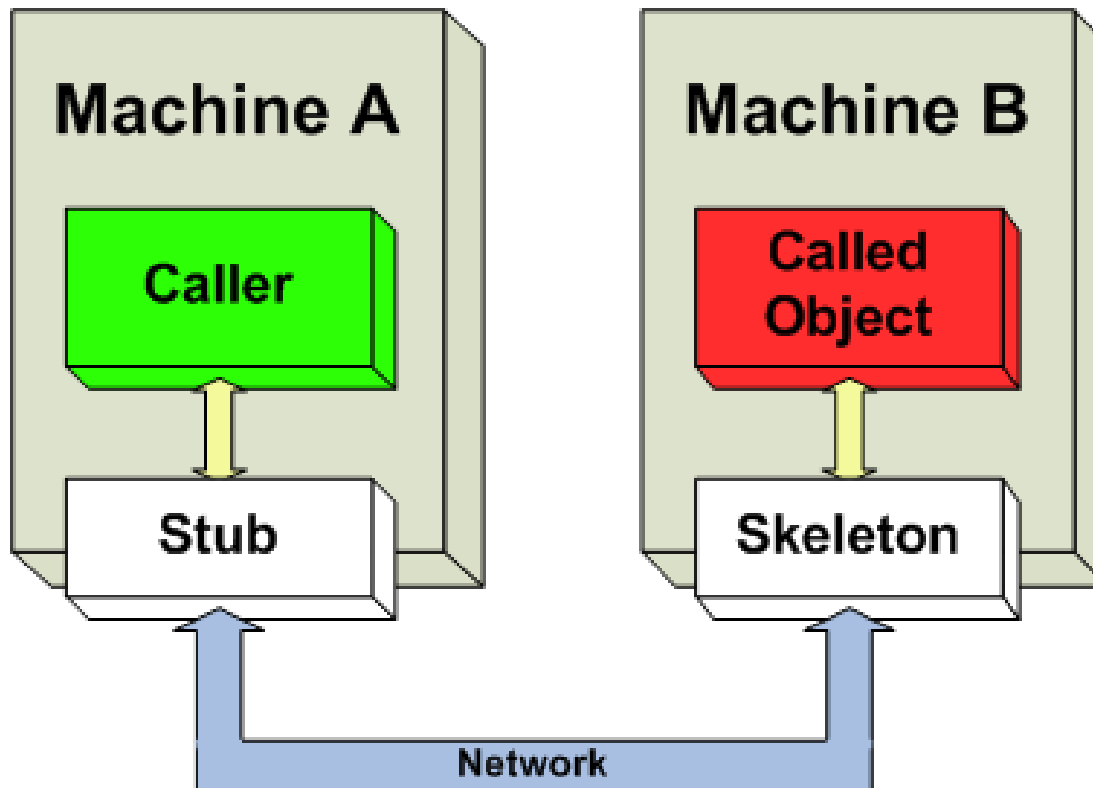
# CORBA

- Une API de communication indépendante du langage
  - Déclaration des services dans un langage neutre :
    - IDL (Interface Definition Language)
  - ✓ Module (+- un namespace) + Interface(s) portant de opérations
    - Signatures uniquement
    - Cf Connecteurs Medvidovic ADL

```
module HelloApp
{
  interface Hello
  {
    string sayHello();
    oneway void shutdown();
  };
};
```

# Compilation de l'IDL

- Construire pour une cible particulière
  - Stub : un *proxy* pour le client, représente localement l'objet distant
  - Skeleton : un élément côté serveur qui permet de recevoir les requêtes
  - Le même IDL peut générer du Java, du C, de l'ADA, du Fortran ...





# DP Proxy

---

Proxy : objet qui fait semblant d'être un autre objet

Par exemple le proxy réseau : de votre browser se comporte comme un gateway, internet (e.g. comme une box) mais rajoute des traitements (filtres, cache...)

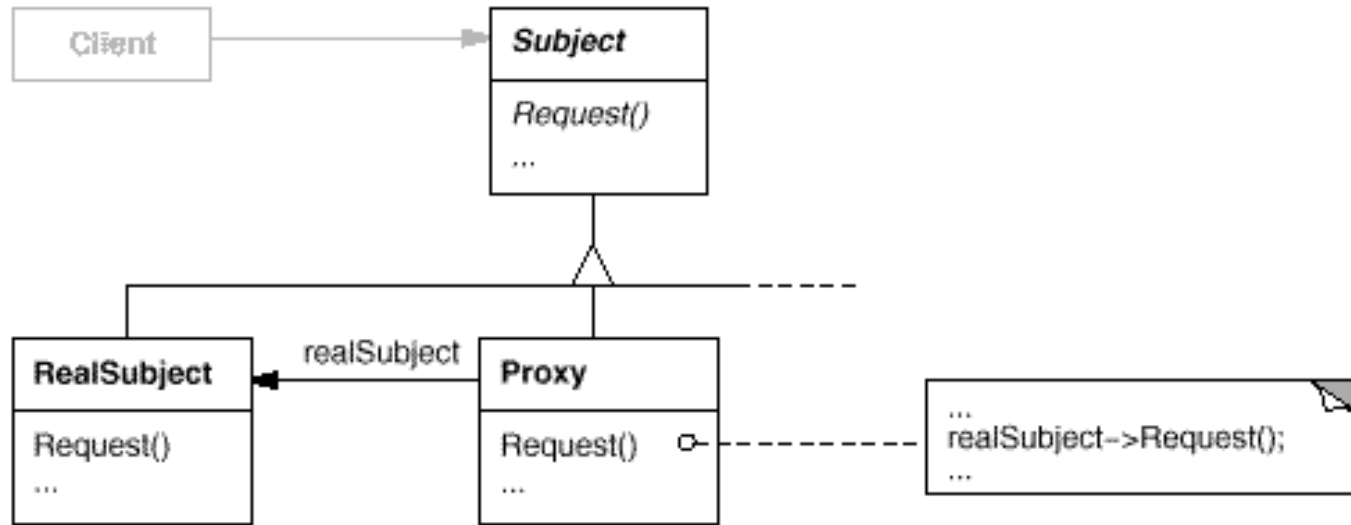
Pour le DP, le proxy est une classe qui implémente les mêmes opérations que l'objet qu'elle protège/contrôle.

Plusieurs variantes de Proxy, selon la finalité :

- ✓ Proxy Virtuel : retarder les allocations/calculs couteux
- ✓ Proxy de Sécurité : filtre/contrôle les accès à un objet
- ✓ Proxy Distant : objet local qui se comporte comme l'objet distant et cache le réseau
- ✓ Smart Reference : proxy qui compte les références (C, C++)

# DP Proxy : Structure

GOF, GHVJ'95



Subject : interface manipulée par le client

RealSubject : un objet lourd à instancier

Proxy : retarde la création du sujet réel

Délégation particulière, où délégat (Proxy) et délégué (RealSubject) réalisent la même interface

# DP Proxy Distant : Principes

---

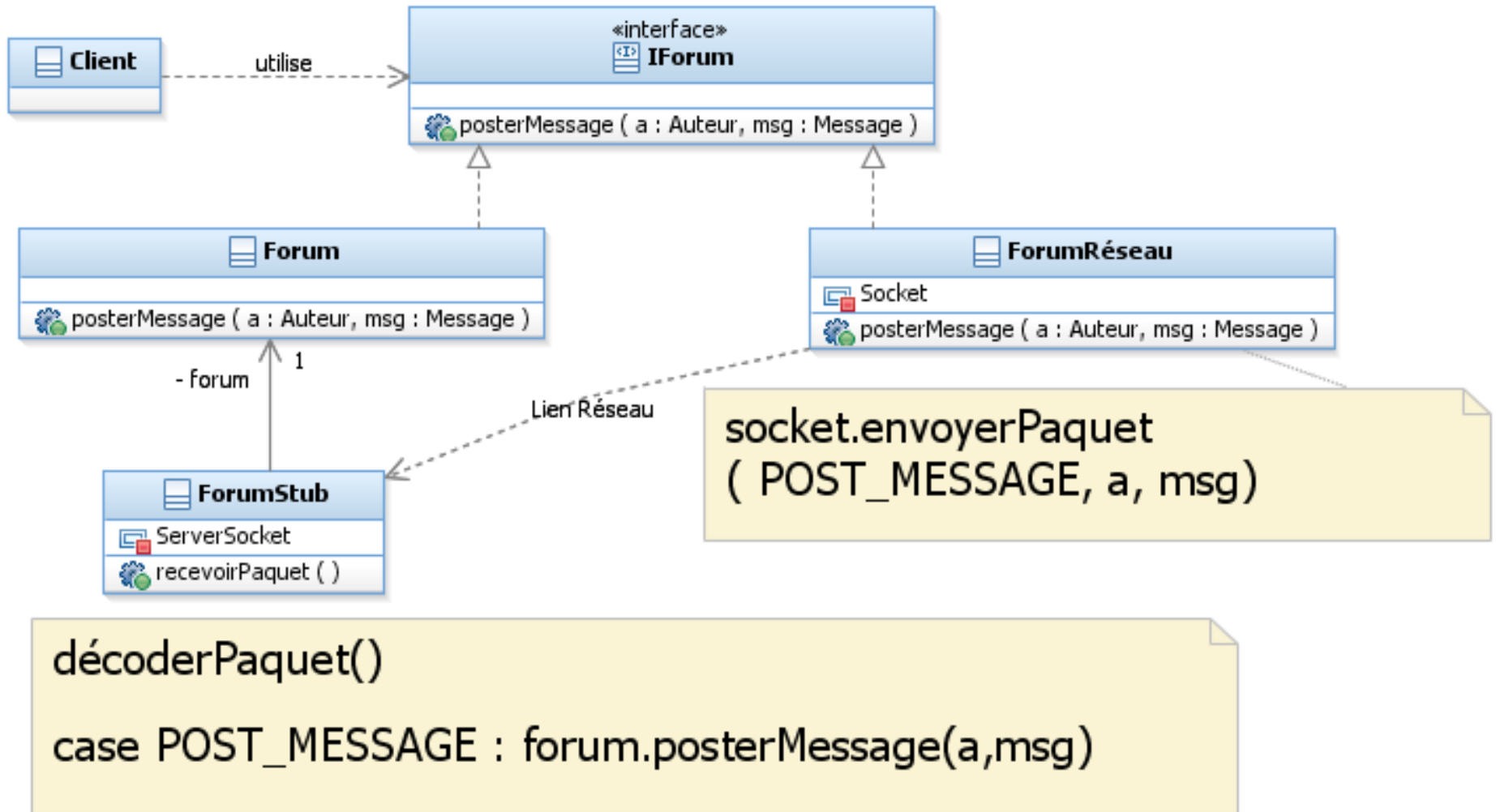
On a une application répartie sur plusieurs machines

On voudrait développer l'application sans trop se soucier de où sont physiquement stockés les objets

Proxy réseau : objet local à la machine, qui se comporte comme l'objet distant, mais répercute ses opérations sur l'objet distant via le réseau

- ✓ Comportement par délégation, mais avec le réseau interposé

# DP Proxy distant



## Proxy distant: conclusions

---

Généralise la notion de RPC (remote procedure call)

Rends transparent la localisation de objets

La réalisation du Proxy réseau et du stub suit une ligne standard

De nombreux frameworks offrent de générer cette glu automatiquement (et/ou de la cacher)

- ✓ Java RMI : remote method invocation
- ✓ CORBA...

# CORBA : Services

---

## Services offert par le bus logiciel

- Gestion des communications
  - Passage par copie des arguments, marshall/unmarshall = sérialisation
  - Gestion du réseau
  - Service de nommage ou adressage
    - Obtenir des références à des objets distants
  - ✓ Création, Destruction, configuration des composants
    - Cycle de vie des composants
  - ✓ Autres services
    - Transactions, Sécurité, Singleton, ...

CORBA est un *middleware*

- fournit une abstraction des plateformes techniques

# Plateformes Orientées Composant Modernes

# Microsoft

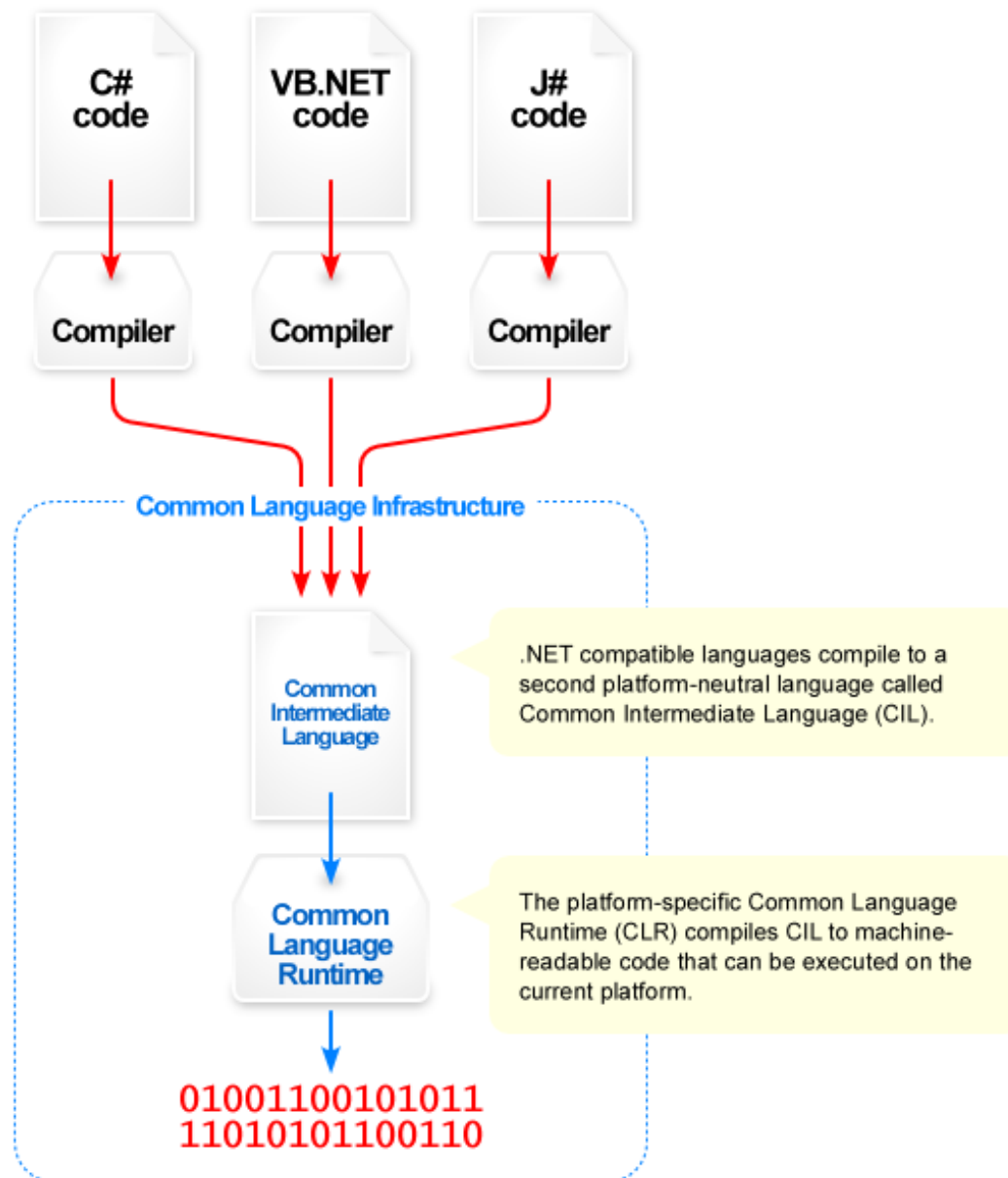
---

- Trois générations de plateformes composant
  - COM -> DCOM -> .Net (Inter-opérables)
  - Langages : C#, VB, C++, python, scheme, standard ML...
  - Une seule plateforme (Win32), plutôt vision locale à la machine
- Déclarations de :
  - Interfaces des Services : CTS : common type system
    - Agnostique au langage
  - ✓ Services : Binaires en CIL Common Interface Language
    - + ou – des DLL + des points d'entrée bien définis
  - ✓ Runtime : CRE Common Runtime Engine
    - Lie les différents composants entre eux, cycle de vie des composants...



# Microsoft (2)

wikipedia



- Plateforme orientée composant et centrée sur Java
  - Sous-jacent dans eclipse, mais applications diverses -> domotique
- Services :
  - Déclarés via des interfaces Java + configuration XML + Manifest.MF
    - Finalement neutre vis-à-vis d'une réalisation e.g. en C++
  - Notion de points d'extensions de la plateforme
    - Réaliser une certaine interface + se déclarer = plugin
- Composants / Bundle
  - Notion d'Activator
    - Install, start, stop, update
      - Gestion du cycle de vie du composant
  - Intanciation *lazy* des dépendances (résolution sur *start*), versioning

# OSGI/Eclipse

- Services offerts par la plateforme eclipse
  - Registry
    - Les services actuellement disponibles sont inspectables par reflection
    - Les nouveaux services s'enregistrent et sont démarrés quand on les consulte
      - Intérêt du XML de configuration
    - Matérialise un annuaire
  - Exemple : fournir un éditeur

```
<extension point="org.eclipse.ui.editors">
```

```
<editor
```

```
id="com.xyz.XMLEditor"
```

```
name="Fancy XYZ XML editor"
```

```
icon="./icons/XMLEditor.png"
```

```
extensions="xml"
```

```
class="com.xyz.XMLEditor"
```

```
contributorClass="com.xyz.XMLEditorContributor"
```

```
symbolicFontName="org.eclipse.jface.textfont"
```

```
default="false">
```

```
</editor>
```

```
</extension>
```

+ réalisation de l'interface Java  
Colorisation, interception des  
commandes, complétion,  
erreurs...

# J2EE

---

- Couche RMI : Remote Method Invocation
  - Réalisation d'un proxy distant général
    - Les arguments des opérations doivent être Serializable
  - ✓ Pour une interface Java annotée
    - Génération des Stub/skeleton + marshall/unmarshall par copie
    - Service de nommage
- J2EE Enterprise Edition
  - Basé sur RMI
  - Trois couches pour un composant (Bean)
    - Home (Logique applicative), Session (données locales), Entity (données stockées)
    - Déploiement dans divers contexte, e.g. serveur web ou d'applications
  - ✓ Pur Java, solution déclinante (lourdeur, interopérabilité)

- Couche transport : Protobuf
  - Définition neutre du format des données
    - Génération pour divers langages des stub/skeleton
    - GO, C++, Java, Python, Ruby, ...
  - Protocole de sérialisation efficace
    - Aussi utilisé pour stocker les données persistées
    - Gestion de versions des données
- Couche Services : GRPC
  - Déclaration de services via des signatures
    - Homogène avec la description protobuf + transport http
  - ✓ Très populaire, eg. Netflix, la plupart des services Google, cisco, ...solutions d'hébergement dans le cloud

# Exemple Hello World gRPC

---

// The greeting service definition.

```
service Greeter {  
    // Sends a greeting  
    rpc SayHello (HelloRequest) returns (HelloReply) {}  
}
```

// The request message containing the user's name.

```
message HelloRequest {  
    string name = 1;  
}
```

// The response message containing the greetings

```
message HelloReply {  
    string message = 1;  
}
```

# Composants -> Service

---

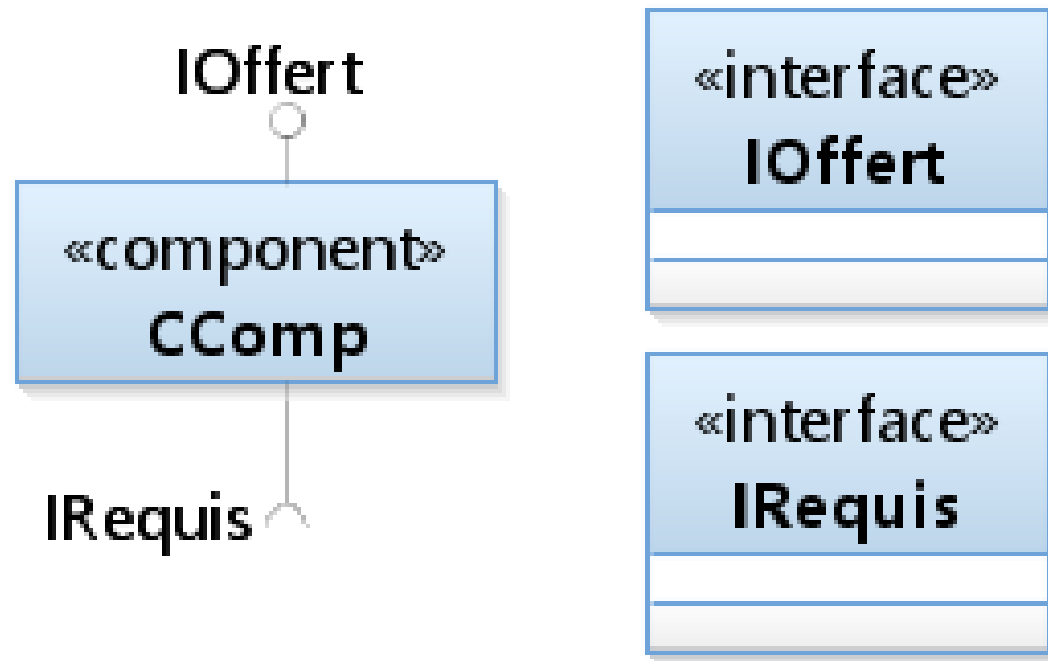
- L'orienté composant amène la notion de service
  - Service oriented-architecture SOA, Software-as-a-Service SaaS
  - Exposer du logiciel sous la forme de services dématérialisés
- Interfaces neutres pour les webservices
  - Certains standards comme WSDL, plus ou moins suivis
- Protocoles d'échanges assez bien balisés
  - http + SOAP + XML ou JSON
  - Requêtes http nature POST ou GET
  - Facilité d'interaction quelque soit le langage du client (shell ok)

# Une conception détaillée en J2SE des composants



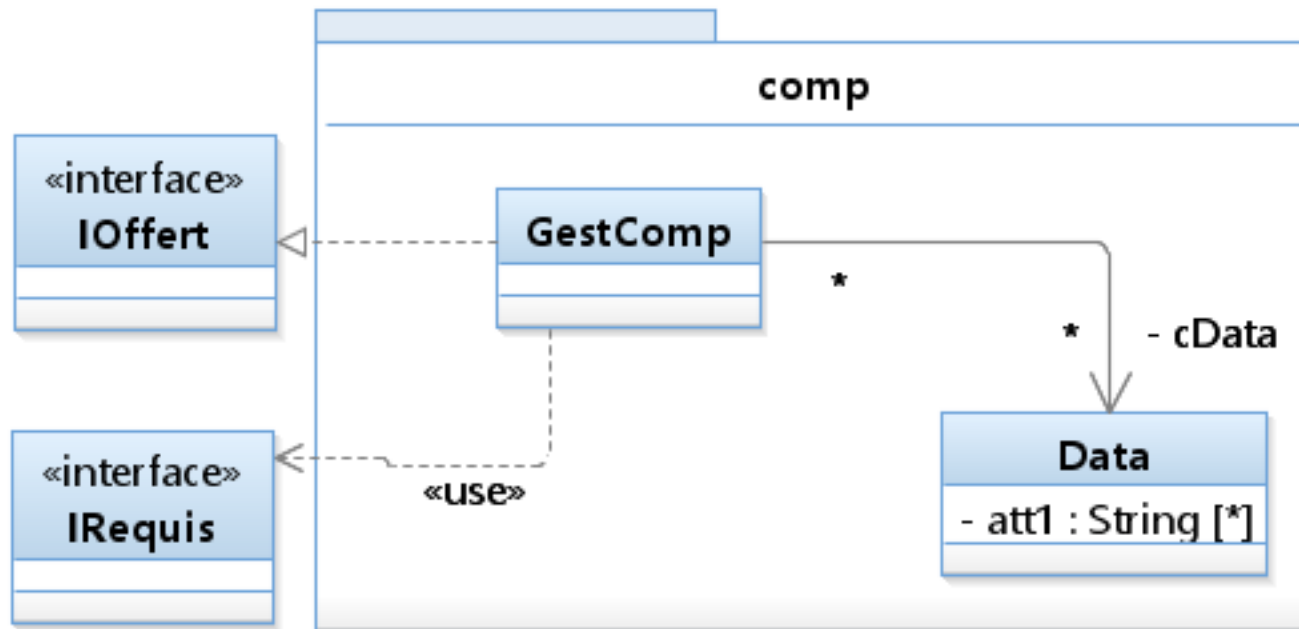
# Des composants Java

- Objectifs :
  - matérialiser le discours, proposer une réalisation avec des outils connus
  - **Cohérence** entre la conception architecturale et détaillée
- Scope :
  - Intra-composant



# Structure du Composant

- Cohérence :
  - Interfaces offertes :
    - implémentées par une classe du composant
  - Interfaces requises :
    - utilisées,
      - soit stockées dans des attributs
      - soit simplement invoquées, e.g. Singleton
  - /!\ multiplicité n'est pas visible sur le diagramme de composants



# Mise en place

- Préparation de la génération de code
  - => structure du modèle importante
- Un composant **CComp** => un package **comp**
  - On trouve plus généralement `api` + `api.impl`, cf aussi modules Java 9+
- Un package pour loger les interfaces
  - Simpliste, principalement *ne pas placer les interfaces dans les package qui contiennent des implantations*
- Ne référer *que* aux interfaces quand on veut sortir du package/composant courant
  - Pas de dépendances structurelles entre package
  - Que des types simples ou des interfaces dans les signatures d'interfaces
  - Respect/Cohérence avec le diagramme de composants

# Une façade pour le composant

---

- Une classe qui réalise l'interface offerte principale du composant
  - Composant **CComp** => classe **GestComp**
  - Une seule instance par instance de composant
  - Point d'entrée du composant
- Accède essentiellement à toutes les informations internes au composant
  - Cf la vision qu'a le système en analyse sur les données métier
  - Mais à réaliser maintenant :
    - Associations \*
    - Map<clé, objet> pour représenter des index...
- Par défaut penser que GestComp réalise les interfaces offertes et utilise ou stocke les interfaces requises.

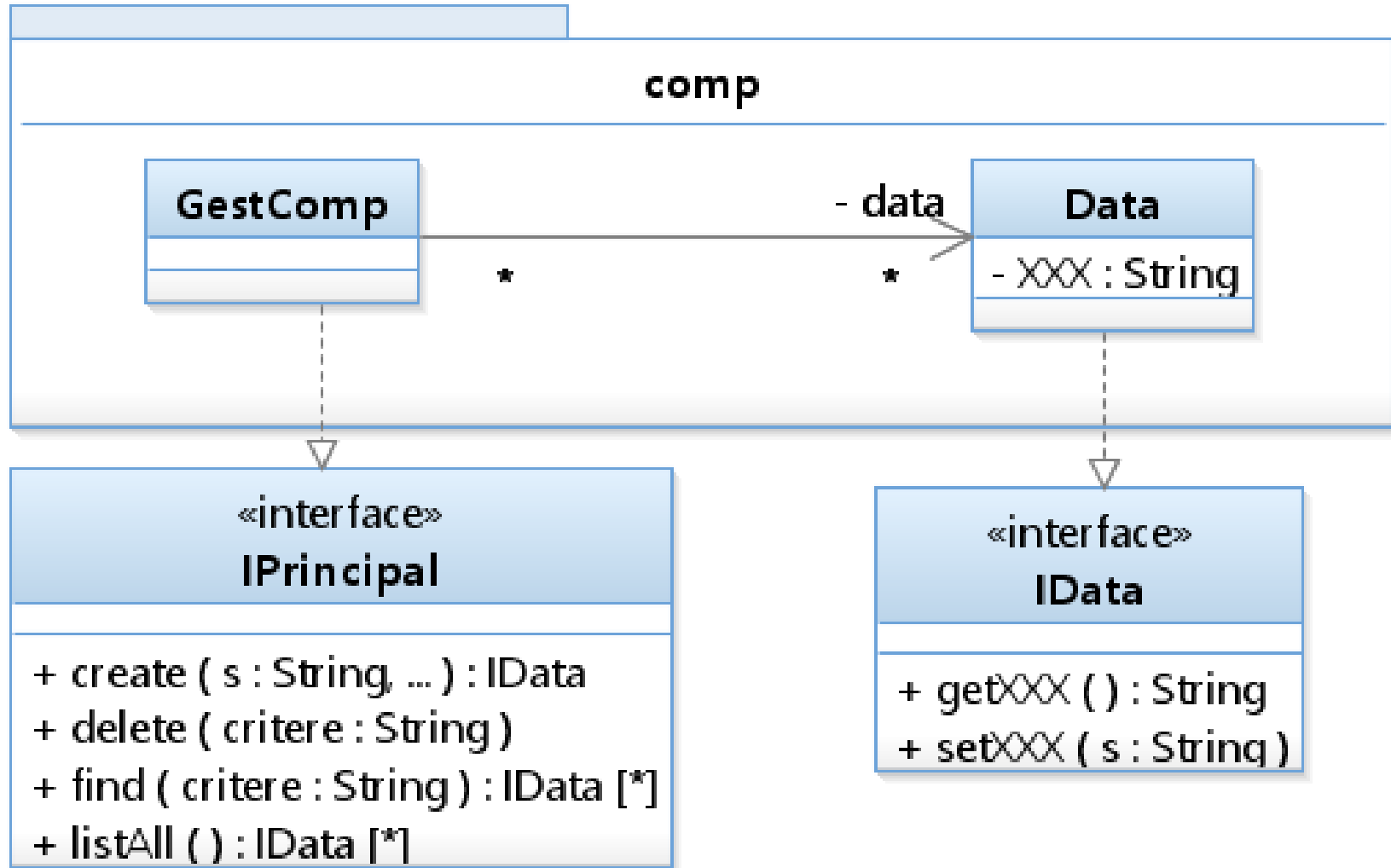
# Composant de stockage

---

- Découpage des classes métier amène naturellement des composants de stockage
  - Responsabilité = gérer l'ensemble des exemplaires, leur état, leur statut...
  - Effet => GestComp connaît \* occurrences de Data
- API CRUD : Create, Replace, Update, Delete
  - Deux options principales :
    - API par sous interfaces, orienté objet mais respectant les contraintes du composant
    - API orienté composant pur, utilisant beaucoup d'identifiants

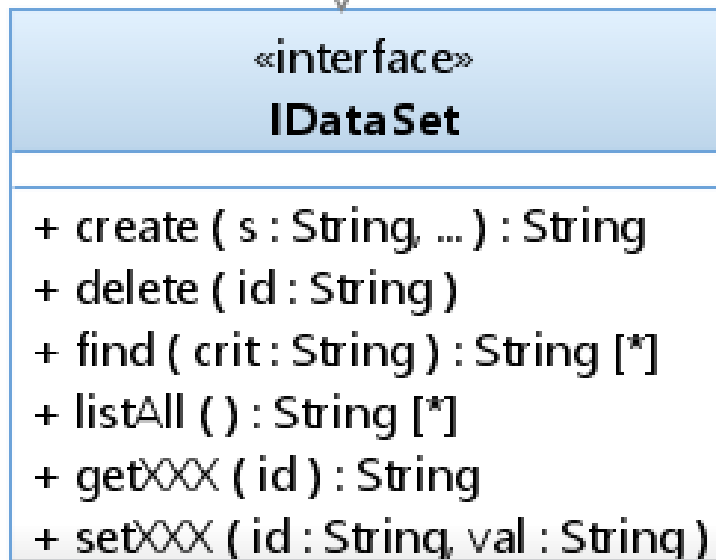
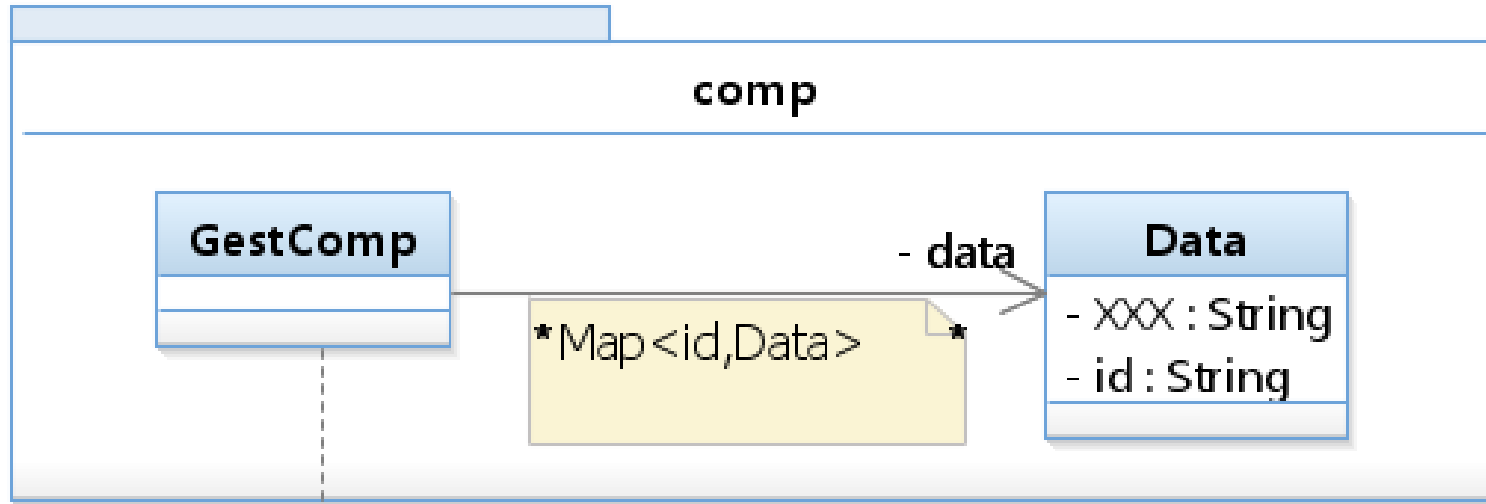
# Version OO

CComp offre deux interfaces : IPrincipal et IData



# Version Composant « pur » par ID

On évite d'exposer des interfaces nouvellement créées / internes



- Limites des « types simples dans les signatures »
- On peut accepter des types POD (pures struct) de façon à obtenir plusieurs attributs en un seul get
- On peut penser aussi à une implémentation sur SGBDR

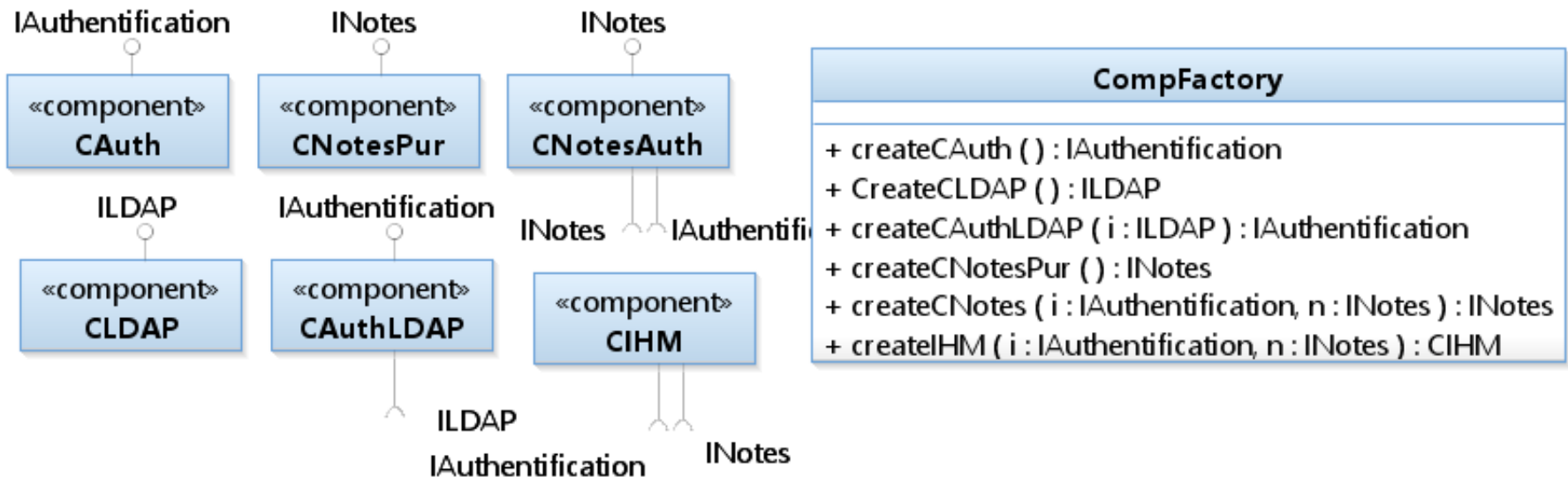
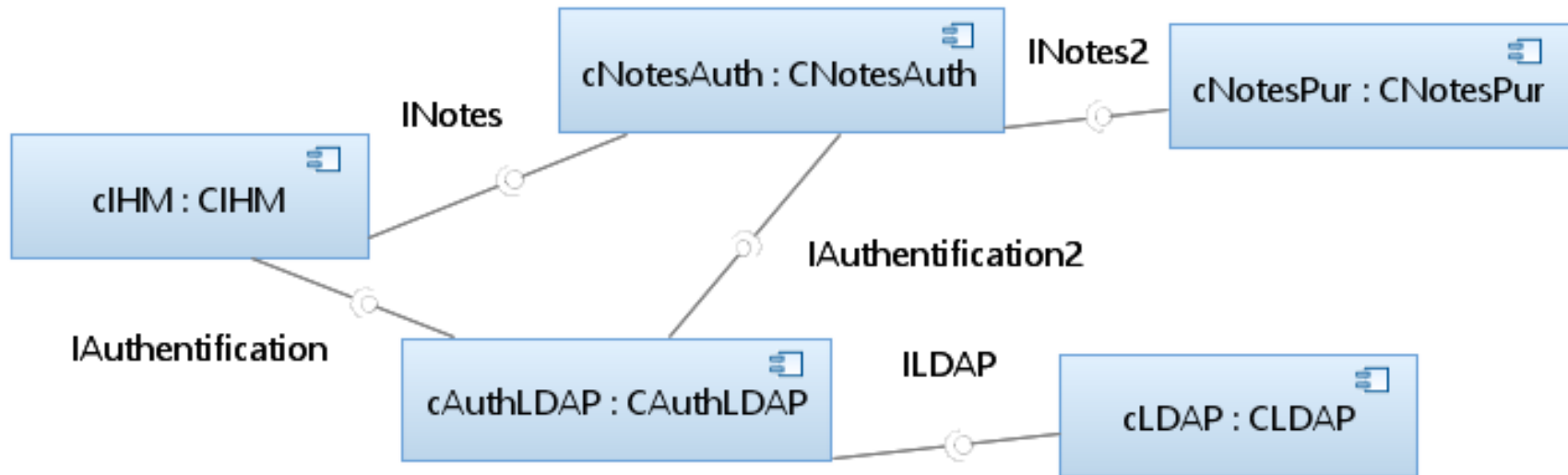
# Instanciation et Configuration

Comment instancier ces classes en respectant le modèle de composants ?

- Application d'un DP Factory
  - Une classe chargée de construire des abstractions sans exposer les implémentations
  - `createCarre(x,y,a):IForme`
- Choix basique dans l'UE :
  - une seule classe de factory static isolée dans son package
  - Grosses dépendances sur tout le reste OK
- Opérations de la factory :
  - Création de composants, rendus via un typage sur interface offerte
  - Les dépendances des composants sont passés si possible à la construction
    - Sous la forme d'interfaces !
  - ✓ Si possible on ordonne les créations => DAG de construction
  - ✓ S'il existe des cycles
    - => il existe des setter, provider, singleton pour fermer le cycle après instanciation



# Example



# Dependency Injection

---

- Au-delà de Factory
  - Pattern général : injection de dépendance
  - Le client déclare ses dépendances, un mécanisme externe se charge de les lui fournir
- Exemple Google Guice
  - Intégration dans Java d'un framework de DI
  - Généralisation des stratégies :
    - `@Inject IRequis req;`
  - ✓ Définition séparée de « modules »
    - Ensemble de bindings entre une interface et une réalisation concrète
    - Externe au code qu'on configure
  - ✓ Facilité de mise en place des tests d'intégration par exemple
  - ✓ Reconfigurations aisée, frameworks extensibles e.g. XText...

# Guice example

```
public class BillingModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(TransactionLog.class).to(DatabaseTransactionLog.class);
        bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
        bind(BillingService.class).to(RealBillingService.class);
    }
}
```

```
public class RealBillingService implements BillingService {
    private final CreditCardProcessor processor;
    private final TransactionLog transactionLog;

    @Inject
    public RealBillingService(CreditCardProcessor processor,
        TransactionLog transactionLog) {
        this.processor = processor;
        this.transactionLog = transactionLog;
    }
}
```

# Conception Détaillée : Bilan

# Une conception « fine »

---

- Objectifs :
  - Lever les points d'ombre sur la façon de réaliser les traitements,
  - organiser le code,
  - choisir des structures de données,
  - introduire des DP...
- Choix techniques guidés par les API
  - E.g. index/map si on fournit une interface par ID
- Mais savoir s'arrêter !
  - Quand c'est clair, coder, puis resynchroniser les diagrammes sur le code (round-trip engineering)
- Un choix possible abordé dans l'UE mais nombreuses façons de réaliser des composants
  - Dépendances fonctionnelles *seulement* entre parties du système