Master II Informatique
Parcours : Génie logiciel

SILOC : Spécification et Implémentation des Langages à Objets et à Composants - Spécification et transformation de contraintes sur les architectures à base de composants -

Chouki.TIBERMACINE@lirmm.fr
Maître de conférences en informatique

http://www.lirmm.fr/~tibermacin/ens/siloc/

Plan du cours

- **Partie 1 :** Généralités sur les composants et architectures logicielles à base de composants
- **Partie 2 :** Spécification des contraintes architecturales
- **Partie 3 :** Un modèle de composants pour les contraintes architecturales
- **Partie 4 :** Transformation des contraintes architecturales
- **Partie 5 :** Génération de code à objets à partir de contraintes architecturales
- **Partie 6 :** Génération de composants à partir des contraintes

Plan du cours

- **Partie 1 :** Généralités sur les composants et architectures logicielles à base de composants
- **Partie 2 :** Spécification des contraintes architecturales
- **Partie 3 :** Un modèle de composants pour les contraintes architecturales
- **Partie 4 :** Transformation des contraintes architecturales
- **Partie 5 :** Génération de code à objets à partir de contraintes architecturales
- **Partie 6 :** Génération de composants à partir des contraintes

Plan de cette partie

- Rappels sur le développement par objets et ses limites
- Modèles pour le développement par composants
- Quelques définitions autour des architectures à composants

Plan de cette partie

- Rappels sur le développement par objets et ses limites
- Modèles pour le développement par composants
- Quelques définitions autour des architectures à composants

Développement par objets : avantages

- Un objet : encapsulation de données + fonctions associées
- Les données encapsulées peuvent être des références vers d'autres objets : notion de composition
- Un objet peut être passé en argument : paramétrage plus riche (que le passage en argument de données ou de fonctions séparées)

Développement par objets : avantages -suite-

- Les descripteurs des objets (classes) peuvent être réutilisés et spécialisés : notion d'héritage

- L'héritage apporte un tas d'avantages :
  - Réutilisation de code
  - Construction de hiérarchies de concepts
  - Sous-typage => polymorphisme

Limites du développement par objets

- Problème de couplage fort
- Entrelacement des aspects fonctionnels et non-fonctionnels
- Entrelacement de la logique métier et l'architecture

Couplage fort

- Dans beaucoup d'applications à objets, à l'intérieur d'une classe donnée, on instancie un (ou plusieurs) objet(s) d'une autre (d'autres) classe(s)

```
class A{
        B b = new B();
        public ma(){ b.mb(); }
}
class B{
        public mb(){ ... }
}
```

La classe A dépend alors directement d'une autre classe B

- Si on décide d'instancier une autre classe (C au lieu de B) il faut modifier le code de la première classe (A), recompiler, ...

Couplage fort – les solutions

1. Le patron de conception Factory :
   - Une classe fabrique (usine) d'objets
   - Une classe avec des méthodes statiques qui ont pour seul rôle de créer et retourner des objets
   - Les objets retournés peuvent avoir un type abstrait commun : type de retour de ces méthodes
   - Les classes « clientes » dépendent : 1) de ces fabriques et non des classes qu'elles instancient, et 2) des types abstraits communs aux objets « usinés » (et non d'implémentations)
2. L'injection de dépendances :
   - Constructeurs paramétrables (paramètres = objets cibles)
   - Mutateurs (setters) dans les objets « dépendants »

- Java EE et Spring fournissent des mécanismes d'injection automatique de dépendances

Entrelacement des aspects fonctionnels et non-fonctionnels

- Dans le code des applications à objets classiques, on retrouve un mélange entre code implémentant la logique métier et code se chargeant des aspects non-fonctionnels : distribution, gestion des transactions, de la sécurité, ...
- La plate-forme Java EE fournit des mécanismes pour séparer ces aspects
- Elle propose par ailleurs de gérer certains aspects non-fonctionnels avec des comportements par défaut

Entrelacement de la logique métier et l'architecture

- Dans le code des applications à objets, la description d'architecture n'est pas explicite
- Elle est mélangée avec le code de la logique métier
- Deux aspects de l'architecture sont noyés dans le code :
  - L'expression du requis
  - La description des instanciations et des mises en place des dépendances (les connexions)
- Le framework Spring permet par exemple de séparer ces aspects

Description d'architecture implicite

- Le requis implicite :
  - Dans une classe, on importe et utilise un type concret pour les attributs (~ ports requis)
- Les instanciations non-factorisées :
  - Les fameux « new » (et/ou appels de constructeurs) répartis dans tout le code, alors qu'une partie peut être groupée : architecture statique
- Les connexions codées en dur :
  - Les affectations des attributs au milieu du code
- Les solutions sont multiples, mais le développement par composants se veut comme la solution unificatrice

Plan de cette partie

- Rappels sur le développement par objets et ses limites
- Modèles pour le développement par composants
- Quelques définitions autour des architectures à composants

Composants logiciels

- **Définition 1**[1] :
  - Un composant logiciel est une unité de **composition** qui spécifie **par contractualisation** ses **interfaces**, et qui **explicite ses dépendances** de contexte
  - Un composant logiciel peut être **déployé indépendamment** et est sujet à une **composition par de tierces entités**

[1]Clemens Szyperski. Component Software : Beyond Object-Oriented Programming, Addison Wesley (2002)

Composants logiciels -suite-

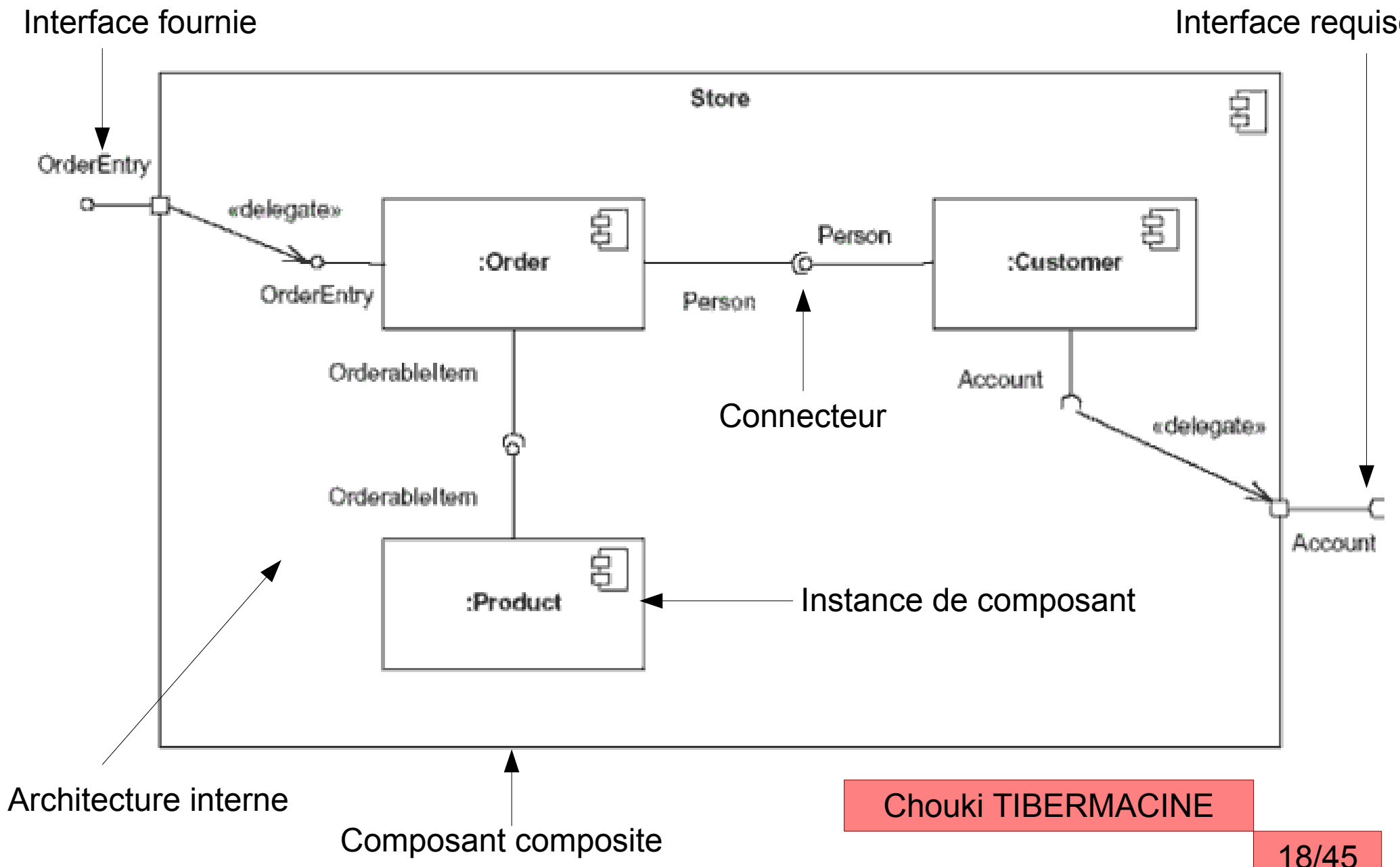
- **Définition 2**[2] :
  - Un composant est une partie **modulaire** d'un système qui **encapsule son contenu** et est **substituable**
  - Un composant définit son comportement en terme **d'interfaces fournies** et **requises**
  - Un composant sert comme un **type défini par ses interfaces** fournies et requises (incluant à la fois leur sémantique statique et dynamique)

[2]Spécification UML de l'OMG (2010)

Différences entre un composant et un objet

1. Les interfaces requises d'un composant sont explicites :
   - Interface requise : ensemble des opérations utilisées pour implémenter les interfaces fournies
   - Pour qu'un composant soit opérationnel, ses interface requises doivent être fournies (implémentées) par d'autres composants
   - Le développeur (l'architecte) définira alors un connecteur entre ces composants pour que toutes les interfaces requises soient satisfaites
2. Si le composant est composite, son architecture interne est explicite :
   - Le composant décrit de façon déclarative quelles instances le composent
   - Le composant décrit les connecteurs entre ces instances

Représentation graphique des composants (en UML 2)



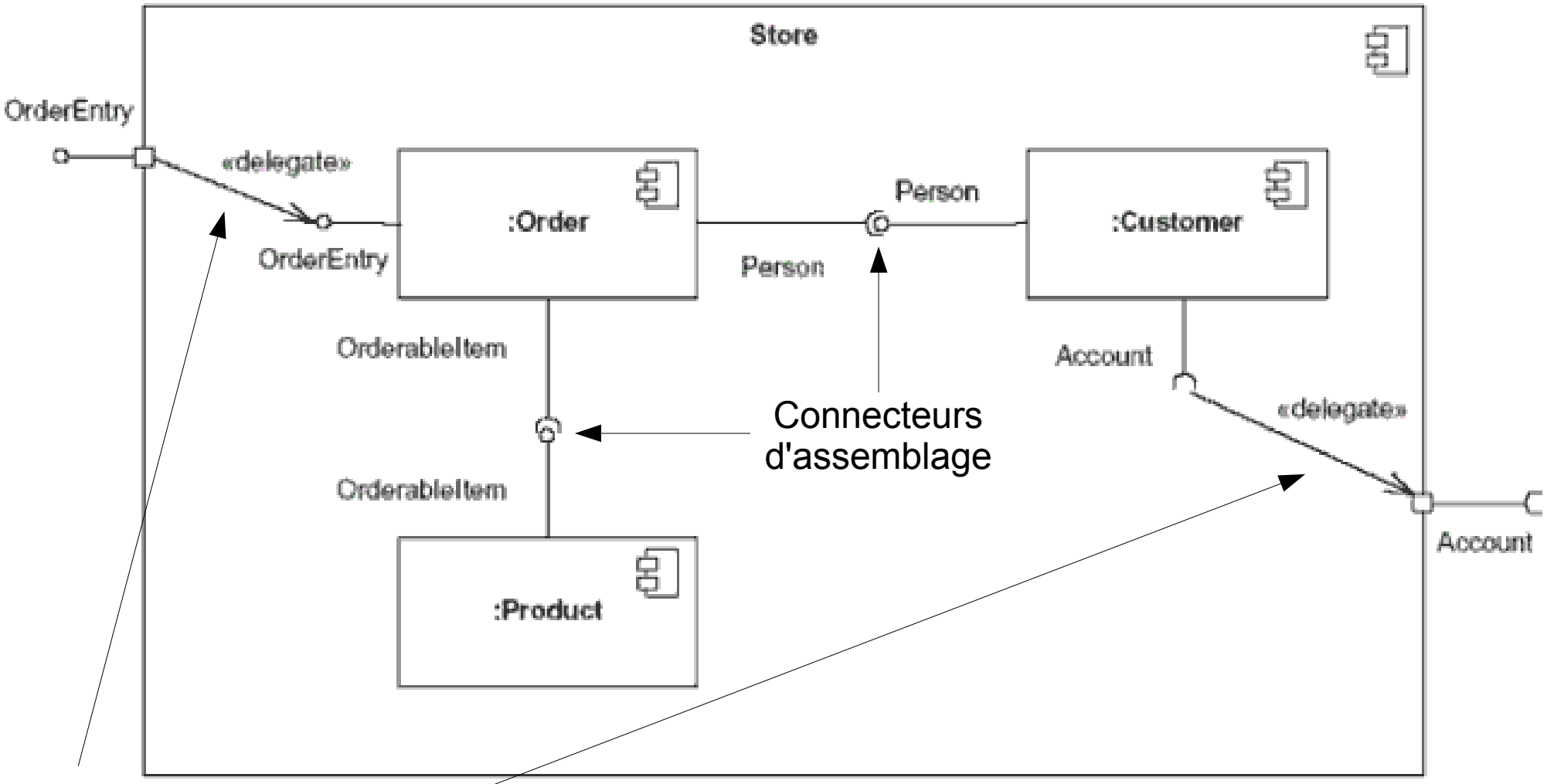
Descripteurs vs instances de composants

- Dichotomie comme dans les langage à objets :
  - Classe vs Objet
  - Descripteur de composant vs Instance de composant (ou composant)
- Descripteur de composant :
  - Déclare les interfaces fournies et éventuellement requises
  - Définit les implémentations des opérations de ses interfaces fournies
  - Si le composant est composite, déclare une architecture interne
  - Exemple : Descripteur de composant « Store »
- Instance de composant :
  - Création à partir d'un descripteur dans une architecture interne
  - Exemple : Composant « :Customer »

Composants composites et connecteurs de délégation

- Si un composant est composite, son architecture interne est constituée de :
  - Instances de composants
  - Connecteurs entre composants :
    - Connecteurs simples (d'assemblage) : relient une interface requise à une interface fournie
    - Connecteurs hiérarchiques (de délégation) : relient des interfaces de même genre (fournie ou requise) de deux composants de niveaux hiérarchiques différents

Connecteur d'assemblage et connecteurs de délégation

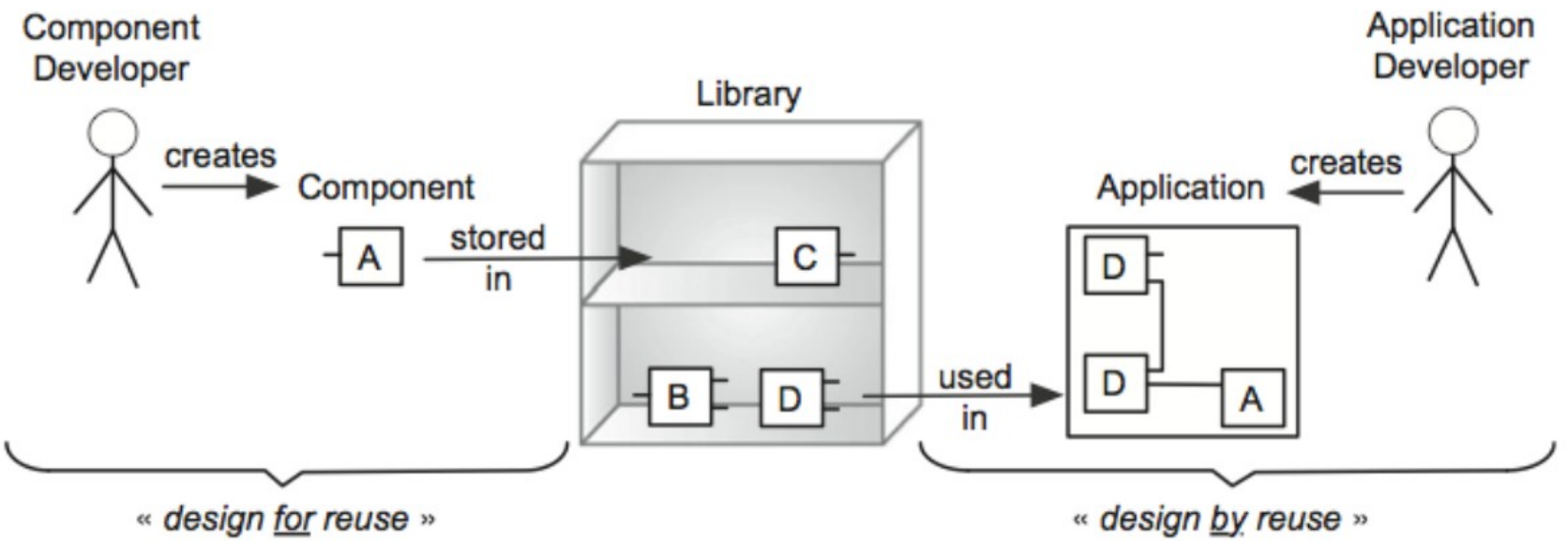


Notion de port

- Un composant encapsule son contenu : de l'extérieur, on ne voit que ses interfaces fournies et requises
- Un port représente le point d'interaction avec un composant
- Le descripteur d'un port est son interface (ou ses interfaces)
- Un port peut être : Input (décrit par une interface fournie), Output (interface requise) ou les deux à la fois (dans UML)

Développement par et pour la réutilisation

L. Fabresse et al. / Computer Languages, Systems & Structures 38 (2012) 29–43



Un petit historique sur les composants (thèse D. Hoareau)

- 1960s Components would be a good idea ...
  Mass produced software components (M.D. McIlroy 1967)
- 1970s ... 1980s : Modules, Objects ... executable et pipe (unix)
- 1991 ...Composants distribués CORBA 1.1
- 1993 COM : Microsoft Component Object Model
  Empaquetage d'un programme dans un composant doté d'interfaces. Précurseur de .NET
- 1997 JavaBeans, ActiveX :
  Construction de composés par assemblage de composants, schéma "MVC", application à l'interfaçage
- 1997 ACME (Architecture Description Interchange Language), une tentative de langage minimal pour la description échangeable de descriptions d'architectures

Un petit historique sur les composants - suite

- 1998 SOFA (Software Applicances) : fournir une plate-forme pour le développement d'applications par "composition" d'un ensemble de composants.
  ADL : Langage de description d'arch. à base de composants

- 1999 Enterprise JavaBeans (EJB), composants distribués Java

- 2002 Fractal Component Model, Corba Comp. Model (CCM)
- 2002 ArchJava - Connect components - Pass Objects

- 2006 ... Langages de programmation et de modélisation par composants ... Evolution des langages à objets vers les composants ... Nombreuses visions et recherches

Modèle de composants

- Un modèle de composants est une définition de :
  - la syntaxe des composants
  - leur sémantique
  - les règles de leur assemblage
- Classification des modèles de composants de Lau et Wang[1] :
  - 13 modèles de composants
  - Classification selon la composition (à la conception, au déploiement)
  - Classification selon les éléments de définition d'un modèle
- Exemples de modèles de composants
  - PECOS, Fractal et JavaBeans
  - EJB, .NET, CCM et Web services
  - Koala, SOFA et KobrA

[1]Kung-Kiu Lau et Zheng Wang. Software Component Models. IEEE TSE, vol. 33, num. 10 IEEE Computer Society (2007)

Modèle de composants -suite-

- Classification des modèles de composants de Crnkovic et. al.[1] :
  - Critères de classification :
    - Cycle de vie : modélisation, impl., packaging, déploiement, ...
    - Construction : spec. d'interfaces (syntaxe, comportement, …), mécanismes de connexion (délégation, ...), interactions (asynchrone, événements, ...)
    - Propriétés non-fonctionnelles : spec, gestion et composition
- C'est une classification plus riche, de 24 modèles :
  - EJB, .NET, CCM et Web services
  - Fractal, Koala, KobrA, AutoSAR, OSGi, SOFA, …
  - ...
- Article accessible sur la page Web du cours

[1]Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis et Michel R. V. Chaudron. A Classication Framework for Software Component Models. IEEE TSE, vol. 37, num. 5 IEEE Computer Society (2011)

Modèles de composants -suite-

- Implémentations possibles d'un modèle de composants :
  - Un langage de **description d'architecture** (ex : voir plus loin) :
    - Édition (graphique) d'architectures à base de composants
    - Analyse et validation de ces architectures
    - Génération de squelettes de code
  - Un langage de programmation par composants dédié (ex : ArchJava, ComponentJ, SCL, Compo, ...)
  - Un framework pour un langage de programmation existant (ex : OSGi, Spring, ...)

- Modèle de composants générique vs. Modèle de composants spécifique à un domaine particulier (Web, interfaces graphiques, ...)

Plan de cette partie

- Rappels sur le développement par objets et ses limites
- Modèles pour le développement par composants
- Quelques définitions autour des architectures à composants

Description d'une architecture logicielle

- Définition[1,2] : "*The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution*"
- Éléments d'architecture :
  - **Composants :** éléments de traitement et de stockage
  - **Connecteurs :** éléments d'interaction entre composants
  - **Configuration :** structure globale du système sous la forme d'instances de composants et de connecteurs
  - **Contraintes :** choix de conception et des stratégies d'évolution

[1]ANSI/IEEE Std 1471-2000, Recommended Practice for Arch. Desc. of SW-Intensive Sys.
[2] Autres définitions : http://www.sei.cmu.edu/architecture/definitions.html

Langages de description d'architecture

- Défintion[1] : un ADL est un langage (formel ou semi-formel) qui fournit des dispositifs pour modéliser l'architecture conceptuelle d'un système logiciel
- Classification de Medvidovic[1] : état de l'art des ADL (10 langages) selon les éléments cités dans le transparent précédent
- Exemples d'ADL :
  - xADL ou Acme (xAcme) : ADL à objectif général
  - Koala ou Darwin : Description d'arch. de systèmes distribués à reconfiguration dynamique
  - C2SADL : Description d'arch. de systèmes événementiels (GUI)
  - UML (profil UML pour les architectures de Kandé et al.[KAN00], modélisation des arch. avec UML1.5 [MED02] et UML2.0 [IVE04])

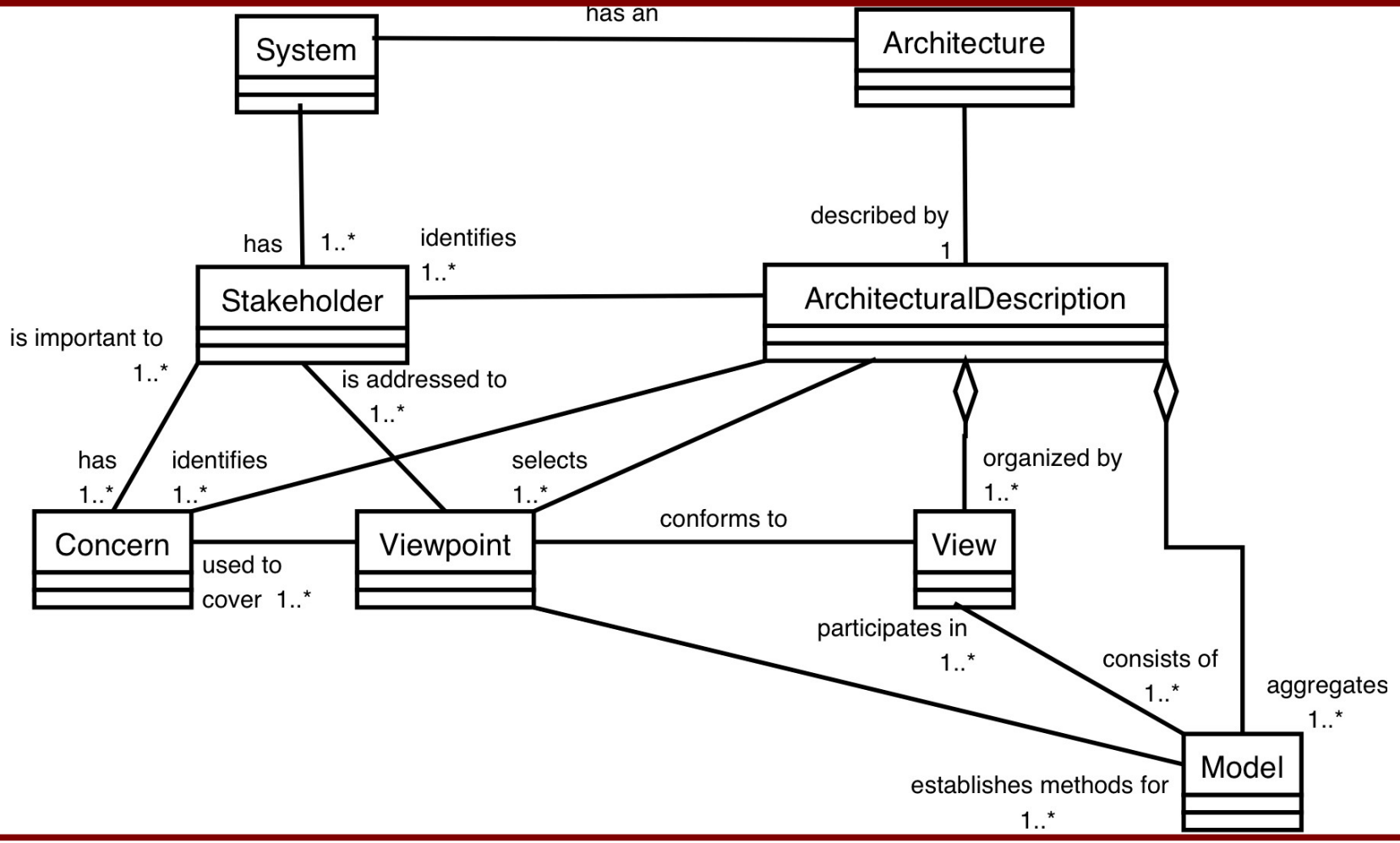
[1]N. Medvidovic et N. R. Taylor. "*A Classification and Comparison Framework for SW Architecture Description Languages*". IEEE TSE, vol. 26, num. 1 (2000) 70–93

Vues d'une architecture logicielle[1] : ancêtres des diagrammes UML

- **Vue logique :** la spécification des besoins fonctionnels d'un point de vue comportemental
- **Vue de processus :** les objets sont projetés en processus (distribution, concurrence, tolérance aux pannes, ...)
- **Vue de développement :** l'organisation des modules et des librairies dans le processus de développement
- **Vue physique :** Cette vue projette les autres éléments aux nœuds de traitement et de communication
- **Vue "plus un" :** Dans cette vue, des mappings entre les différentes vues sont définis (combinaison de vues)

[1]Philippe Kruchten. "*The 4+1 view model of architecture*". IEEE Software, Vol. 12, num. 6 (1995) pages 42–50

Vues d'une architecture logicielle (le standard IEEE)[1]



[1]ANSI/IEEE Std 1471-2000, Recommended Practice for Architecture Description of Software-Intensive Systems (2000)

Vues des architectures logicielles revisitées[1]

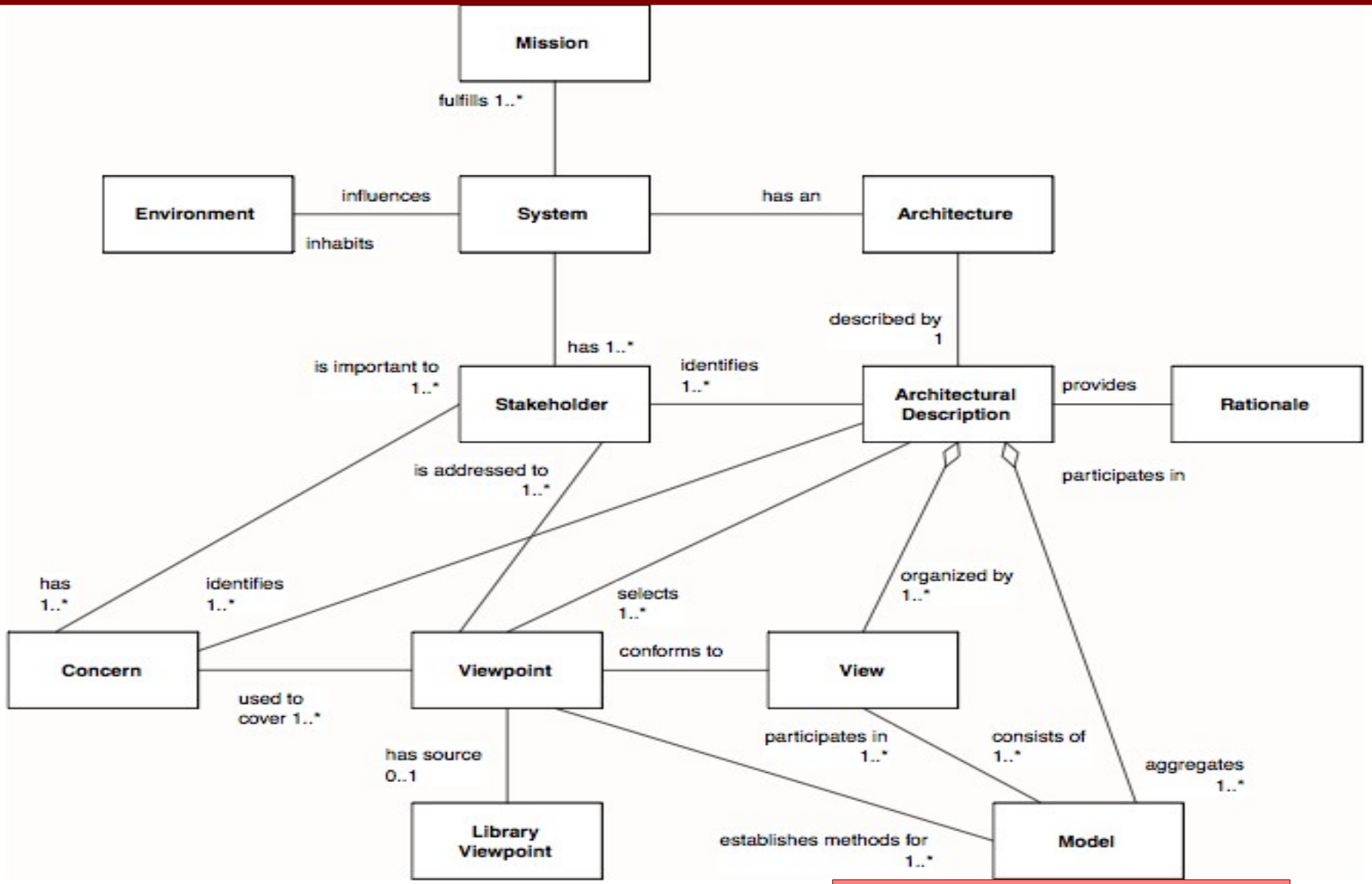
- Une nouvelle vue ajoutée aux architectures logicielles : *Decision View*

- De la formule de Perry et Wolf (1992) :
Software Architecture = {Elements , Form , Rationale}
à la formule suivante :
Architecture Knowledge = {Design Decisions , Design}

- Proposition d'un nouveau modèle pour la documentation des architectures logicielles basé sur les décisions, leurs états (approuvée, rejetée, ...), leurs alternatives, leurs liens avec les besoins, ...

[1]Philippe Kruchten et al.. “The Decision View's Role in *Software Architecture Practice*”. IEEE Software, Vol. 26, No. 2. (2009) pages 36-42

De ANSI/IEEE 1471-2000 à ISO/IEC (2007) 42010[1]



Chouki TIBERMACINE

Nouvelle organisation des vues dans ISO/IEC 42010

- Points de vues :
  - Functional/logical viewpoint
  - Code/module viewpoint
  - Development/structural viewpoint
  - Concurrency/process/runtime/thread viewpoint
  - Physical/deployment/install viewpoint
  - User action/feedback viewpoint
  - Data view/data model

- Dans la norme, pas de consensus sur le langage standard pour exprimer ces différents points de vue

- UML 2 semble être suffisamment expressif pour couvrir ces points de vue

Point de vue Code/Module

- Dans la suite du cours, nous nous focaliserons sur le point de vue « Code/Module » dans la norme ISO/IEC 42010

- Ce point de vue est plus explicitement référencé avec le nom : « Component and Connector Viewtype » dans le livre de Paul Clements

Paul Clements et al. *Documenting Software Architectures: Views and Beyond.* 2nd edition. SEI Series in Software Engineering. Addison-Wesley Professional (2010)

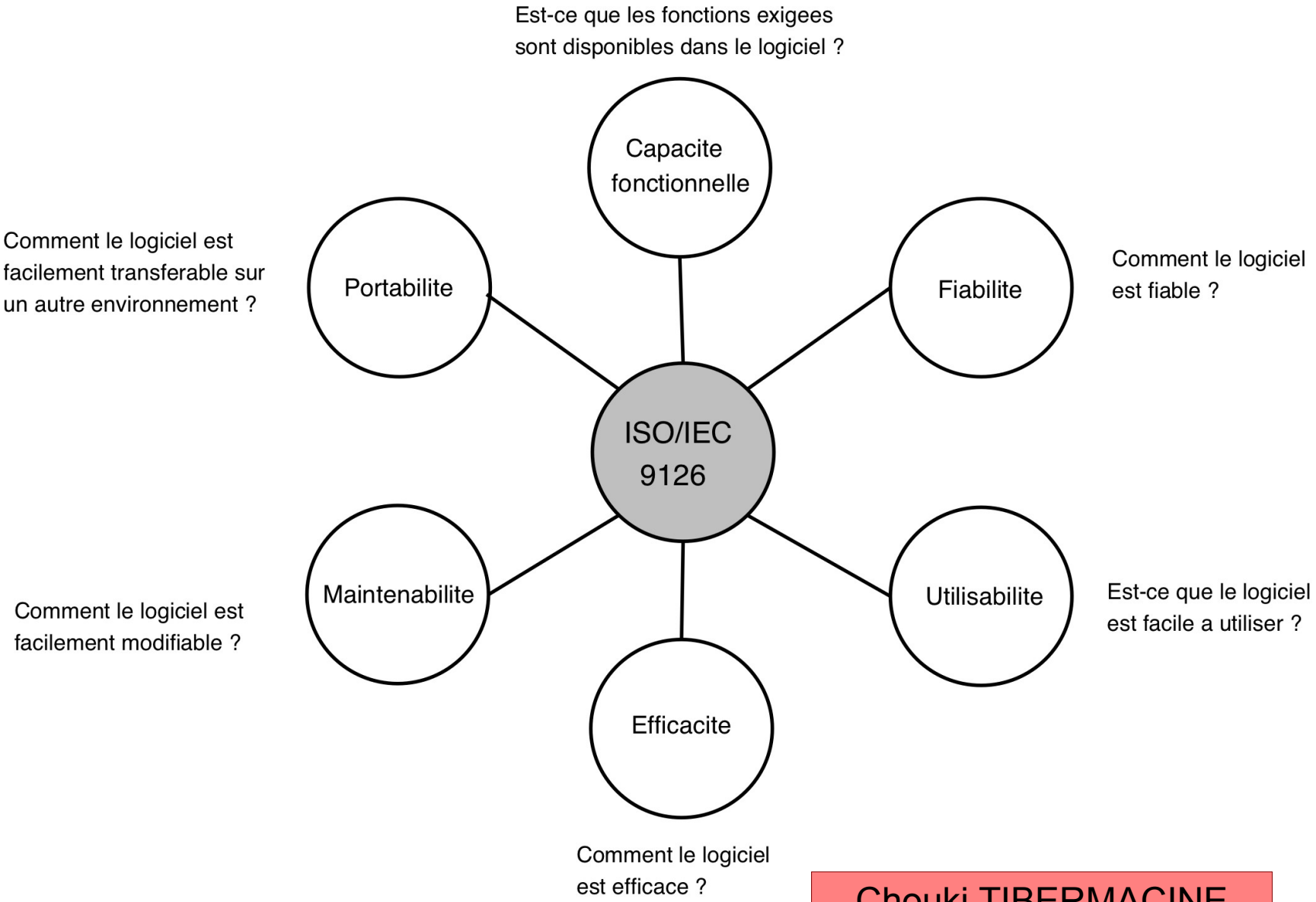
La qualité dans une architecture logicielle

- Architecture logicielle -> première réponse aux besoins de qualité
- Un attribut qualité est une propriété ciblée par un besoin non-fonctionnel spécifié dans le cahier des charges
- Dans l'ingénierie des exigences, on a identifié 3 sortes de besoins :
  - Besoins fonctionnels : Quoi ?
  - Besoins non- (ou extra-) fonctionnels : Comment ?
  - Besoins techniques : Quels technos ?
- Attribut (facteur ou caractéristique) qualité : c'est une propriété observable ou mesurable, statique ou dynamique d'un produit ou d'un processus logiciel
  - Ceux qui nous intéressent, ici, sont ceux des produits logiciels
- Exemples d'attributs qualité : maintenabilité, portabilité, fiabilité, efficacité, facilité d'utilisation, etc.

Modèles de qualité

- Modèle de qualité : c'est une taxonomie des attributs qualité et des relations entre ces attributs (attributs internes et externes)
- Objectifs d'un modèle de qualité :
  - permettre une spécification détaillée des besoins d'un système
  - permettre une évaluation de la conception
  - permettre des tests du système
- Exemples de modèle de qualité :
  - Modèle de McCall (1977) : 3 niveaux et 11 facteurs
  - Modèle de Boehm (1978) : 3 niveaux hiérarchiques et une douzaine de caractéristiques + métriques
  - Modèle du SEI (1995, puis 2002) : modèle ciblant les arch. logicielles, composé de 3 niveaux : *system quality*, *business quality*, *arch. quality*
  - Modèle du standard ISO 9126 (2001) : 2 niveaux, 6 caractéristiques et chacune dispose d'un certain nombre de sous-caractéristiques

Modèle de qualité ISO/IEC 9126



Qualité et décision architecturale

- Les décisions architecturales sont souvent prises en réponse à des exigences non-fonctionnelles (besoins qualité)

- Exemple :
  - **Besoin qualité :** le système doit être facilement maintenable
  - **Décision architecturale :** choix du style architectural « système en couches »

Style architectural

- Un style définit un vocabulaire d'éléments de conception (types de composants et de connecteurs) et l'ensemble de règles de bonne formation (contraintes de style) qui doivent être satisfaites par toute architecture écrite dans ce style.
- Il existe dans la littérature un certain nombre de styles[1] :
  - les systèmes de flux de données, comme le pipe & filter
  - les systèmes d'appels et retours d'appels, comme les systèmes à objets ou les systèmes en couches
  - les systèmes centrés sur les données, comme les tableaux noirs ou les systèmes hypertextes
- Une classification plus récente est disponible ici : http://www.booch.com/architecture/

[1]Mary Shaw et David Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)

Patron architectural

- Un style architectural impose des règles d'un niveau d'abstraction assez élevé : existence de *pipes* et *filters*, par exemple

- Un patron architectural : ~ une transposition de schémas de réutilisation connus dans le développement de logiciels à objets dans le domaine des composants

- Exemple de patron architectural : Façade
  - Existence d'un composant unique dans l'architecture interne d'un composant composite qui exporte une interface fournie

Contraintes architecturales

- Les langages existants fournissent les moyens nécessaires pour exprimer des descriptions d'architecture et pour documenter les décisions architecturales de façon informelle
- Peu de langages se sont intéressés à comment exprimer de façon formelle ces décisions
- Exprimer de façon formelle, c'est décrire les **contraintes** imposées par les décisions architecturales (styles et patrons, entre autres) de façon à pouvoir automatiser leur vérification
- On ne s'intéresse pas ici aux autres aspects de la description des décisions architecturales[1], on se focalisera uniquement sur les contraintes architecturales

[1] J. Tyree et A. Akerman. Architecture Decisions: Demystifying Architecture. Vol. 22 , Issue 2, Pages 19 – 27, IEEE Software, 2005.

Questions