

IFT6803: Génie logiciel du commerce électronique

Chapitre 3: Conception orientée objet

Section 1: Conception: méthodes et principes

Sommaire

Chapitre 3, Section 1

«Conception: méthodes et principes»

3.1.1 La conception : introduction

3.1.2 Concevoir pour changer

3.1.3 Famille de produits

3.1.4 Modularisation

3.1.5 Gérer les anomalies

3.1.6 Stratégies de conception

3.1.7 Qualités d'un bon design

3.1.1 La conception: introduction

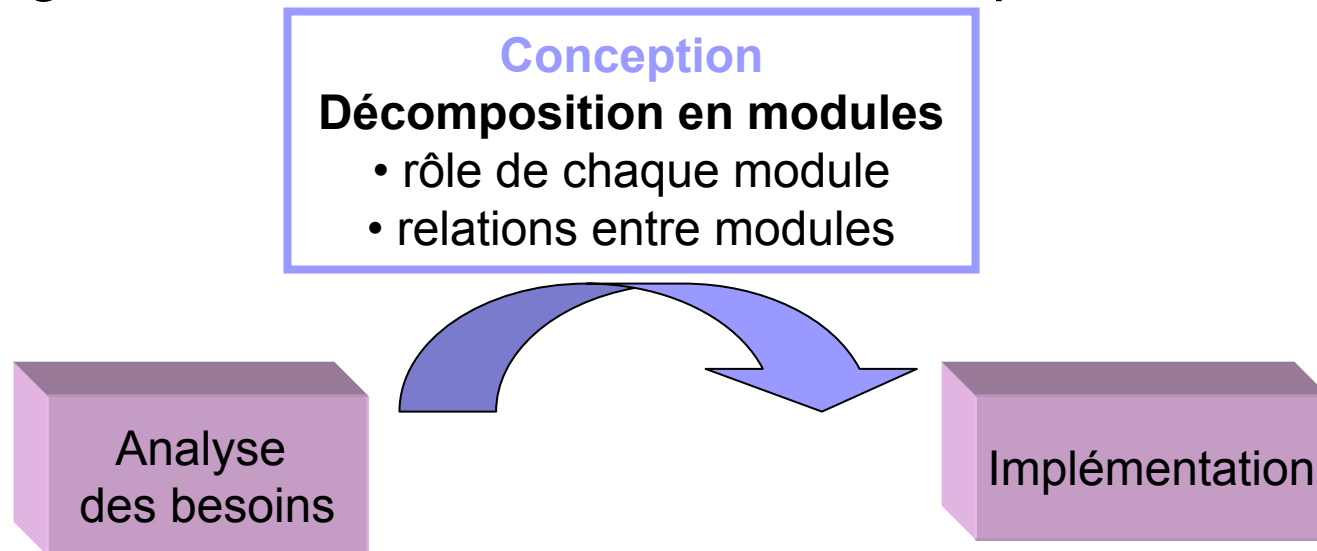
Activité de conception

- Étape cruciale du développement logiciel: pont entre l'analyse des besoins et l'implémentation.
- Activité exigeant créativité.
- Pas de recettes toute faites.
 - mais il existe des méthodologies (principes & bonnes pratiques) pouvant être de bon conseil
- Résultat de la phase de conception: une conception (aussi appelée « design »)
- Une bonne conception contribue à la qualité du logiciel: fiabilité, correction, évolutivité, etc.

La conception: introduction

Activité de conception

- Pont entre la définition des besoins et l'implémentation
 - Activité itérative/incrémentale qui transforme progressivement les besoins vers un produit final.

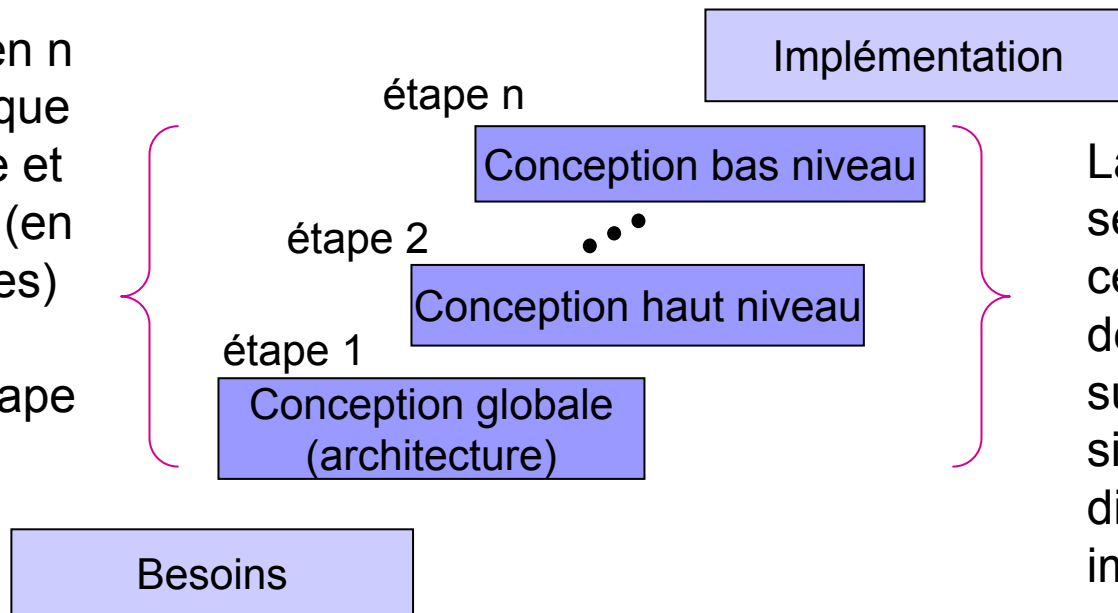


La conception: introduction

Activité de conception

- But: Décomposition progressive du système en modules de plus en plus détaillés.

Processus en n étapes: chaque étape raffine et décompose (en sous-modules) les modules définis à l'étape précédente.



La décomposition se poursuit jusqu'à ce qu'on obtienne des modules suffisamment simples pour être directement implémentés.

La conception: introduction

Types de conception

- **Conception architecturale** (conception de haut niveau, conception globale)
 - Structure et organisation générale du système à concevoir
 - Première étape qui consiste à définir les fonctions des éléments d'un système et leurs relations fonctionnelles.
 - Contient: description des éléments principaux, les relations entre eux, les contraintes à respecter, les motifs et la logique de cette décomposition.
- **Conception détaillée**
 - Étape qui consiste à détailler les résultats de l'analyse fonctionnelle, jusqu'à un niveau suffisant pour en permettre finalement le codage dans un langage de programmation choisi.
 - Définition d'un design logiciel respectant le plan de la conception architecturale.

La conception: introduction

Caractéristiques d'une bonne conception

- Une bonne décomposition en modules (i.e. un bon *design*) devra favoriser:
 - **une forte cohésion**: les éléments ne sont pas réunis dans un même module par hasard: ils forment un tout dans le but de réaliser une tâche commune.
 - **un faible couplage**: les modules sont relativement indépendants, ne dépendent pas trop des éléments définis dans d'autres modules.
 - **l'abstraction et le masquage d'information**

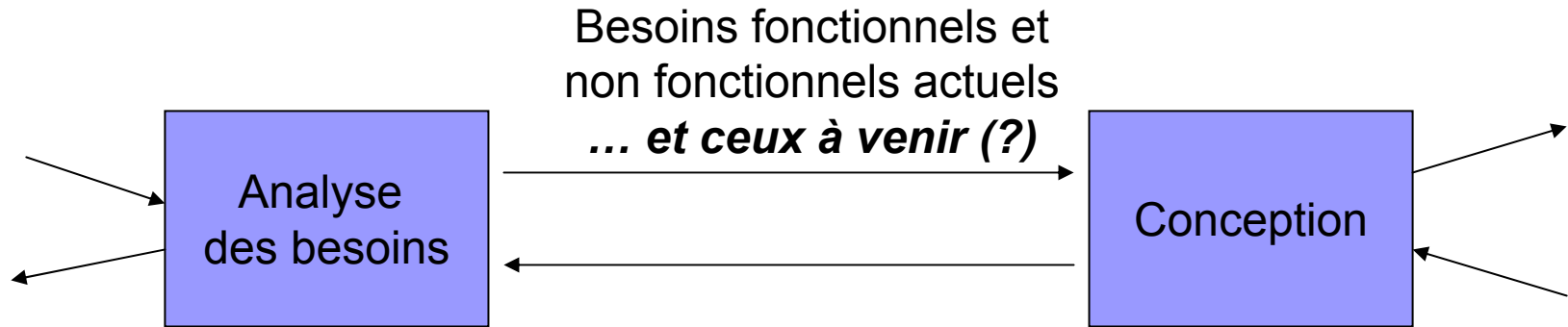
La conception: introduction

Activité de conception

- Qualités logicielles importante en jeu:
 - **Évolubilité**: si le logiciel est difficile à changer, les autres qualités sont directement compromises: fiabilité, performance, etc.
 - **Réutilisation**: on veut minimiser les coûts et rentabiliser les efforts de développement.

- Principes mis en œuvre:
 - Rigueur et formalité
 - Séparation des questions
 - Modularisation
 - Abstraction
 - **Anticipation des changements**
 - « Concevoir pour changer »
 - **Incrémentalité**
 - « Famille de produits »

3.1.2 «Concevoir pour changer»



Un des principaux soucis de l'activité de conception:
Développer un design qui facilite l'ajustement du système aux changements.

Pendant la conception:

- Importante qualité logicielle en jeu: **évolubilité**
- Principe mis en pratique: **anticipation du changement**

«Concevoir pour changer»

Pourquoi?

Quelques conséquences indésirables

- Un design, même « merveilleux », peut se révéler extrêmement difficile et coûteux à adapter.
 - Conséquence: nécessité de refaire un tout nouveau design pour intégrer un changement apparemment mineur...

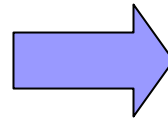
- En essayant d'accommoder l'architecture aux changements, le concepteur risque de briser l'élégante structure initiale du design.
 - Conséquence: application de plus en plus difficile à maintenir et inspirant peu confiance (fiabilité compromise ?)

«Concevoir pour changer»

Types de changements

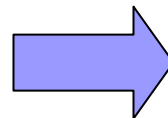
Sources de vrais changements

Changements
(perfectionnements) du
système imposés par les
nouvelles exigences du
client ou utilisateur



Maintenance perfective

Changements
(adaptations) imposés par
la modification de
l'environnement matériel,
social, etc.



Maintenance adaptative

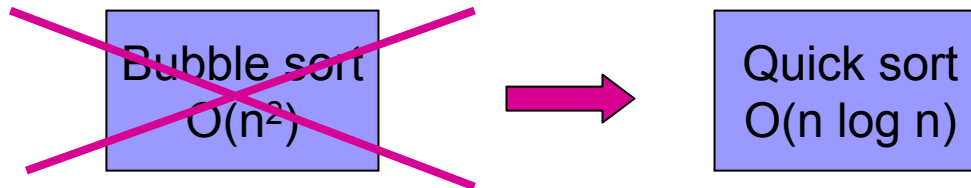
«Concevoir pour changer»

Types de changements

- 1. Changement d'algorithme

Perfectionnement souhaité: Améliorer l'efficacité, mieux répondre aux contraintes de sécurité, réduire l'espace utilisé, etc.

Ex. Changer l'algorithme de tri en fonction du nombre ou de la distribution des données...



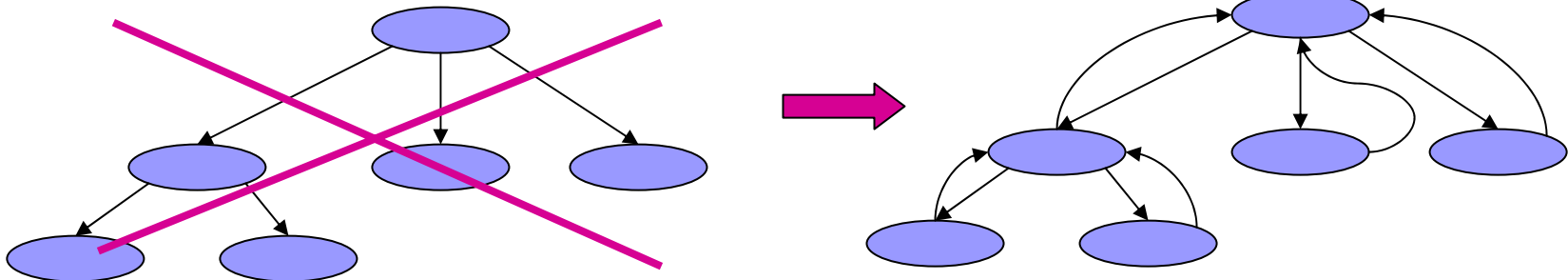
«Concevoir pour changer»

Types de changements

■ 2. Changement de représentation des données

Perfectionnement souhaité: Améliorer l'efficacité, réduire l'espace utilisé, offrir plus de flexibilité de traitement, ajouter de l'information, etc.

- Ex. Passer d'un tableau à une représentation par liste chaînée. Coût réduit pour l'insertion d'un élément.
- Ex. Ajout de pointeurs vers les noeuds parents dans un arbre. Navigation ascendante facilitée.

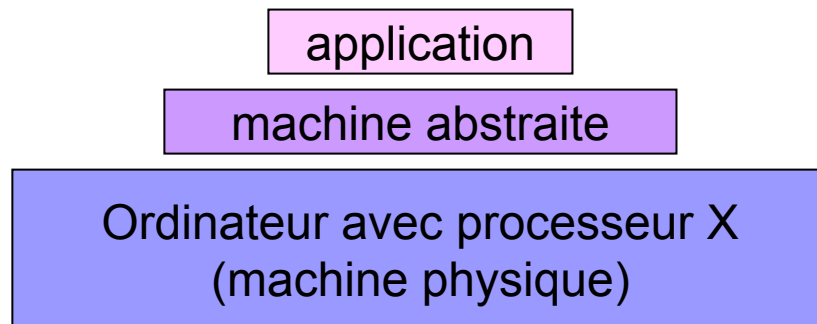


«Concevoir pour changer»

Types de changements

■ 3. Changement au niveau de la machine abstraite sous-jacente

Machine abstraite = Logiciel installé sur un ordinateur permettant de simuler le fonctionnement d'un dispositif matériel qui normalement ne pourrait pas fonctionner avec cet ordinateur.



- Application: byte code java, programme SQL, etc
- Machine abstraite: JVM, Oracle DBMS, etc.
- Ordinateur: Sparc, processeur Pentium, etc

«Concevoir pour changer»

Types de changements

- Changement de la machine abstraite sous-jacente (suite)

Remarque: Changement apporté, non pas au système développé, mais à l'environnement d'exécution pour, par exemple, optimiser les calculs, réduire l'espace utilisé, etc.

Adaptation possiblement nécessaire.

- Ex. Reformater la base de données pour satisfaire nouvelles contraintes de stockages des données par le DBMS.

«Concevoir pour changer»

Types de changements

■ 4. Changement au niveau des périphériques

Remarque: Changements apportés aux périphériques (pas au système développé) pour, par exemple, les rendre plus intelligents et autonomes.

Adaptation possiblement nécessaire pour profiter et s'accommoder de ces modifications aux périphériques...

«Concevoir pour changer»

Types de changements

■ 5. Changement de l'environnement social

Remarque: L'environnement « social » (non informatique) pour lequel le système a été développé change et évolue...

Adaptation possiblement nécessaire pour répondre aux contraintes de cet environnement.

- Ex. Modification de la loi de taxation. Nécessité de modifier la formule de calcul des taxes.
- Ex. Passage à l'euro. Nécessité de convertir le format des données monétaires.

«Concevoir pour changer»

Types de changements

■ 6. Changements dus au processus de développement

Si la nature du processus de développement est itérative et/ou incrémentale.

A prévoir: les versions livrées seront inévitablement l'objet de modifications ultérieures:

- De nouvelles parties pourront venir s'ajouter et devront s'ajuster aux anciennes (développement incrémental)
- Des parties raffinées pourront venir remplacer certaines anciennes parties moins détaillées. (développement itératif)

3.1.3 Famille de produits

Le développement itératif et/ou incrémental donne lieu à la production de différentes versions logicielles livrables.

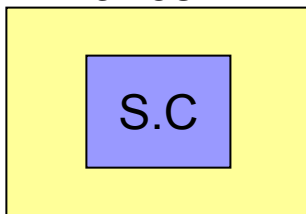
Nouvelle version produite pour

- remplacer une ancienne version (une version écrase l'autre, développement en séquence)
- Ou co-exister avec les autres versions de la famille (version se différenciant par ses nouvelles fonctionnalités, ou parce que destinée pour un nouvel environnement, etc.)

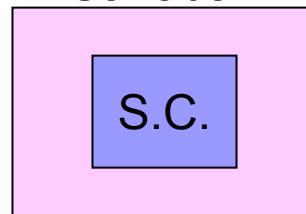
Famille de produits

- Famille de produits = Ensemble de versions partageant des parties communes:
 - Architecture, composants, interfaces, fonctionnalités, etc.
- Exemple: Famille de téléphones cellulaires
 - Partie commune: système de contrôle comportant les fonctionnalités de base (faire/recevoir un appel, annuaire, etc.)
 - Parties différentes: interface avec l'environnement externe (fonctions de gestion du réseau, langues de l'interface utilisateur, gestion de la sécurité, etc.)

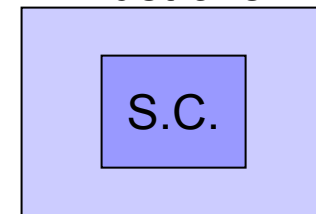
v.France



v. Canada



v.Australie



Famille de produits

- Développement de familles de produits

On désire maximiser les parties communes entre les différentes versions.

- Permet de réduire les risques d'incohérences entre les versions.
- Permet de réduire l'effort global de maintenance (« en connaître une, c'est les connaître toutes... ou presque! »).

3.1.4 Modularisation

Définitions

- Module: Unité fournissant des ressources et/ou des services.
- Objectif: déterminer la structure modulaire du système à développer
 - Quels sont les modules ?
 - Quelles relations lient les modules entre eux ?

En UML:

module = classe, package, composant
(aussi: cas d'utilisation)

Modularisation

Définitions

- Soit S un système logiciel composé des **modules** M_1, M_2, \dots, M_n et une **relation** r (irréflexive) définie sur $S \times S$.

$$S = \{M_1, \dots, M_n\}$$

$$r \subseteq S \times S$$

- La **fermeture transitive** de r , notée r^+ , est définie comme suit:

Soit $M_i, M_j \in S$, $M_i r^+ M_j$ si et seulement si $M_i r M_j$

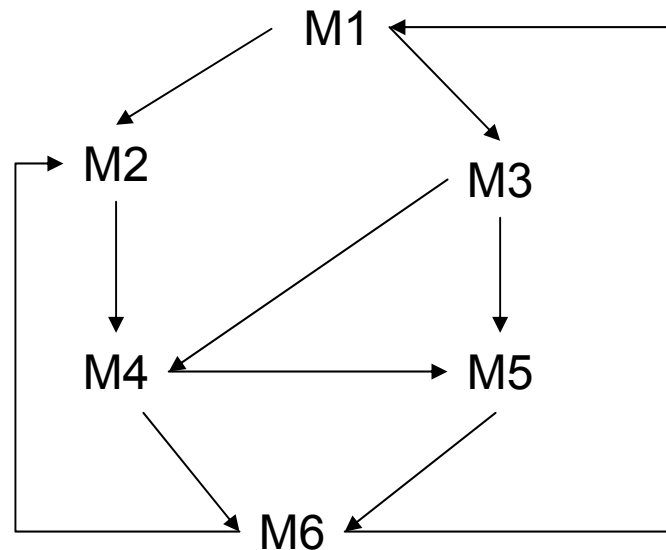
ou $\exists M_k \in S$ tel que $M_i r M_k$ et $M_k r^+ M_j$

- Une relation r est une **hiérarchie** si et seulement si il n'existe pas $M_i, M_j \in S$ tels que $M_i r^+ M_j$ et $M_j r^+ M_i$

Modularisation

Définitions

On peut représenter la relation r par un graphe.



- S'il y a un chemin de M_i à M_j alors $M_i r^+ M_j$
- S'il n'y a pas de cycle dans le graphe, r est une hiérarchie.

Modularisation

Relations entre modules

Deux types de relations utiles pour décrire la structure d'un système

□ Relation « UTILISE »

- UML: association, agrégation/composition, généralisation, liens de dépendance.

□ Relation « CONTIENT »

- UML: regroupement en paquetages.

Modularisation

Relation « UTILISE »

- On dit que M_i UTILISE M_j si M_i requiert la présence de M_j car M_j lui fournit des ressources ou des services

Modularisation

Relation « UTILISE »

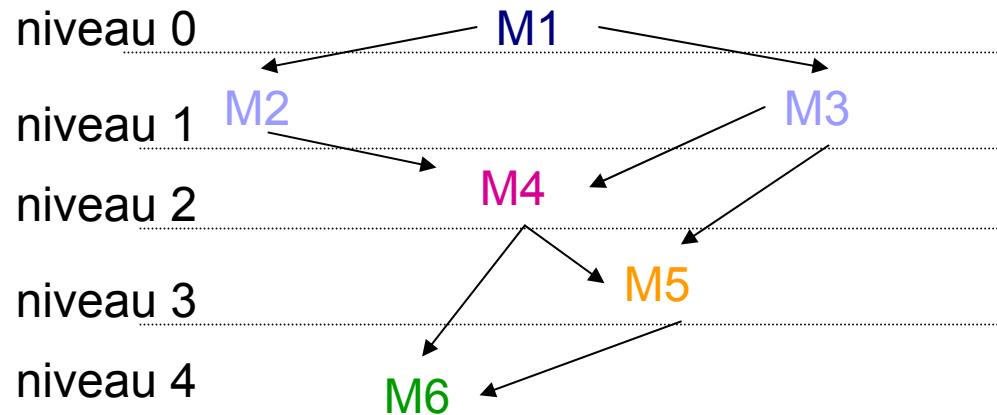
La relation UTILISE doit idéalement être une **hiérarchie**.
Pourquoi est-ce préférable ?

- Facilite la compréhension de la structure (par niveau d'abstraction)
- Facilite le test unitaire...
- Autrement: on peut se retrouver avec un système qui ne marche pas jusqu'à ce que tout marche (Parnas)

Modularisation

Relation « UTILISE »

La relation définit des *niveaux d'abstraction*



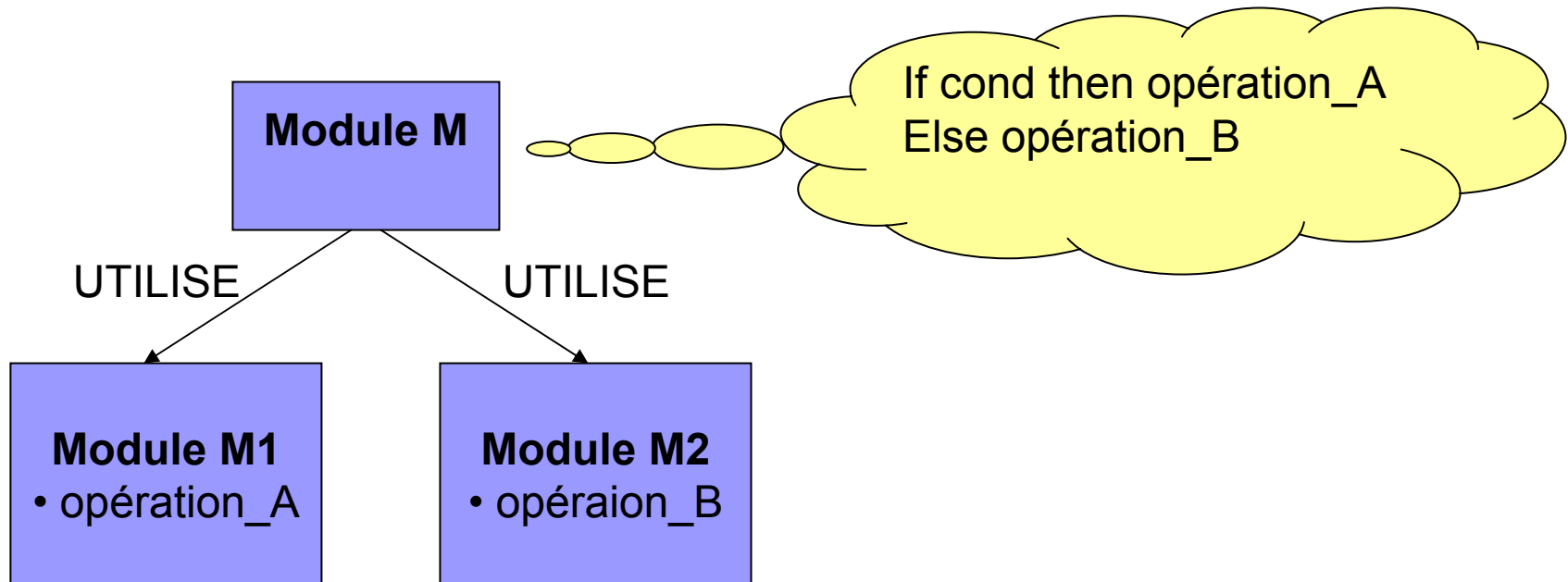
On définit le niveau d'un module M_i dans un hiérarchie r comme suit:

- S' il n'existe pas de M_k tel que $M_k r M_i$, $\text{niveau}(M_i)=0$
- Sinon $\text{niveau}(M_i) = \max(\{\text{niveau}(M_k) \mid M_k r M_i\}) + 1$

Modularisation

Relation « UTILISE »

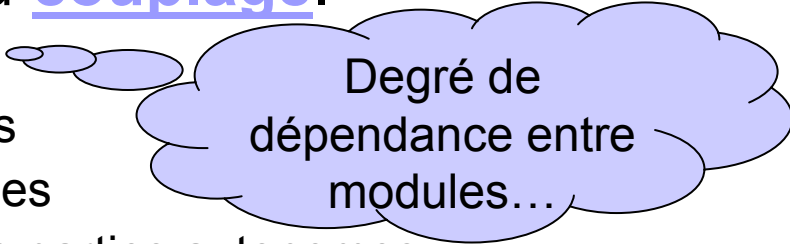
- Relation définie statiquement i.e. indépendante de toute exécution du logiciel.



Modularisation

Relation « UTILISE »

- Donne une indication *partielle* du couplage.
 - Si la cardinalité de r est $n(n-1)$
 - chaque module utilise tous les autres
 - Très forte dépendance inter – modules
 - Très difficile de diviser le système en parties autonomes.
 - Très fort couplage!
 - Si la cardinalité de r est 0
 - Chaque module est totalement indépendant.
 - On peut diviser le système en parties autonomes.
 - Aucun couplage!
 - Peu réaliste...
 - En général, cardinalité entre 0 et n^2 ... chercher un compromis.

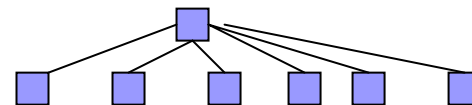
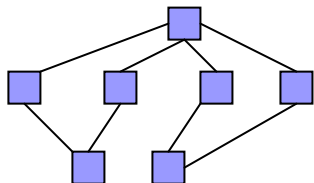


Degré de dépendance entre modules...

Modularisation

Relation « UTILISE »

- On doit trouver une solution de compromis raisonnable qui offre un couplage faible.
- Un approche suggérée:
Un bon design devrait limiter la sortance (fan-out) et favoriser l'entrance surtout dans les niveaux plus profonds de la relation UTILISE). Pourquoi ?
 - Entrance : nombre d'arêtes entrantes dans un module.
 - Sortance: nombre d'arêtes sortantes dans un module.
- On recommande une relation « UTILISE » qui présente une silhouette en cigare... A éviter: les galettes!



Modularisation

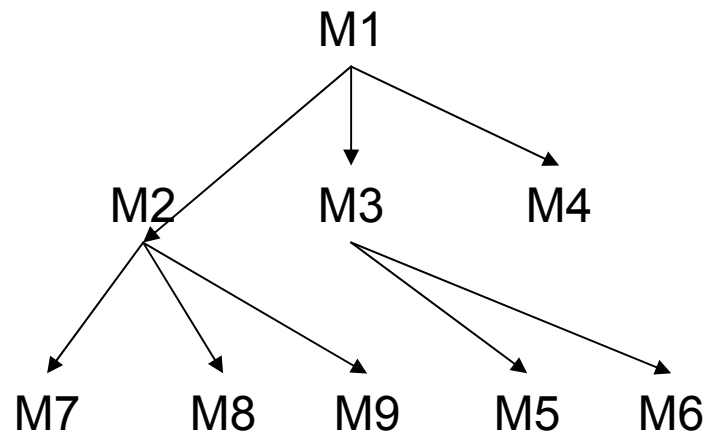
Relation « UTILISE »

- À noter qu'il ne suffit pas d'évaluer la relation UTILISE d'un design pour évaluer le couplage. D'autres types d'interactions peuvent contribuer au couplage.
 - Ex. C, Pascal: Communication entre deux modules via une variable globale.
 - Ex. Assembleur: modification des instructions d'un module par un autre module.
 - Etc.

Modularisation

Relation « CONTIENT »

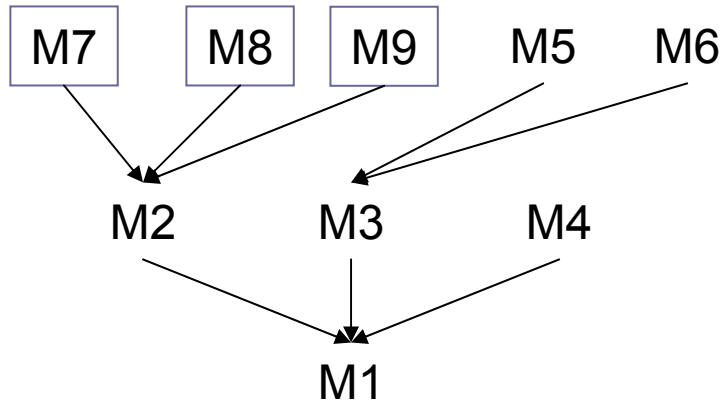
- On dit que M_i CONTIENT M_k si M_i est réalisé en assemblant un ou plusieurs modules dont le module M_k



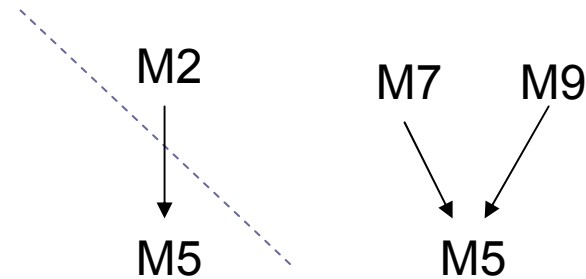
« CONTIENT »

Modularisation

Interface et implémentation



« EST_COMPOSANT_DE »



« UTILISE »

- Raffinement de la conception: on décide que M2 est composé de M7, M8, M9.
- Qu'advient-il de la relation M2 UTILISE M5 ? On doit savoir ce que M2 utilisait de M5 et ce que M5 rendait effectivement disponible... On doit définir ce que M7, M8, M9 utilisent de M5...

Comment connaître la nature exacte de la relation « UTILISE » entre deux modules ?

Ce qui va nous aider: INTERFACE DU MODULE

Modularisation

Interface et implémentation

Partie publique d'un module

- Interface: Vitrine décrivant l'ensemble des ressources (opérations, attributs et autres informations pertinentes) rendues accessibles aux modules clients.
 - La visibilité des éléments de l'interface peut être contrôlée en utilisant les mécanismes « public », « private », « protected ».
 - Pour faire la conception d'un module M, on a besoin que des interfaces des autres modules que M pourra utiliser.

Partie secrète d'un module

- Implémentation : Façon dont les ressources sont concrètement représentées et réalisées dans le module.

Principe de séparation des aspects:

- Décider quoi offrir ? (analyse & conception)
- Décider comment le réaliser ? (conception et implémentation)

Modularisation

Interface et implémentation

Masquage d'information

- Les clients d'un module n'ont accès qu'à l'information disponible dans son interface et satisfaisant aux contraintes de visibilité.
- Avantages de cette approche ? Meilleure adaptation aux changements (cf. section 4.1.1)
- Défi : déterminer ce qu'on doit rendre accessible, de ce qu'on doit cacher et ce, en ayant pour objectif un design ayant un couplage faible et une cohésion forte.

Modularisation

Interface et implémentation

Les interfaces doivent contenir...

- Seulement l'info nécessaire:
 - L'interface doit révéler le moins d'information possible...
- Toute l'info nécessaire:
 - L'interface doit donner suffisamment d'information aux autres modules pour qu'ils puissent utiliser les ressources offertes
- Pour favoriser l'évolutibilité du système
 - Offrir une interface abstraite
 - Cacher les détails de bas niveau (algorithme, représentation des données, etc.)

3.1.5 Gérer les anomalies

Conception défensive

- Malgré la rigueur du développement, impossible d'avoir une confiance inconditionnelle dans le système développé.
- On doit développer des logiciels **robustes** i.e. qui ont un comportement raisonnable même dans les circonstances imprévues.
- État d'un module qui ne peut offrir le service tel qu'attendu et spécifié par son interface. -> **état anormal**

Il faut pouvoir anticiper ces état anormaux, les gérer et les tolérer au mieux...

Gérer les anomalies

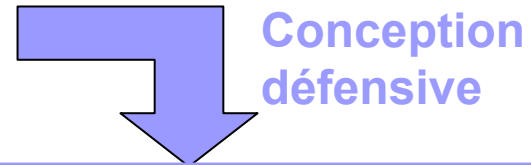
Conception défensive.

- État d'un module qui ne peut offrir le service tel qu'attendu et spécifié par son interface. -> **état anormal**
- Causes possibles de la défaillance d'un module M :
 - Service de M invoqué avec paramètres inadéquats.
 - M utilise un service défaillant exporté par un autre module
 - Circonstances imprévisibles (débordement, etc.)
 - etc.

Gérer les anomalies

Conception défensive

- Un module serveur (i.e. offrant le service)
 - soit s'exécute correctement et offre le service attendu tel que spécifié,
 - soit entre dans un état anormal



Le module serveur signale l'anomalie en levant une exception auprès du module client.

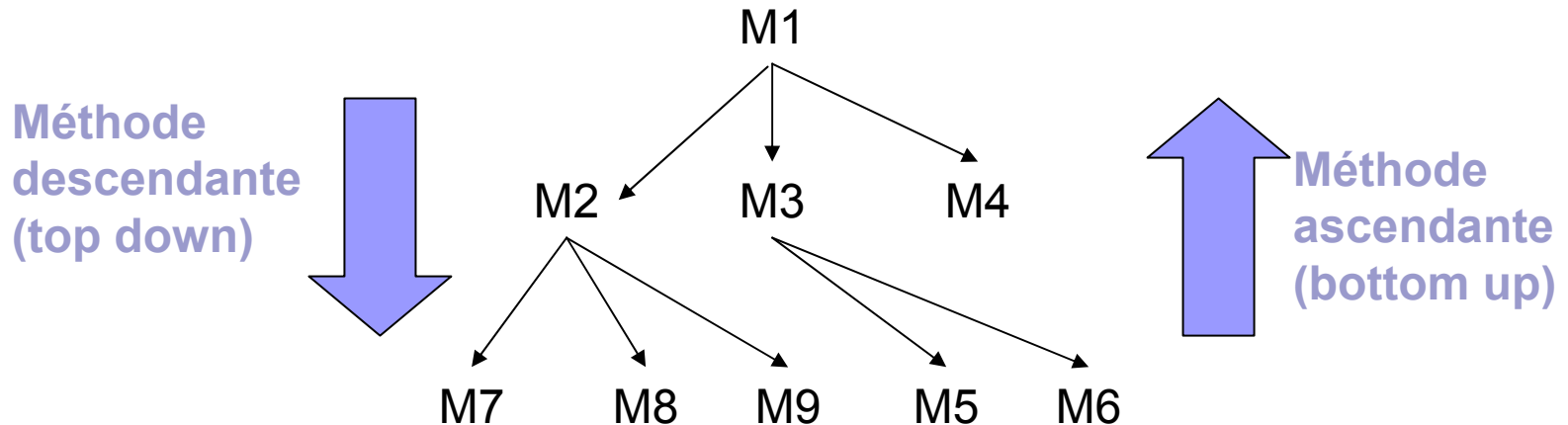
- Serveur termine son exécution
- Client notifié de l'anomalie, se charge de gérer l'exception adéquatement en activant un exception handler.

Les exception qu'un module serveur peut lever doivent être indiquées dans son interface.

3.1.6 Stratégies de conception

Comment procéder à la décomposition en modules ?

Relation « COMPREND »



Stratégies de conception

Stratégie descendante

■ Avantages:

- Vue d'ensemble du problème et de ce qu'on désire réaliser.
- Facilite la compréhension des problèmes et de leur décomposition. « Diviser pour régner »
- Recommandé pour documenter le design.

■ Inconvénients:

- Les sous - problèmes ont tendance à être analysés de façon isolé
- Principes de généralisation et masquage d'information non mis à profit.
- Ne favorise pas la réutilisation.

Stratégies de conception

Stratégie ascendante

■ Avantages:

- Permet d'identifier ce qu'il y a de commun entre les modules et permet d'appliquer les principes de masquage d'information et de généralisation.
- Favorise la réutilisation.

■ Inconvénients:

- Pas de vue d'ensemble. Quoi mettre dans les sous - modules ??
- Risque de consacrer beaucoup d'efforts à réaliser un module qui ne sera pas utilisé ...

Stratégies de conception

Stratégie mixte (« yo-yo »)

Design

= activité créative qui nécessite du jugement.

- Un bon concepteur appliquera une stratégie mixte:
 - ↓
 - On commencera par une stratégie descendante pour identifier les sous-systèmes cibles.
 - ↑
 - On verra ensuite à faire la synthèse des sous-systèmes en termes d'une hiérarchie de modules réutilisables construits en appliquant les principes de généralisation et de masquage d'information.

Stratégies de conception

De la conception vers l'implémentation

- Après validation du design:
 - Respect des propriétés fonctionnelles et non fonctionnelles.

- ... l'implémentation:
 - Stratégie descendante: tous les modules qui utilise le module M sont implémenté avant que M ne le soit.
 - M est temporairement simulé par un module *stub*.
 - Stratégie ascendante: tous les modules utilisés par un module M sont implémentés et testés avant que M soit implémenté.
 - M est temporairement simulé par un module *driver*.
 - Stratégie mixte: stratégie encouragée.

3.1.7 Qualités d'un bon design

Un bon design devrait favoriser
l'indépendance des modules

Pour évaluer l'indépendance des modules, on se base généralement sur les concepts suivants:

- Le couplage
- La cohésion

Concepts qui peuvent d'ailleurs s'influencer l'un et l'autre

Qualités d'un bon design

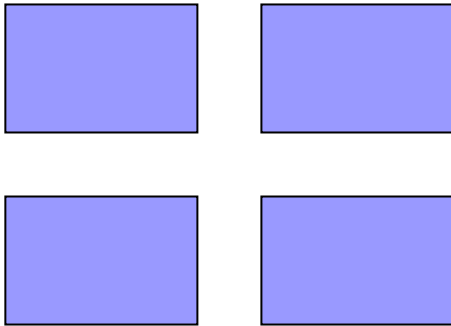
Couplage

- Mesure de l'interdépendance entre deux modules.
- Un ensemble de modules est faiblement couplé si les liens de dépendances (cf. interactions induisant une relation « UTILISE ») entre les modules sont peu nombreux.
- Un faible couplage est précurseur...
 - D'un bon découpage du système: les éléments qui dépendent les uns des autres ne sont pas « éparpillés » à travers les modules du système.
 - D'une facilité de maintenance: une modification dans un module affecte éventuellement un nombre restreint d'autres modules. Nombre de révisions réduites...

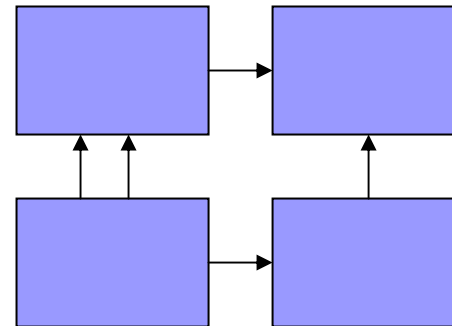
Qualités d'un bon design

Couplage

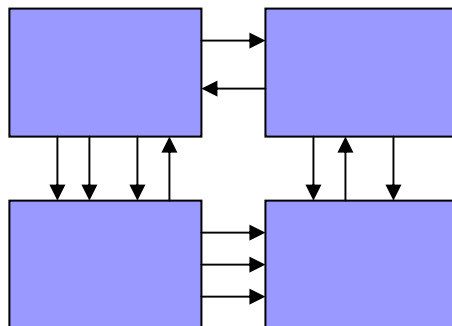
Systeme non couplé
Peu vraisemblable



Systeme faiblement couplé



Systeme fortement couplé

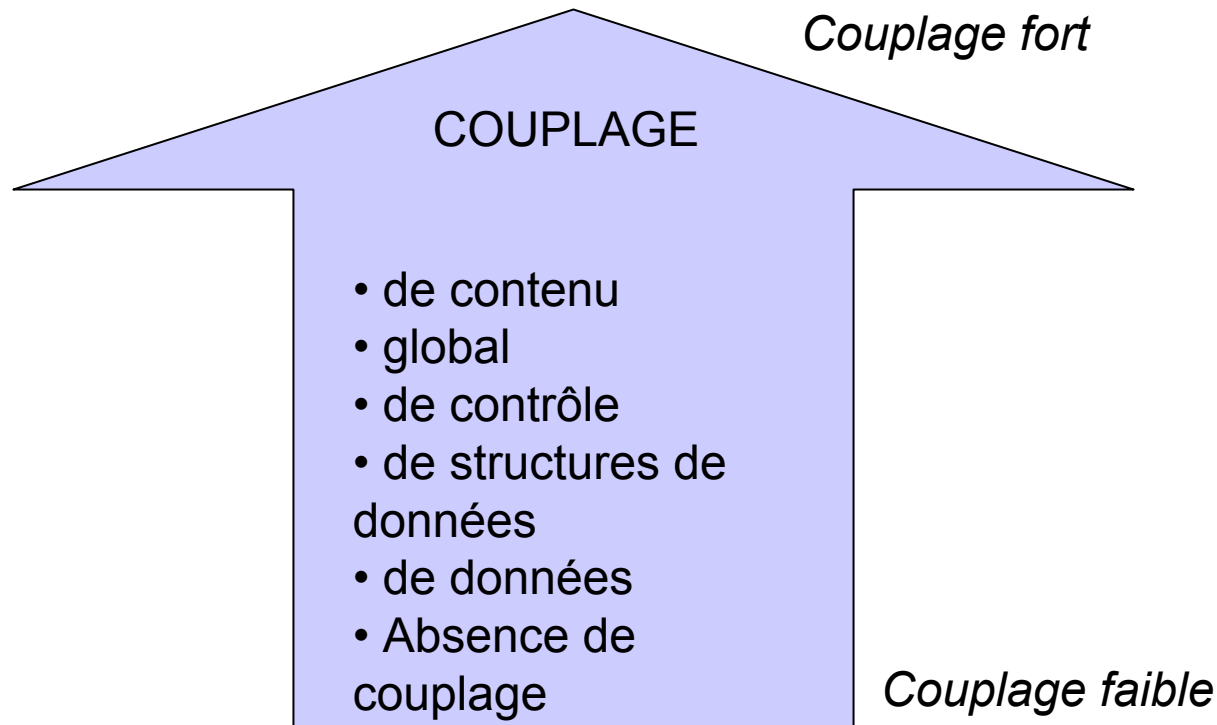


- Interaction d'utilisation
- appel de méthode/procédure
 - utilisation d'une variable
 - etc.

Qualités d'un bon design

Couplage

Types de couplage

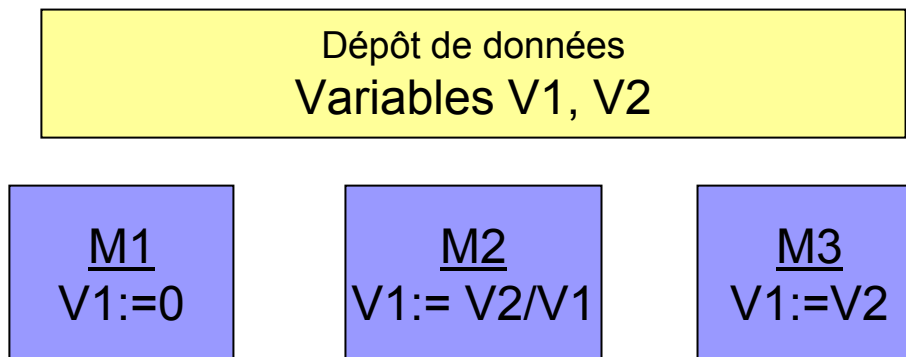


Qualités d'un bon design

Couplage

Types de couplage

- **De contenu:** Il y a couplage de contenu si un module fait référence et/ou modifie une partie du code de l'autre (i.e. intervient dans sa partie qui devrait être cachée (implémentation)).
 - Couplage absurde: généralement impossible dans langage de haut niveau... seulement en assembleur.
- **Global:** Il y a couplage globale si deux modules réfèrent à un même ensemble de données.



M1 risque d'entraîner une erreur chez M2 via la variable V1.

Qualités d'un bon design

Couplage

Types de couplage

- **De contrôle:** Il y a couplage de contrôle si un module contrôle la logique d'un autre module via des paramètres de contrôle ou des flags.

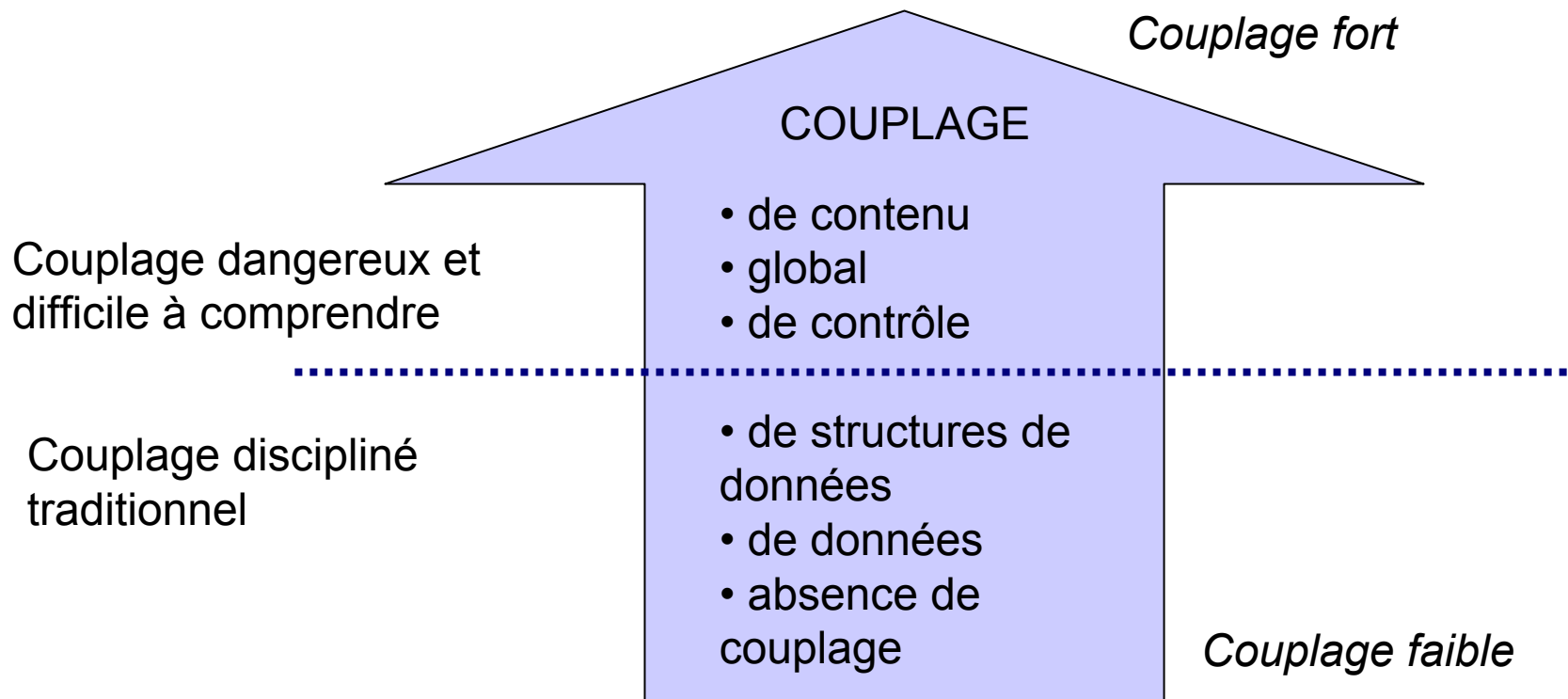
Manières disciplinées et traditionnelles d'interagir

- **De structures de données:** Il y a couplage de structures de données si un module passe une structure de données par argument à un autre module. Le module appelé n'a pas besoin de tous les éléments contenus dans la structure de données.
- **De données:** Il y a couplage de données si un module passe des données par argument à un autre module. Le module appelé utilise pas toutes le don

Qualités d'un bon design

Couplage

Types de couplage



Qualités d'un bon design

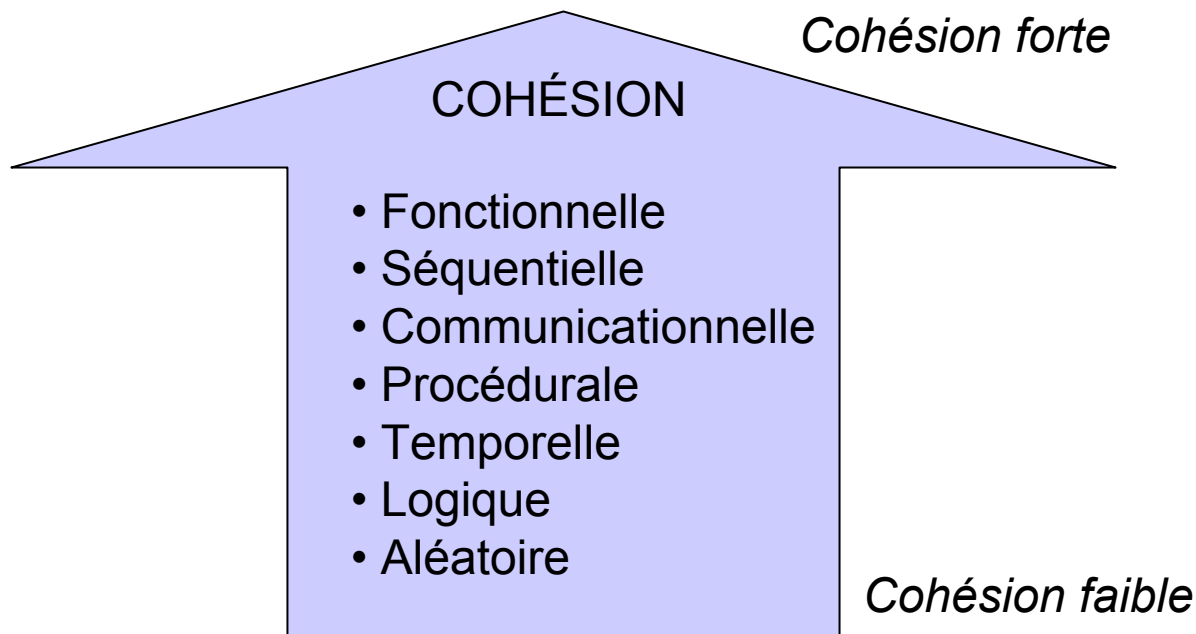
Cohésion

- Mesure de la force des relations qui unissent les éléments fonctionnels à l'intérieur d'un module.
- Un module est fortement cohésif si tous ses éléments sont destinés et sont essentiels à la réalisation d'une tâche commune unique.
- Une forte cohésion est précurseur...
 - D'un bon découpage du système: les éléments qui ont rapport les uns avec les autres se retrouvent dans un même module.
 - D'une facilité de maintenance: les éléments destinés à une même tâche sont regroupés et on peut facilement les retrouver.
 - D'un faible couplage: les éléments inter-dépendants se trouvant dans le même module, les dépendances inter-modules sont moindres...

Qualités d'un bon design

Cohésion

Types de cohésion



Qualités d'un bon design

Cohésion

Types de cohésion

■ **Aléatoire:** Les fonctions du module n'ont rien à voir les unes avec les autres. Réunies par pure commodité.

F1
F2
F3

F1: Réparer la voiture
F2: Faire un gâteau
F3: Étudier IFT2251

■ **Logique:** Fonctions réunies autour d'un thème commun...

F1
F2
F3

F1: Voyager en voiture
F2: Voyager en avion
F3: Voyager en train

■ **Temporelle:** Fonctions réunies ensemble car leurs moments d'exécution sont reliés dans le temps...

F1 [T]
F2 [T+X]
F3 [T+2X]

Au coucher...

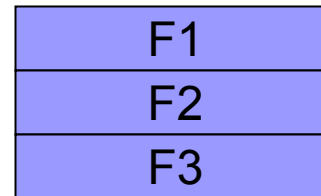
F1: Fermer la TV
F2: Se brosser les dents
F3: Fermer les lumières

Qualités d'un bon design

Cohésion

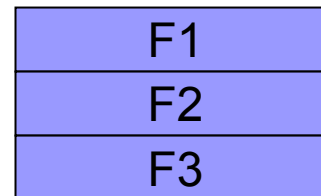
Types de cohésion

■ **Procédurale:** Fonctions réunies parce qu'elles doivent être exécutées dans un ordre donné. (flot de contrôle passe d'une fonction à une autre, boucle, conditionnelle....)



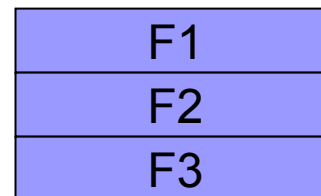
F1: Préparer la dinde
F2: Préparer les légumes
F3: Mettre la table

■ **Communicationnelle:** Fonctions réunies car elle produisent ou opère sur le même type de données.



F1: Trouver titre d'un livre
F2: Trouver prix d'un livre
F3: Trouver auteur d'un livre

■ **Séquentielle:** Fonctions réunies car les sorties de l'une servent d'entrées pour l'autre... (flot de données...)



F1: Remplir les trous
F2: Sabler la voiture
F3: Donner une première couche de peinture

Qualités d'un bon design

Cohésion

Types de cohésion

■ **Fonctionnelle:** Toutes les fonctions réunies contribuent à l'exécution d'une même et unique tâche.

Le module contient tous et seulement les fonctions nécessaires pour la réalisation de la tâche. Le module réalise une seule et unique tâche.

F1
F2
F3
F4
F5

Repeindre une voiture

F1: Laver la voiture

F2: Remplir les trous

F3: Sabler la voiture

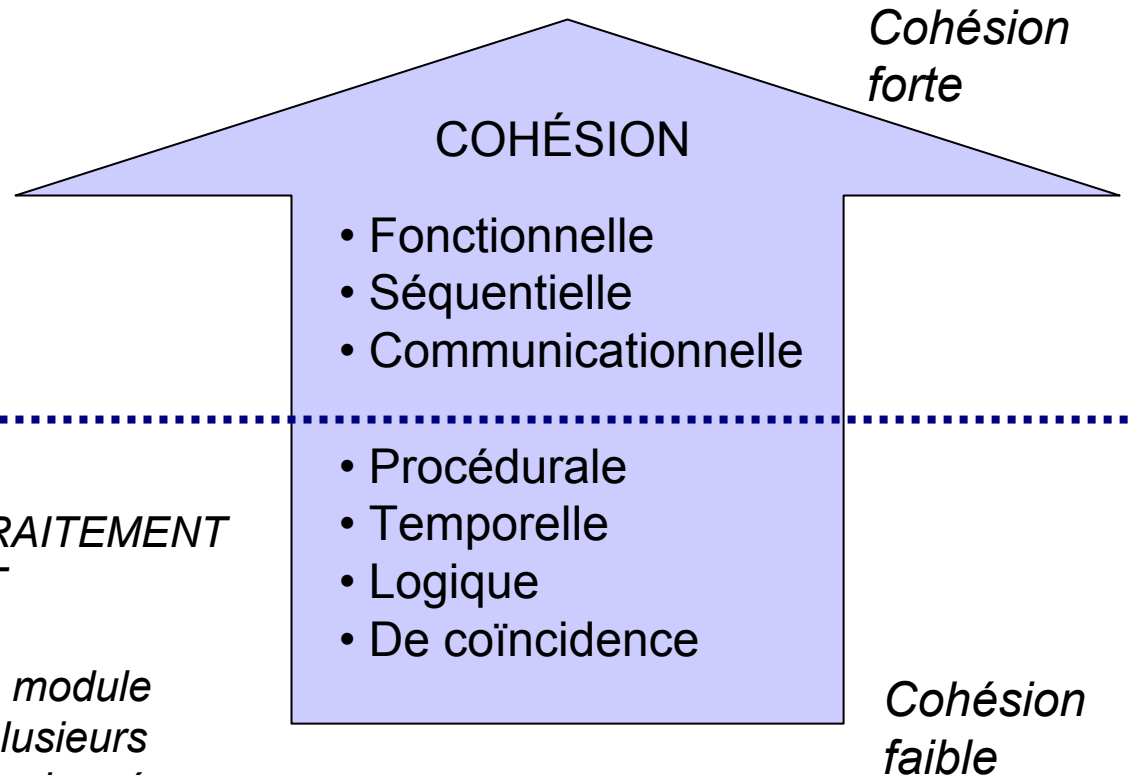
F4: Donner une première couche de peinture

F5: Donner une couche finale de peinture

Qualités d'un bon design

Cohésion

Facilité de maintenance



Plus facile à maintenir

*FONCTIONS S'APPLIQUENT À
DES DONNÉES COMMUNES*

Plus difficile à maintenir

*FONCTIONS CONCERNENT LE TRAITEMENT
DE DONNÉES ÉVENTUELLEMENT
DISPARATES...*

*Changement d'une fonction dans un module
peut avoir des répercussions dans plusieurs
modules qui traitent le même type de données...*