POO ?

- Programmation Orientée Objet
- Design Patterns

- 6 cours, 1 examen
- 6 TD/TP

- Cours : Philippe Finkel, finkel@univ-mlv.fr
- TD/TP : Philippe Finkel, Marc Berthoux

Objectifs

- Apprendre et maîtriser les grands principes de la Programmation Orienté Objet
- Apprendre et maitriser les principaux Design Pattern

- Et aussi :
  - Introduction au Génie Logiciel
  - « Visiter » l'utilisation de Tests
  - « Visiter » le refactoring

POOurquoi ??

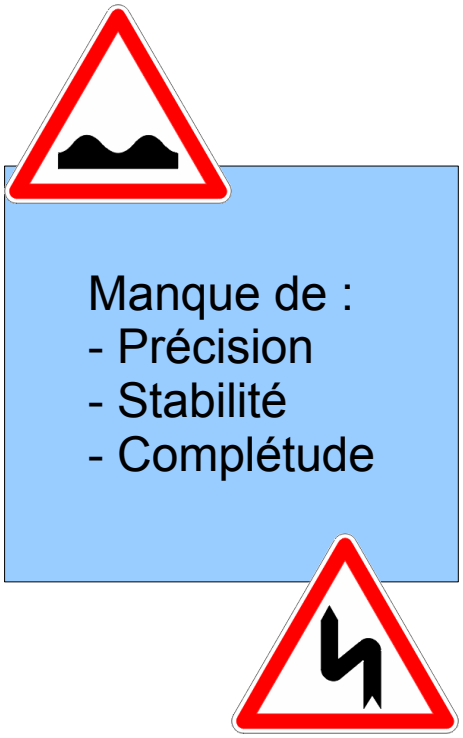
- La plupart des projets informatiques se passent mal !
  - dépassement important des <u>délais</u>
  - dépassement important des <u>coûts</u>
  - non <u>conformité</u> :
    - bugs,
    - limitations,
    - "flou" car comportement attendu "discutable" (imprécision, quiproquo, ...)
  - difficulté et coût important de de <u>maintenance</u>

POOurquoi : qualité

- Qualité insuffisante !

  - la qualité concerne bien sûr toute la chaîne :
    - expression de besoins
    - cahier des charges fonctionnel
    - préparation des cas tests
    - organisation des recettes,
    - …
  - qualité et l'implication des participants !!!
  - qualité et efficacité des organisations

  - Qualité du développement logiciel : conception, developpement, cycle de vie du logiciel

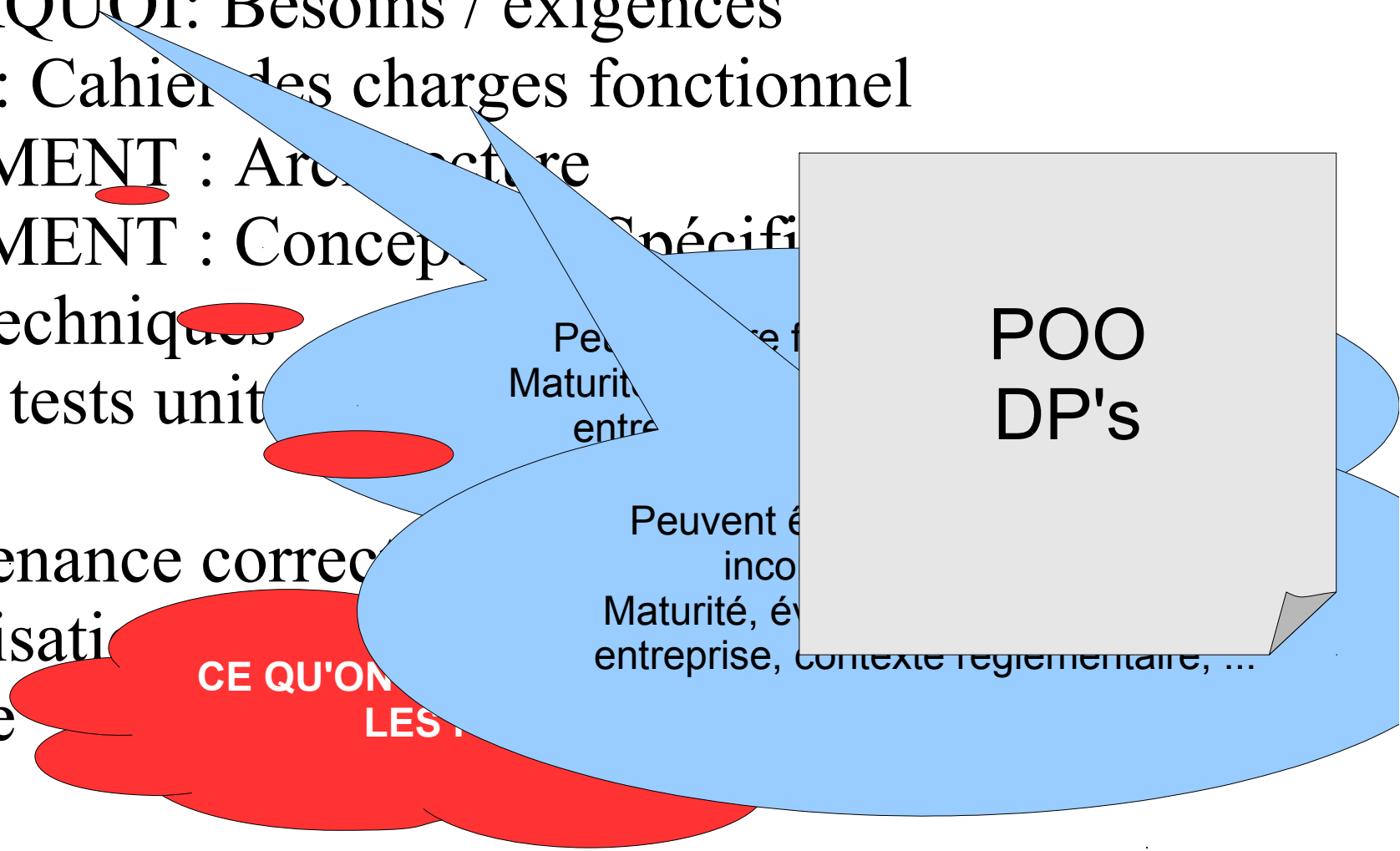


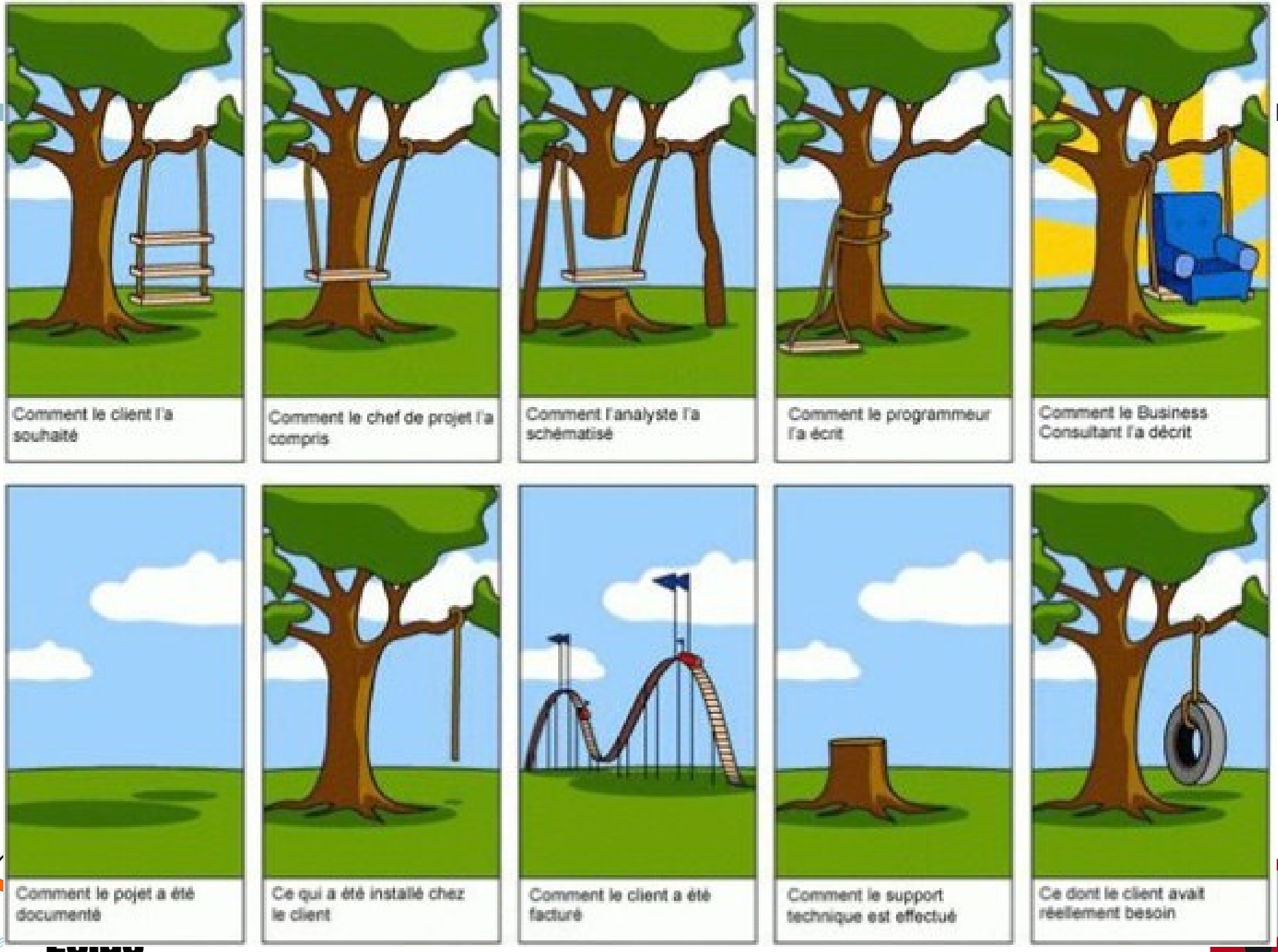
Histoire...

- crise du logiciel en 1968.
  - baisse significative de la qualité des logiciels
  - + puissance de calcul des ordinateurs => logiciels beaucoup plus complexes qu'auparavant
- La création de ces nouveaux logiciels plus complexes
  - Retards, dépassement de coût, - fiabilité, - performant
  - Coût important de maintenance
- baisse du coût du matériel informatique // augmentation du coût du logiciel.
  - études sur les méthodes de travail adaptées à la complexité des logiciels contemporains => **génie logiciel**
- l'utilisation des méthodes de génie logiciel se répand *doucement* dans l'industrie du logiciel.

Vie d'un logiciel

- POURQUOI: Besoins / exigences
- QUOI: Cahier des charges fonctionnel
- COMMENT : Architecture
- COMMENT : Conception Spécifications techniques
- Dev + tests unitaires + intégration
- Maintenance corrective
- Réutilisation, refonte




Comment le client l'a souhaité
Comment le chef de projet l'a compris
Comment l'analyste l'a schématisé
Comment le programmeur l'a écrit
Comment le Business Consultant l'a décrit
Comment le pojet a été documenté
Ce qui a été installé chez le client
Comment le client a été facturé
Comment le support technique est effectué
Ce dont le client avait réellement besoin

Architecture

- Une description de niveau supérieur du logiciel:
  - les composants principaux et la manière dont ils interagissent
  - le découpage de plus haut niveau d'un système en parties/sous-systèmes/composants
- Qui prend en compte le matériel, les autres logiciels et l'utilisateur, les canaux de communication
- Sans toutes ces information une erreur fondamentale peut arriver

- Du bon sens : *« Prendre les décisions qui seront dures à changer »*

Conception

- Définition générale :
  activité créatrice qui consiste à élaborer un projet ou une partie des éléments le constituant, en partant
  - des besoins exprimés
  - des moyens existants
  - des possibilités technologiques

dans le but de créer un produit ou un service

Analyse / Conception fonctionnelle

- Première étape de la conception
  - Définit les fonctions/responsabilités des composantes d'un système
  - Décrit leurs interactions fonctionnelles

- Fonctions principales, sans préjuger des solutions
  - « Lier » les deux parties et non « visser »...
  - «réduire la hauteur de l'herbe » vs « tondre »

- Prise en compte des contraintes
  - Sécurité
  - Respect de standard, de normes
  - Fixée par le client...

Conception descendante

- Le problème global est décomposé en sous-problèmes, eux-même décomposés en opérations plus élémentaires jusqu'à obtenir la granularité souhaitée

Conception ascendante

- On commence par définir les fonctions les plus élémentaires, pour ensuite les utiliser et définir des fonctions de plus en plus spécifiques et com-plexes

Conception de système

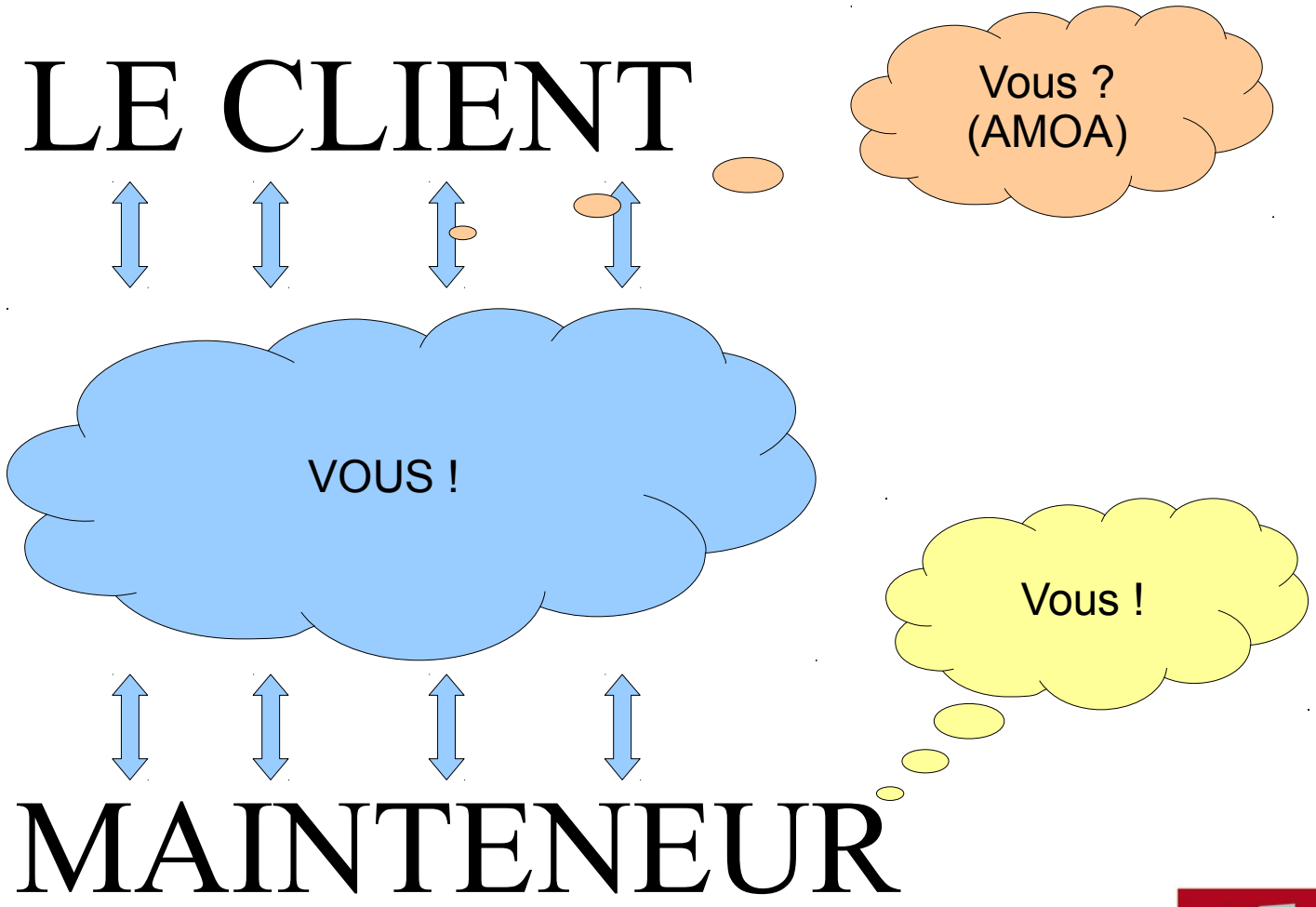
- La conception de système inclut
  - architecture matérielle
  - architecture logicielle
  - définition des composants
  - définition des interfaces
  - définition des *données*
- Recherche de solutions

Et alors ??

- La définition de conception ne nous donne pas d'idée sur comment en faire une...
- Il faut traduire les exigences en solutions

- Analyse = comprendre les exigences
- Concevoir = construire la solution

Deux AXES majeurs !



Client / Mainteneur

- Client
  - Veut être satisfait
  - Est celui qui paye !
  - N'est pas concerné par « comment » mais uniquement par « quoi »
- Mainteneur
  - Veut en faire le minimum !
  - SAIT que :
    - « changer = temps + bugs + régression + ... »
      => douleur
  - → VEUT minimiser les changements

Les risques

- Rigidité
  - difficile de faire évoluer le produit
  - Exemple : impossibilité d'ajouter une option au logiciel
- Fragilité
  - si certains points tombent, le logiciel complet s'écroule !
  - exemple mettre un CD rayé dans windows : tout est bloqué pendant 5 mn
- Immobilité
  - impossibilité de changer une fonctionnalité du système

Changements ... Dépendances

- Besoins de changements => risques
  - Immobilité : ne pas pouvoir les réaliser
  - Rigidité : c'est dur !
  - Fragilité : ça casse un peu partout

- Les risques sont liés aux impacts des changements
  - Attention particulière aux dépendances

Gestion des dépendances

- Dans un projet, les points cruciaux sont les dépendances entre modules
- Besoin de rigueur
- Trois cibles
  - Robustesse
  - Flexibilité / Extensibilité
  - Ré-utilisabilité

Dépendance

- A dépend de B
  - Si B est absent, dysfonctionnement
  - Si B est différent, dysfonctionnement
  - Si B est modifié
    - Recompilation ?
    - Continuité Syntaxique ?
    - Continuité Sémantique ?

Se protéger de ces difficultés :

- On fait un conception préalable
- On code en objet
- On utilise des interfaces
- Elles utilisent des *design patterns*
- On utilise *UML*

Modules

- **Définition :** Un module est une unité de code
- De toute taille
- À forte cohésion
  - classe, fichier
  - paquetage
  - bibliothèque

Module Ouvert

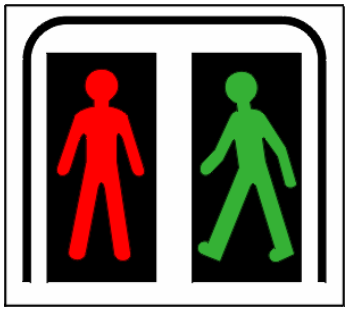
- Un module dans lequel il est encore possible de réaliser des modifications
- Ajout d'attributs, changement de code, ajout de classe publique...
- C'est ce que demande le concepteur d'un module qui veut pouvoir : le corriger, l'étendre, l'adapter

Module Fermé

- Compilable, validé, testé
- **Un module utilisable par d'autres** (ce qu'ils veulent)
- La valeur d'un module est plus grande s'il est fermé
  - avancement
  - debugging
  - réutilisation

Un rêve : Ouvert ET Fermé

- On aimerai à avoir des paquetages ouverts et fermés
  - sécurisé
  - extensible



Modules

- Les modules doivent correspondre à une unité syntaxique du langage
  - fichier
  - classe
  - bibliothèque
  - **paquetage**

Paquetages

- Les paquetages offrent les mêmes challenges que les classes
  - lisibilité
  - réutilisabilité
  - ouverture/fermeture
- Unité de programmation plus efficace pour l'organisation du travail et pour définir l'architecture logicielle
  - Conception
  - Cohésion
  - Documentation
  - Livraison

POO

- Constitution des modules
- Ecrire des modules
- Principes POO

Principe d'encapsulation

- Cacher le fonctionnement interne d'un module
- Les changements internes au module n'ont pas d'influence sur le code extérieur
- On cherche à obtenir des interfaces (au sens de la chimie)
- Concepts importants : contrat et responsabilité

Interface

- Quand deux matériaux ont une interface, cela veut dire qu'il n'y a pas d'échange de matière entre les deux
  - exemple : l'huile et l'eau
  - contre-exemple : l'or et l'argent
- On peut les séparer

Interface

- On cherche à écrire des modules qui ont des interfaces (au sens chimique)
- On veut que ni les corrections, ni les bugs ne se propagent d'un module à l'autre
- En particulier, les changements d'implémentation d'un module ne doivent pas avoir d'incidence sur les modules qui l'utilisent

Module client

- Quand deux modules interagissent, celui qui **utilise l'autre** est appelé module client
  - Ex Voiture/Conducteur
- Le conducteur utilise l'interface de la voiture et pas l'inverse
- Ex Administrateur/Administré
- Les deux modules font des demandes croisées. Les deux modules sont clients l'un de l'autre.

Interface

- L'interface est la partie publique d'un module
- Ce doit être la partie stable d'un module
- Un module client utilise l'interface
- Si l'interface change, les modules clients doivent changer

Interface

- **Une bonne partie du travail de conception est la définition des interfaces entre les différents modules**
- Il est important de bien définir l'interface
  - Syntaxique : prototype
  - Sémantique :
    - fonctionnel
    - contrat complet (prérequis, cas d'erreur, effet de bord?, ...)

De l'art d'écrire des Modules

- Le travail d'étude se différencie du travail de production
  - il s'applique à quelque chose de Neuf.
  - Ainsi le risque d'erreur y est beaucoup plus prégnant.
    Toutes les méthodes cherchent a réduire le coût de la correction d'erreur avec une stratégie qui consiste a faire cette détection au plus tôt.

- L'art d'écrire les modules à pour objectif de réduire le nombre d'erreurs.

Construction des modules

- Principe d'ouverture/fermeture
- Interfaces
- Fermeture commune
- Stabilité et abstraction
- Dépendances acycliques
- Livraison
- Ré-ouvrir ?

Principe d'ouverture/fermeture

- On cherche à avoir des modules ouverts/fermés
- Cette stratégie s'applique quelle que soit le type de module (classe, paquetage, .c)

Interfaces

- Le choix d'une interface est une activité primordiale qui doit être identifiée et maîtrisée en tant que telle
- C'est un savoir faire spécifique
- La qualité des interfaces est un préalable à la qualité du logiciel

Interfaces

- Objectifs pour une interface
  - réduire les risques d'ouverture
  - assurer la compréhensibilité
  - assurer la réutilisabilité
  - continuité

Petites interfaces

- Les interfaces pléthoriques vont à l'encontre de la com-préhensibilité du module
- Une interface importante n'aide pas à la réutilisabilité du module
- Si l'interface est grande, le module et la maintenance sont complexes

Peu d'interfaces

- Il faut réduire le nombre d'interfaces utilisées par un module :
  - un module doit être le client d'un minimum d'autres
- + de fournisseurs => + de chances d'avoir à changer
- + de fournisseurs => + de complexité

Interfaces explicites

- Les modules doivent avoir une interface explicite où rien n'est caché ou tordu
- La documentation du module devrait pouvoir se limiter à l'interface

Fermeture commune

- Les entités qui dépendent d'un même concept doivent être placé dans le même module
- Si le concept change, les classes concernées sont toutes dans le même paquetage
- **L'unité de fermeture est le module**
- Changer un concept force la réouverture, il est important qu'un nombre minimal de modules soit rouverts

Stabilité

- La stabilité est l'inverse du risque de changement
- Plus le module est client d'autre modules, moins il est stable
- Mesure mathématique : hauteur dans le graphe de dépendance

Valeur de la stabilité

- Plus un module est fermé donc stable, plus il a de valeur pour les développeurs
- Plus les modules stables (qui ont donc beaucoup de clients directs et indirects) sont effectivement stable, plus l'application est stable dans son ensemble

Stabilité et abstraction

- Plus un module est abstrait, plus il est facile de le garder stable
- Pour fabriquer un paquetage stable, il faut identifier les bonnes abstractions
  - ne rien sous-entendre sur l'implémentation
  - *plus c'est abstrait, plus c'est stable*

Stabilité

- Identifier et concevoir les paquetages qui doivent être stables est une tâche importante
- Penser que l'on peut agrandir son paquetage avec de nouvelles interfaces sans trop de danger

- *Un package ne doit dépendre que de packages plus stables que lui.*

Dépendances acycliques

- Un module doit toujours dépendre de modules plus stables que lui
- En respectant cette règle, **le graphe de dépendances ne doit pas être cyclique**
- En cas de cycle, il faut faire de *refactoring* pour éliminer le cycle

Livraison

- Un module livré est un paquet fourni aux clients
- Livraison = publication = mise-en-forme
- livré : numéro de version
  release.version.revision
- Seulement une fois livré, c'est réutilisable

Livraison / Configuration

- Penser à spécifier dans votre paquetage livré la configuration cible
  - version de logiciels et bibliothèques utilisés
  - configuration hardware
  - Les contraintes de configuration

Livraison => Réutilisation

- **On ne peut réutiliser qu'un paquetage livré**
- On réutilise un paquetage complet et non une classe
- Le code est la propriété de l'auteur, c'est lui qui corrige les bugs
- *« Je ne veux pas le savoir du moment que ça marche, je ne cherche pas à savoir pourquoi »*
  - … sauf au coin café, après avoir épuisé les sujets de discussions usuels

Cycle de développement

- Un cycle est
  - analyse
    - conception
      - Ecriture du Code
    - test (unitaires,...)
  - Livraison
- Plus un problème arrive tard, plus le changement est important
- Il faut des cycles courts
  - Mais ...

ne veut pas forcément dire graver le CD !

Comment ré-ouvrir ?

- Deux approches pour les modules O&F
- Interface (blackbox)
  - On ferme l'interface pas l'implémentation
- Héritage/Extension (whitebox)
  - On crée un nouveau module qui ne fait qu'étendre le module précédent

POO : principes de base

Rappel : programmation objet

- champs = état / comportement = méthodes
- Encapsulation
- Héritage
- Polymorphisme

Bonnes pratiques que vous connaissez déjà !

- Une responsabilité => un objet
- I/A/C sans abus

POO : principes de base

- Séparer ce qui varie de ce qui ne varie pas !
  - Changement plus localisés
  - Moins de risques
  - Dépendances limitées
  - Moins de risques d'ouverture

- Plus de flexibilité
- Moins de risques

POO : principes de base

- Encapsulation : la base
  - Séparer intérieur / extérieur
  - Responsabiliser
  - Contractualiser
  - → On ne se préoccupe pas de l'implémentation de ce qu'on utilise

- Encapsulation = je ne sais pas *COMMENT* tu fais
  - Je ne peux pas coder une logique qui dépend de comment ça marche !
  - Cercle vertueux de solidité

POO : principes de base
Principe de substitution de Liskov *(LSP)*

- « Si une propriété P est vraie pour une instance x d'un type T, alors cette propriété P doit rester vraie pour tout instance y d'un sous-type de T »
- Implications :
  - Le contrat défini par la classe de base (pour chacune de ses méthodes) doit être respecté par les classe dérivées
  - L'appelant n'a pas à connaître le type exact de la classe qu'il manipule : n'importe quelle classe dérivée peut être *substituée* à la classe qu'il utilise
- → Principe de base du polymorphisme :
  - Si on substitue une classe par une autre dérivée de la même hiérarchie: comportement (bien sûr) différent mais *conforme*.

POO : principes de base

- « Program to interface, not implementation »
  - Je ne sais pas *QUI* tu es
  - Je ne peux pas coder une logique qui dépend « qui fait quoi » !
  - Cercle vertueux « d'ignorance » / de généricité

POO : principes de base

- Préférer la composition à l'héritage
  - L'héritage a été mis en avant pour la réutilisation
    - Trop !
  - Souvent, l'héritage est rigide et la composition est souple
  - Flexibilité

POO : principes de base

- OCP (Open-Closed Principles)
  - Fermeture: Le code a été écrit, testé, debuggé → on n'y touche plus
  - Ouverture: Le code est prévu pour être extensible

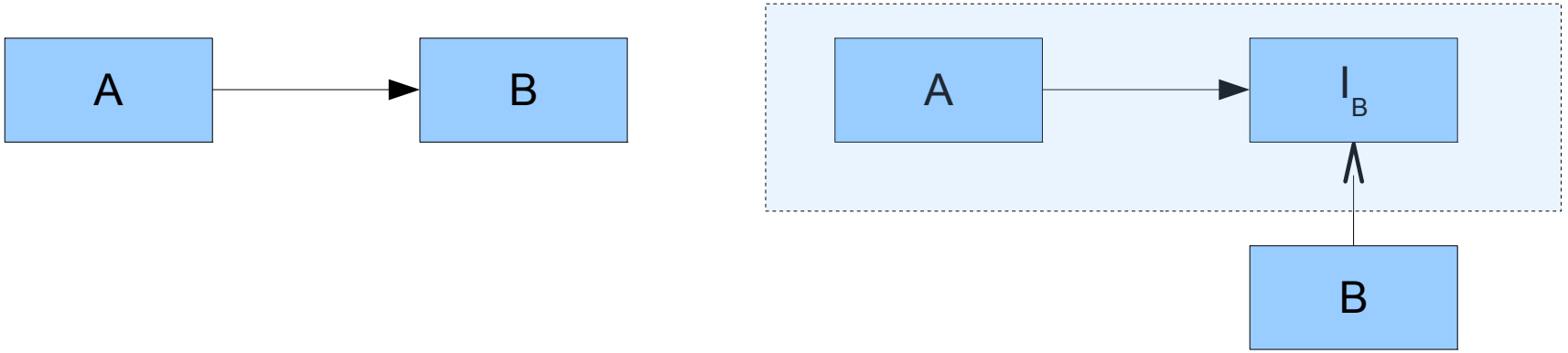
POO : principes de base

- Inversion de dépendance
  1. Réduire les dépendances sur les classes concrètes
  2. Program to interface, not implementation »
  3. → Ne dépendre QUE sur des abstractions, y compris pour les classes de haut niveau
  4. Les abstractions ne doivent pas dépendre de détails.
     → Les détails doivent dépendre d'abstractions.

  - Extensibilité

POO : principes de base

- Inversion de dépendance : comment ?



POO : principes de base

- Des règles apparemment strictes ...
  - Des interfaces cohérentes et pas trop grosses
    - Sinon, on découpe
  - Des objets qui interagissent avec peu d'objets
    - Sinon, il manque des « chefs » ou des intermédiaires
  - Ne pas parler directement aux amis de ses amis !
    - On fait transmettre les messages

Mais à appliquer avec pragmatisme !

L'essentiel

- Les difficultés
  - Supporter le changement
  - Maintenance
  - Gestion des dépendances

- Art du design
  - Prendre soin des Interfaces !
  - Savoir marier
    - Héritage (IS-A) et
    - Composition (HAS-A)
  - Open-Close Principle
  - Savoir se servir de l'Inversion de dépendance