PLD Agile

Christine Solnon

INSA de Lyon - 4IF - 2016 / 2017

Référentiel des compétences

Mettre en oeuvre une méthodologie pour développer des logiciels de qualité

- Mettre en oeuvre un processus de développement logiciel itératif
- Mettre en œuvre les principes du manifeste Agile
- Mettre en œuvre une démarche qualité

Concevoir l'architecture d'un logiciel orienté objet

- Structurer un logiciel en classes faiblement couplées et fortement cohésives
- Utiliser des Design Patterns
- Mettre en œuvre à bon escient les mécanismes offerts par les langages de programmation orientés objet : héritage, généricité, délégation, ...

Utiliser des diagrammes UML pour modéliser un objet d'étude

- Interpréter un diagramme UML donné
- Concevoir un diagramme UML modélisant un objet d'étude
- Vérifier la cohérence de différents diagrammes modélisant un même objet

Organisation

Cours (8 séances)

- Partie 1 : Développement logiciel itératif et agile
  (2 séances - C. Solnon)
- Partie 2 : Principes et patrons de conception orientée objet
  (2 séances - C. Solnon)
- Partie 3 : Retours d'expériences Agiles à Esker
  (1 séance - S. Gosselin et S. Sorlin)
- Partie 4 : Qualité logicielle (2 séances - P.-E. Portier)
- Partie 5 : Présentation du PLD (1 séance - P.-E. Portier et C. Solnon)

Projet Longue Durée (8 séances de 4 heures)

Planification de tournées de livraisons (inspiré de *Optimod'Lyon*)

Evaluation

- DS + Note de projet

Quelques livres à emprunter à DOC'INSA

- Le processus unifié de développement logiciel
  Ivar Jacobson, Grady Booch, James Rumbaugh
  $\rightsquigarrow$ *Description d'un processus de développement itératif*
- UML 2 et les design patterns
  Craig Larman
  $\rightsquigarrow$ *Mise en œuvre d'un processus de dév. itératif dans un esprit Agile*
  $\rightsquigarrow$ *Bonne introduction aux design patterns*
- Modélisation Objet avec UML
  Pierre-Alain Muller, Nathalie Gaertner
  $\rightsquigarrow$ *Bon rappel sur UML pour l'analyse et la conception OO*
- Tête la première : Design Patterns
  Eric Freeman & Elizabeth Freeman
  $\rightsquigarrow$ *Introduction aux design patterns, avec de nombreux exemples*
- ...et plein d'autres !

Plan de la partie 1

Développement logiciel itératif et agile

Deux citations pour se motiver

Philippe Kruchten :

Programming is fun, but developing quality software is hard. In between the nice ideas, the requirements or the "vision", and a working software product, there is much more than programming. Analysis and design, defining how to solve the problem, what to program, capturing this design in ways that are easy to communicate, to review, to implement, and to evolve is what... (you will learn in this course.)

Craig Larman :

The proverb "owning a hammer doesn't make one an architect" is especially true with respect to object technology. Knowing an object-oriented language (such as Java) is a necessary but insufficient first step to create object systems. Knowing how to "think in objects" is also critical.

Crise du logiciel ?

1979 : Etude du *Government Accounting Office* sur 163 projets

- 29% des logiciels n'ont jamais été livrés
- 45% des logiciels ont été livrés... mais n'ont pas été utilisés
- 19% des logiciels ont été livrés mais ont du être modifiés
- ... ce qui laisse 7% de logiciels livrés et utilisés en l'état

1994 : Système de convoyage des bagages / aéroport de Denver

- 193 M$ et 16 mois de retard (1,1 M$ perdus par jour de retard)
- Remplacé par un système manuel en 2005

1999 $\rightsquigarrow$ 2011 : New York City Automated Payroll (NYCAP) System

- Budget estimé = 66 M$ $\rightsquigarrow$ Budget réel > 360 M$

Et quelques bugs qui ont couté très cher...

- 1996 : Explosion d'Ariane 5
- 1999 : Perte de NASA Mars Climate Orbiter

Etude du Standish Group (1/3)

Réussite des projets informatiques (sur 50 000 projets par an)

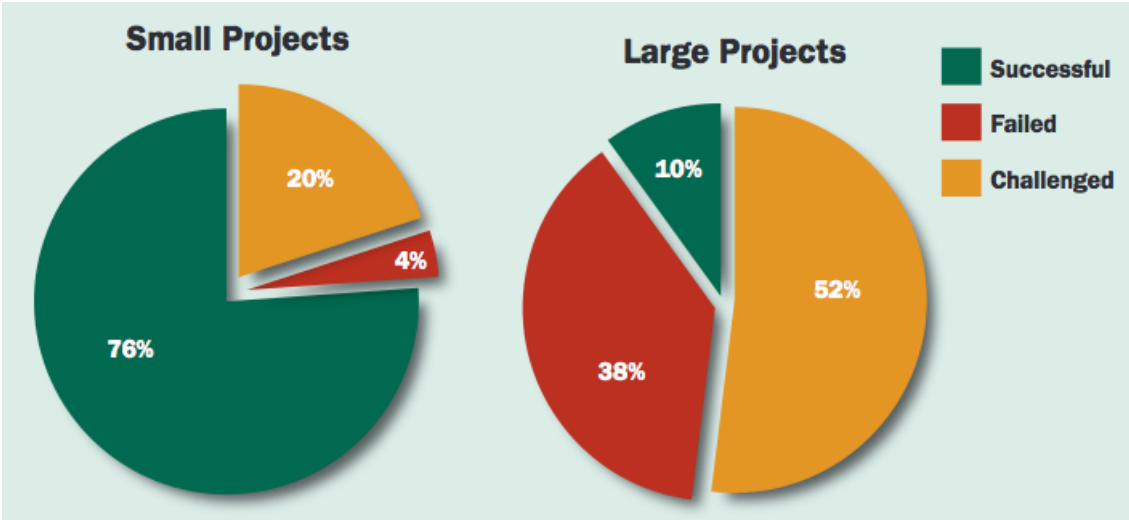
	2011	2012	2013	2014	2015
Succès	29%	27%	31%	28%	29%
Mitigé	49%	56%	50%	55%	52%
Echec	22%	17%	19%	17%	19%

- Succès : livré à temps, sans surcoût et client satisfait
- Mitigé : client non satisfait ou dépassement du coût ou des délais
- Echec : abandonné en cours de route

$\rightsquigarrow$ **Il reste de la marge pour s'améliorer !**

Etude du Standish Group (2/3)

Influence de la taille du projet sur la réussite (en 2012)



- Petit projet : moins de 1 million de dollars
- Gros projet : plus de 10 millions de dollars

Conclusion du Standish group :

It is critical to break down large projects into a sequence of smaller ones, prioritized on direct business value, and install stable, full-time, cross-functional teams that execute these projects following a disciplined agile and optimization approach.

Etude du Standish Group (3/3)

Principales causes des échecs

1. Manque d'implication de l'utilisateur 12,8%
2. Exigences et spécifications incomplètes 12,3%
3. Changement des exigences et spécifications 11,8%

Extrait des conclusions de l'étude :

"... smaller time frames, with delivery of software components early and often, will increase the success rate. Shorter time frames result in an iterative process ... known as growing software as opposed to the old concept of developping software. Growing software engages the user earlier."

	Gros projet		Moyen projet		Petit projet	
	Agile	Cascade	Agile	Cascade	Agile	Cascade
Succès	18%	3%	27%	7%	58%	44%
Mitigé	59%	55%	62%	68%	38%	45%
Echec	23%	42%	11%	25%	4%	11%

Plan de la partie 1

Développement logiciel itératif et agile

Qu'est-ce qu'un logiciel ?

Ensemble d'artefacts

- Codes : Sources, Binaires, Tests, ...
- Documentation pour l'utilisateur : Manuel utilisateur, manuel de référence, tutoriels, ...
- Documentation interne : Cas d'utilisation, Modèle du domaine, Diagrammes d'interaction, Diagrammes de classes, ...
- ...

Conçus par et pour différents acteurs

- Utilisateurs
- Programmeurs
- Hotline
- ...

Qu'est-ce qu'un bon logiciel ?

Différents points de vue

- L'utilisateur — Ce que ça fait ?
  - ⇝ besoins fonctionnels ou non fonctionnels
  - ⇝ besoins exprimés ou implicites, présents ou futurs, ...
- Le programmeur — Comment ça le fait ?
  - ⇝ architecture, structuration, documentation, ...
- Le fournisseur — Combien ça coûte/rapporte ?
  - ⇝ coûts de développement + maintenance, délais, succès ...
- La hotline — Pourquoi ça ne le fait pas/plus ?
  - ⇝ diagnostic, reproductibilité du pb, administration à distance, ...
- ...

Activités d'un processus logiciel (rappel de 3IF) :

- Capture des besoins
  $\rightsquigarrow$ Cahier des charges
- Analyse
  $\rightsquigarrow$ Spécifications fonctionnelles et non fonctionnelles du logiciel
- Conception
  $\rightsquigarrow$ Architecture du logiciel, modèles de conception
- Réalisation / Implantation
  $\rightsquigarrow$ Code
- Validation, intégration et déploiement
  $\rightsquigarrow$ Logiciel livrable
- Maintenance

Enchaînements d'activités et Cycle de vie (rappel de 3IF)

Modèles linéaires

- Cycle en cascade
- Cycle en V

Modèle incrémental

- 3 premières activités exécutées en séquence
  $\rightsquigarrow$ Spécification et architecture figées
- Réalisation, intégration et tests effectués incrémentalement

Problème :

Ces modèles supposent que

- l'analyse est capable de spécifier correctement les besoins
- ces besoins sont stables

Or, 90% des dépenses concernent la maintenance et l'évolution !

Maintenance et évolution

Utilisation des fonctionnalités spécifiées / cycle en cascade [C. Larman] :

- Jamais 45%
- Rarement 19%
- Parfois 16%
- Souvent 13%
- Toujours 7%

Répartition des coûts de maintenance [C. Larman] :

- Extensions utilisateur 41,8%
- Correction d'erreurs 21,4%
- Modification format de données 17,4%
- Modification de matériel 6,2%
- Documentation 5,5%
- Efficacité 4%

If you think writing software is difficult, try re-writting software

Bertrand Meyer

Plan de la partie 1

Développement logiciel itératif et agile

Quelques dates du développement itératif et agile

- 1991 : James Martin $\rightsquigarrow$ RAD (*Rapid Application Development*)
- 1994 : Ivar Jacobson, Grady Booch et James Rumbaugh
  $\rightsquigarrow$ UP (*Unified Software Development Process*)
- 1995 : Jeff Sutherland et Ken Schwaber $\rightsquigarrow$ SCRUM
- 1996 : Kent Beck $\rightsquigarrow$ XP (*eXtreme Programming*)
- ...
- 2001 : K. Beck, A. Cockburn, M. Fowler, J. Sutherland, K. Schwaber, ...
  et 12 autres développeurs réunis dans une station de ski de l'Utah
  $\rightsquigarrow$ *Agile Software Development Manifesto*
- ...
- 2016 : Tout le monde parle d'agilité (même les non informaticiens)...
  mais combien d'équipes de dév. logiciel sont réellement agiles ?

Rapport annuel de VersionOne en 2015

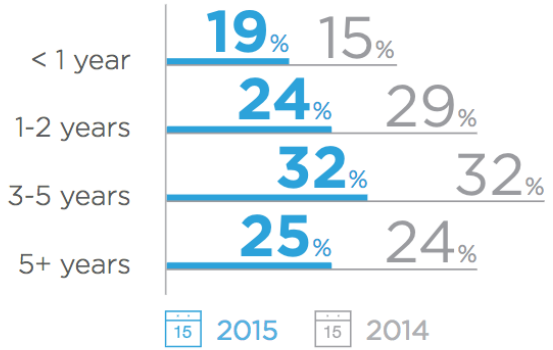
HOW MANY?

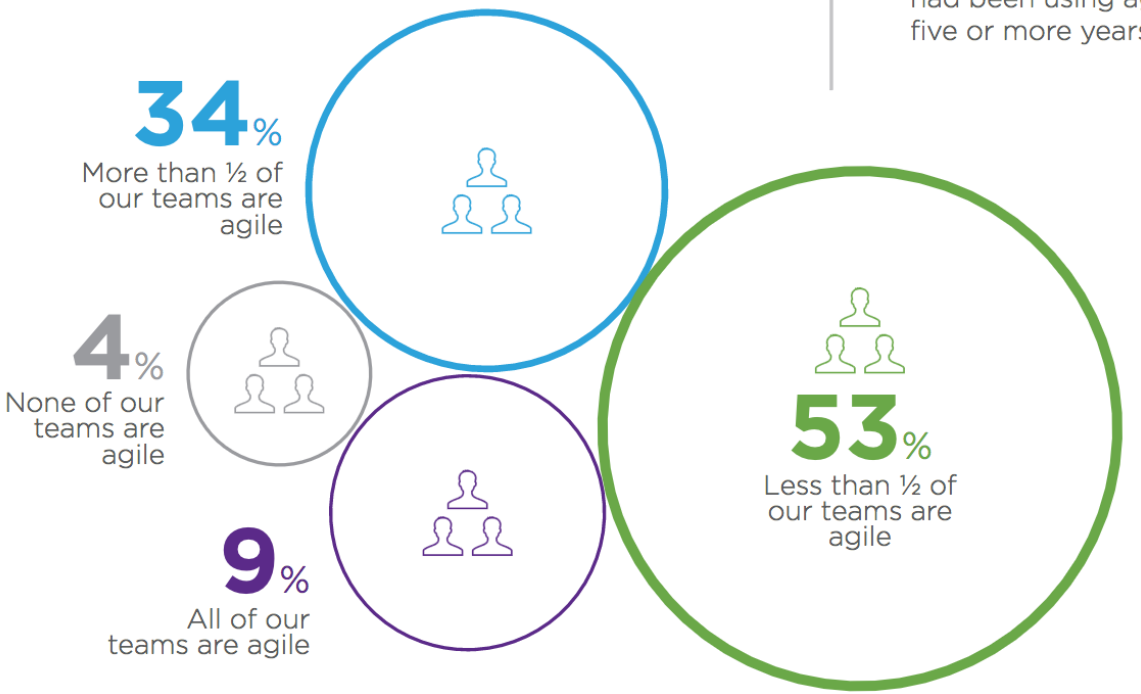
2015 **95%**
The percentage of respondents' organizations that practice agile

&

HOW LONG?

While 33% of respondents said they had 5+ years of agile experience, they reported that the organizations within which they worked had less experience – only 25% of organizations had been using agile for five or more years.





Percentage of Teams Using Agile

A total of 43% of respondents worked in development organizations where the majority of their teams are agile. Only 4% of respondents work in a completely traditional/ non-agile development organization. Contrast this with the 2009 report, in which (31%) of the respondents worked where there were two teams or less practicing agile!

Rapport établi à partir de 3880 réponses à un questionnaire en ligne sur le site de VersionOne ⇝ Biais probable

Agile Software Development Manifesto

www.agilealliance.com

Nous découvrons comment mieux développer des logiciels par la pratique et en aidant les autres à le faire. Ces expériences nous ont amenés à valoriser :

- **Les individus et leurs interactions**
  ... plus que les processus et les outils
- **Des logiciels opérationnels**
  ... plus qu'une documentation exhaustive
- **La collaboration avec les clients**
  ... plus que la négociation contractuelle
- **L'adaptation au changement**
  ... plus que le suivi d'un plan

Nous reconnaissons la valeur des seconds éléments...
...mais privilégions les premiers.

Quelques principes (choisis) du manifeste Agile

- Satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée.
- Accueillir positivement les changements de besoins, même tard.
- Livrer fréquemment un logiciel opérationnel avec des cycles de quelques semaines à quelques mois.
- Faire travailler ensemble utilisateurs et développeurs tout au long du projet.
- Un logiciel opérationnel est la principale mesure d'avancement.
- La simplicité (l'art de minimiser le travail inutile) est essentielle.
- À intervalles réguliers, l'équipe réfléchit aux moyens de devenir plus efficace, puis règle et modifie son comportement en conséquence.

Attention : "Etre agile" ne signifie pas "ne pas modéliser"
$\rightsquigarrow$ Modéliser permet de comprendre, de communiquer et d'explorer

Plan de la partie 1

Développement logiciel itératif et agile

Vue globale de la vie d'un logiciel avec UP

La vie d'un logiciel est composée de cycles

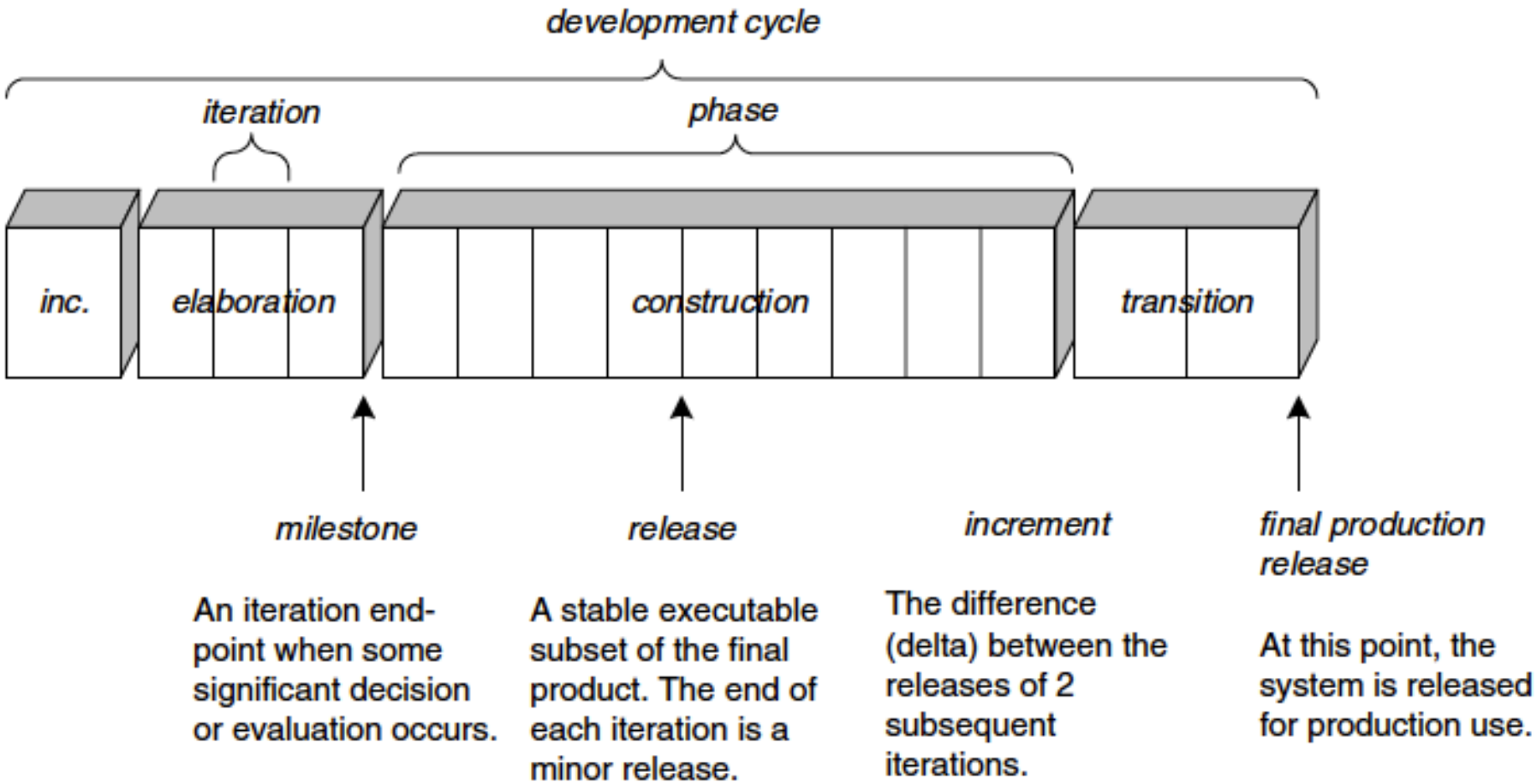
- 1 cycle $\Rightarrow$ 1 nouvelle version du logiciel
- Chaque cycle est composé de 4 phases :
  - Etude préliminaire (*Inception*)
  - Elaboration
  - Construction
  - Transition

Chaque phase d'un cycle est composée d'itérations

- 1 itération $\Rightarrow$ 1 incrément
- Chaque itération est une mini cascade d'activités
  - Capture des besoins
  - Analyse
  - Conception
  - Réalisation
  - Test et intégration

...en proportions variables en fonction du temps

Vue graphique d'un cycle avec UP



[figure extraite du livre de C. Larman]

Plan de la partie 1

Développement logiciel itératif et agile

Phase 1 : Etude préliminaire (*inception*)

- Phase très courte (souvent une seule itération)
- Etape préliminaire à l'élaboration

  $\rightsquigarrow$ Déterminer la faisabilité, les risques et le périmètre du projet

  - Que doit faire le système ?
  - A quoi pourrait ressembler l'architecture ?
  - Quels sont les risques ?
  - Estimation approximative des coûts et des délais

$\rightsquigarrow$ Accepter le projet ?

Phase 2 : Elaboration

Quelques itérations courtes et de durée fixe, pilotées par les risques

- Identification et stabilisation de la plupart des besoins
  $\rightsquigarrow$ Spécification de la plupart des cas d'utilisation
- Conception de l'architecture de base
  $\rightsquigarrow$ Squelette du système à réaliser
- Programmation et test des éléments d'architecture les + importants
  $\rightsquigarrow$ Réalisation des cas d'utilisation critiques ($<$10% des besoins)
  $\rightsquigarrow$ Tester au plus tôt, souvent et de manière réaliste
- Estimation fiable du calendrier et des coûts

$\rightsquigarrow$ Besoins et architecture stables ? Risques contrôlés ?

Phase 3 : Construction

Phase la plus coûteuse ($>$50% du cycle)

- Développement par incréments
  $\rightsquigarrow$ Architecture stable malgré des changements mineurs
- Le produit contient tout ce qui avait été planifié
  $\rightsquigarrow$ Il reste quelques erreurs

$\rightsquigarrow$ Produit suffisamment correct pour être installé ?

Phase 4 : Transition

- Produit délivré (version béta)
- Correction du reliquat d'erreurs
- Essai et amélioration du produit, formation des utilisateurs, installation de l'assistance en ligne...

$\rightsquigarrow$ Tests suffisants ? Produit satisfaisant ? Manuels prêts ?

Plan de la partie 1

Développement logiciel itératif et agile

Processus guidé par les cas d'utilisation

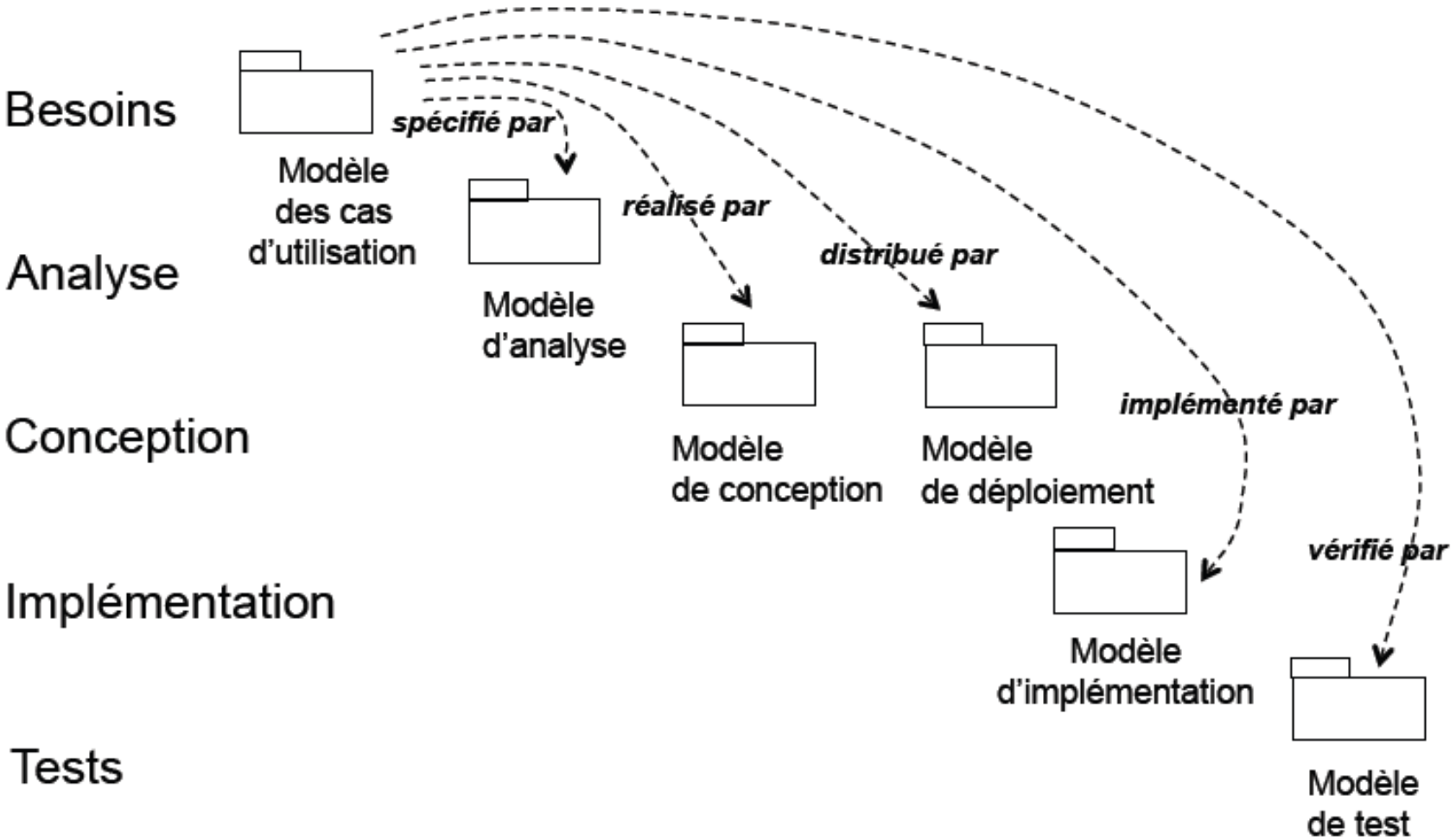
Les cas d'utilisation :

- Décrivent les interactions entre les acteurs et le système
  $\rightsquigarrow$ Scénarios d'utilisation compréhensibles par tous
- Permettent de capturer les vrais besoins des utilisateurs
  $\rightsquigarrow$ Réfléchir en termes d'acteurs et de buts plus que de fonctionnalités
- Guident tout le processus de développement
  $\rightsquigarrow$ Garantissent la cohérence du processus
- Favorisent un développement itératif
  $\rightsquigarrow$ Chaque itération est centrée sur un sous-ensemble de cas

Méthodes Agiles

Cas d'utilisation $\rightsquigarrow$ User Story

Cas d'utilisation / activités



[figure extraite du livre de I. Jacobson, G. Booch, J. Rumbaugh]

Processus itératif et incrémental

Itération = mini projet donnant lieu à un incrément

Avantages :

- Gestion des risques importants lors des premières itérations
  $\rightsquigarrow$ Construire et stabiliser le noyau architectural rapidement
- Feed-back régulier des utilisateurs
  $\rightsquigarrow$ Adaptation permanente du système aux besoins réels
- Feed-back régulier des développeurs et des tests
  $\rightsquigarrow$ Affiner la conception et les modèles
- Complexité mieux gérée
  $\rightsquigarrow$ Etapes plus courtes et moins complexes
- Exploitation des erreurs des itérations précédentes
  $\rightsquigarrow$ Amélioration du processus d'une itération sur l'autre

Méthodes Agiles

Itération $\rightsquigarrow$ Sprint

Plan du cours

Vue globale d'une itération "générique"

Mini cascade qui enchaîne différentes activités :

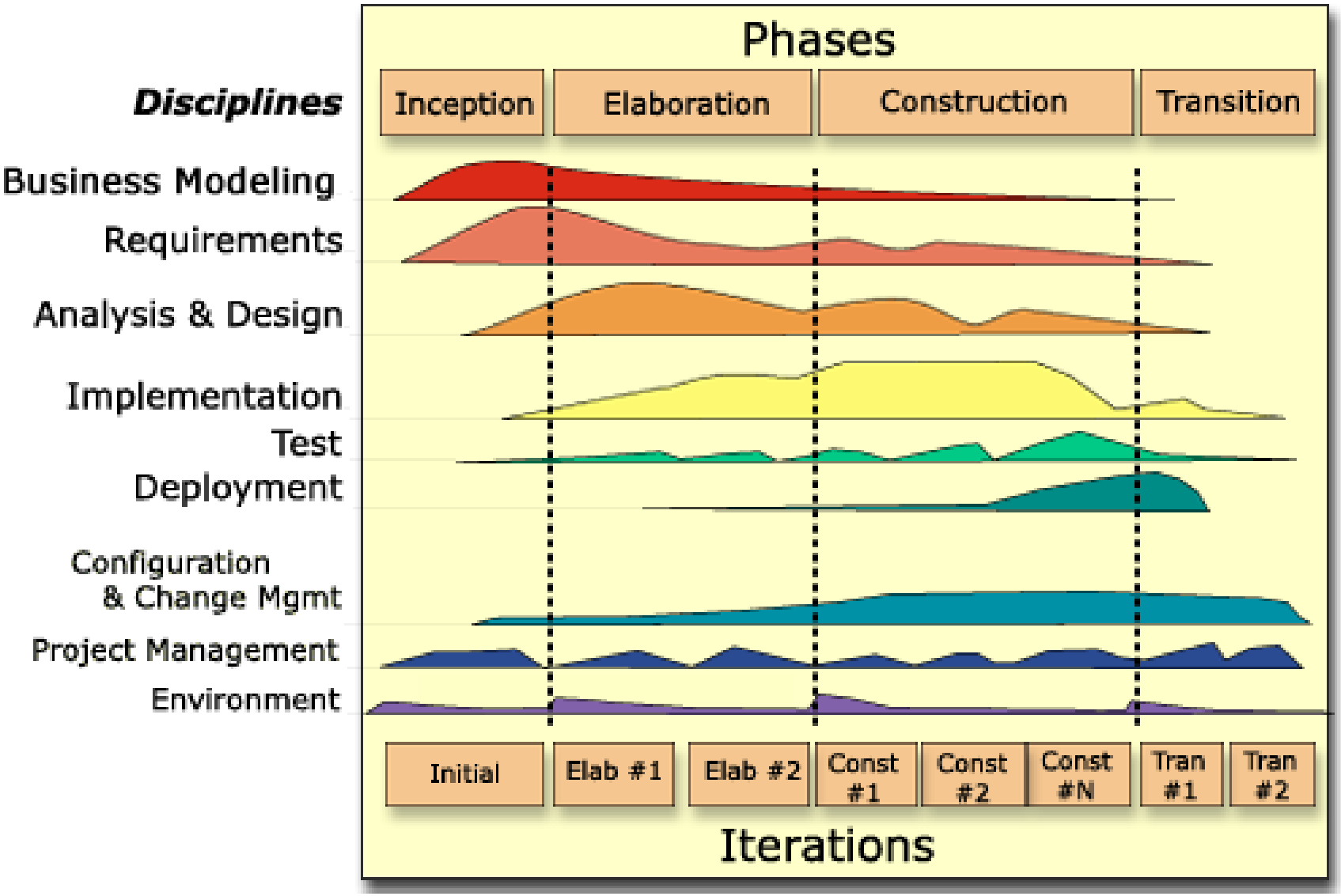
- Capture et analyse des besoins
  ⇝ Modéliser le système vu de l'extérieur
- Conception
  ⇝ Modéliser le système vu de l'intérieur
- Réalisation et tests
  ⇝ Implémenter le système
  ⇝ Valider l'implémentation par rapport aux besoins

⇝ La part de ces activités varie en fonction des phases

Itérations Agiles (sprints) :

- Itérations de courte durée
  ⇝ Quelques semaines (typiquement entre 2 et 4)
- Itérations de durée fixée
  ⇝ Réduire les objectifs de l'itération si nécessaire
  ⇝ Reporter une partie des objectifs aux itérations suivantes

Part des activités d'une itération / Phases



[Figure extraite de http ://www.ibm.com/developerworks/rational]

Plan de la partie 1

Développement logiciel itératif et agile

Activité "Capture et analyse des besoins"

Objectif de l'activité : se mettre d'accord sur le système à construire

- Besoins fonctionnels ($\rightsquigarrow$ comportementaux)
- Besoins non fonctionnels ($\rightsquigarrow$ tous les autres)

Capture vs analyse des besoins

- Capture :
  - Langage du client $\rightsquigarrow$ Informel, redondant
  - Structuration par les cas d'utilisation
- Analyse :
  - Langage du développeur $\rightsquigarrow$ Formel, non redondant
  - Structuration par les classes

Activité difficile car :

- Les utilisateurs ne connaissent pas vraiment leurs besoins
- Les développeurs connaissent mal le domaine de l'application
- Utilisateurs et développeurs ont des langages différents
- Les besoins évoluent
- Il faut trouver un compromis entre services, coûts et délais

Buts et Artefacts

Buts

- Comprendre le contexte du système
- Appréhender les besoins fonctionnels
- Appréhender les besoins non fonctionnels et architecturaux

Artefacts / Livrables

- Modèles du domaine et du métier
- Glossaire
- Modèle des cas d'utilisation
- Exigences supplémentaires

Modèle du domaine

Qu'est-ce qu'un modèle du domaine ?

Diagramme de classes conceptuelles (objets du monde réel)
$\rightsquigarrow$ Peu d'attributs, pas d'opération, pas de classes logicielles

Comment construire un modèle du domaine ?

- Réutiliser (et modifier) des modèles existants !
- Utiliser une liste de catégories :
  - Classes : *Transactions métier, Lignes de transactions, Produits/services liés aux transactions, Acteurs, Lieux, ...*
  - Associations : *est-une-description-de*, *est-membre-de*, ...
- Identifier les noms et groupes nominaux des descriptions textuelles

Modélisation itérative et agile

Objectifs : Comprendre les concepts clés et leurs relations... et communiquer !

- Il n'existe pas de modèle du domaine exhaustif et correct
- Le modèle du domaine évolue sur plusieurs itérations
- Travailler en mode "esquisse"

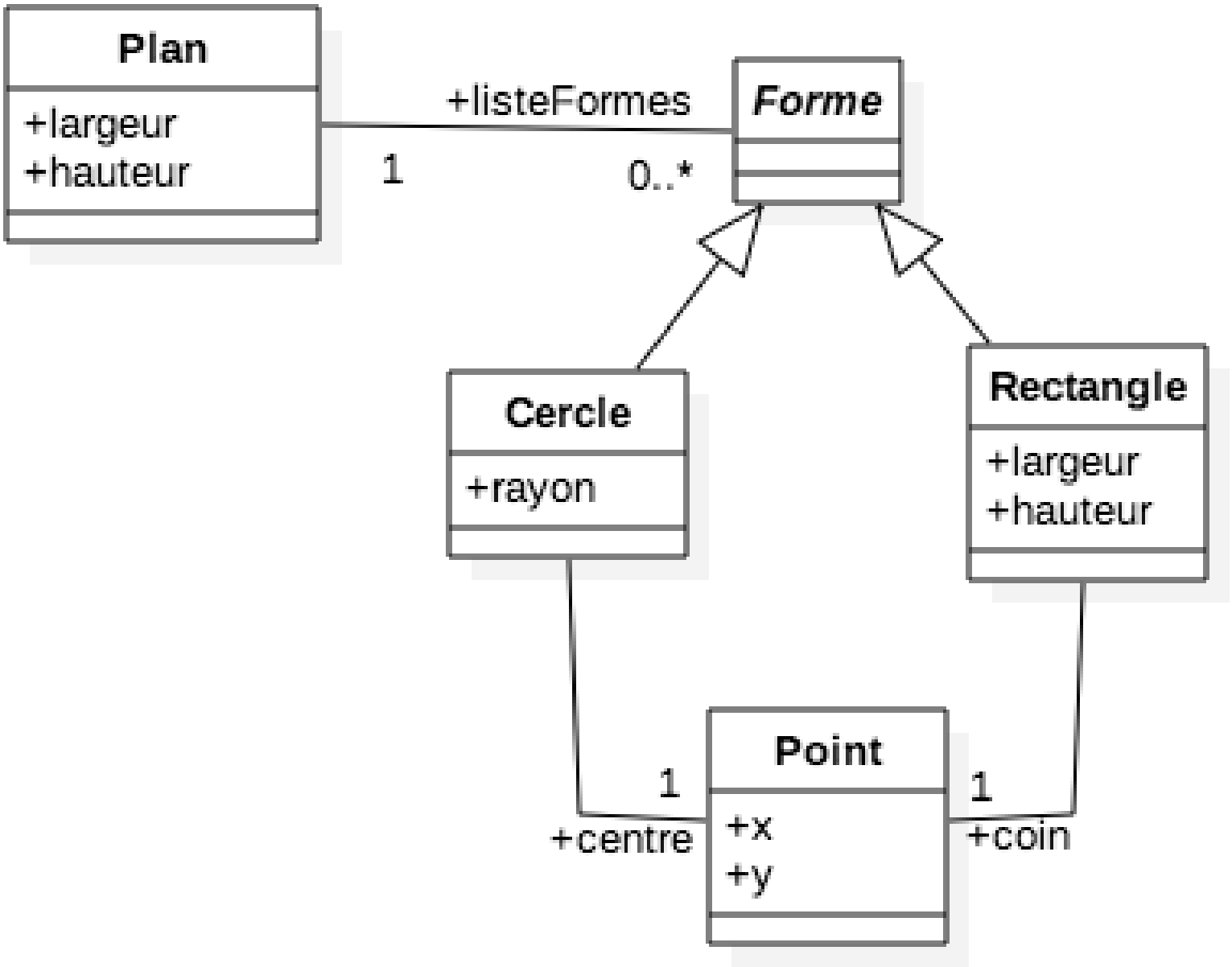
Exercice : Modèle du domaine de PlaCo

Une scierie, équipée d'une machine de découpe de planches au laser, veut un système pour dessiner les plans à transmettre à la machine.

- Chaque plan a une hauteur et une largeur.
- Les formes à positionner sur le plan sont des rectangles et des cercles :
  - un rectangle est défini par les coordonnées de son coin supérieur gauche, sa largeur et sa hauteur ;
  - un cercle est défini par son rayon et les coordonnées de son centre.

  Les coordonnées et les longueurs sont des valeurs entières exprimées dans une unité donnée. Les formes ne doivent pas se superposer.
- L'application doit permettre d'ajouter, supprimer et déplacer des formes sur un plan, de sauvegarder et charger des plans, et de transmettre un plan au système de découpe.

Modèle du domaine de PlaCo



Autres artefacts pour "Comprendre le contexte du système"

Modèle d'objets métier (Business Object Model)

- Modèle plus général que le modèle du domaine

  Abstraction de la façon dont les travailleurs et les entités métier doivent être mis en relation, et de la façon dont ils doivent collaborer afin d'accomplir une activité

  $\rightsquigarrow$ Diagrammes de classe, d'activité, de collaboration et de séquence
  $\rightsquigarrow$ Plus d'infos dans le cours 4IF-ASI

Glossaire

- Définit les termes les plus importants
  $\rightsquigarrow$ Evite les ambiguités
- Dérivé des cas d'utilisation et modèles du domaine et du métier

Buts et Artefacts

Buts

- Comprendre le contexte du système
- Appréhender les besoins fonctionnels
- Appréhender les besoins non fonctionnels et architecturaux

Artefacts / Livrables

- Modèles du domaine et du métier
- Glossaire
- Modèle des cas d'utilisation
- Exigences supplémentaires

Cas d'utilisation (1/2)

Qu'est-ce qu'un cas d'utilisation ?

- Usage qu'un acteur (entité extérieure) fait du système
  $\rightsquigarrow$ Séquence d'interactions entre le système et les acteurs
- Généralement composé de plusieurs scénarios
  $\rightsquigarrow$ Scénario de base et ses variantes (cas particuliers)

Attention : Système = boite noire

$\rightsquigarrow$ Décrire ce que fait le système, et non comment il le fait

Pourquoi des cas d'utilisation ?

- Procédé simple permettant au client de décrire ses besoins
  - Parvenir à un accord (contrat) entre clients et développeurs
- Point d'entrée pour les étapes suivantes du développement
  - Conception et implémentation $\rightsquigarrow$ Réalisation de cas d'utilisation
  - Tests fonctionnels $\rightsquigarrow$ Scénarios de cas d'utilisation

Cas d'utilisation (2/2)

Comment découvrir les cas d'utilisation ?

- Délimiter le périmètre du système
- Identifier les acteurs principaux
  - ⇝ Ceux qui atteignent un but en utilisant le système
- Identifier les buts de chaque acteur principal
- Définir les cas d'utilisation correspondant à ces buts

⇝ Atelier d'expression des besoins réunissant clients, utilisateurs, analystes, développeurs et architectes

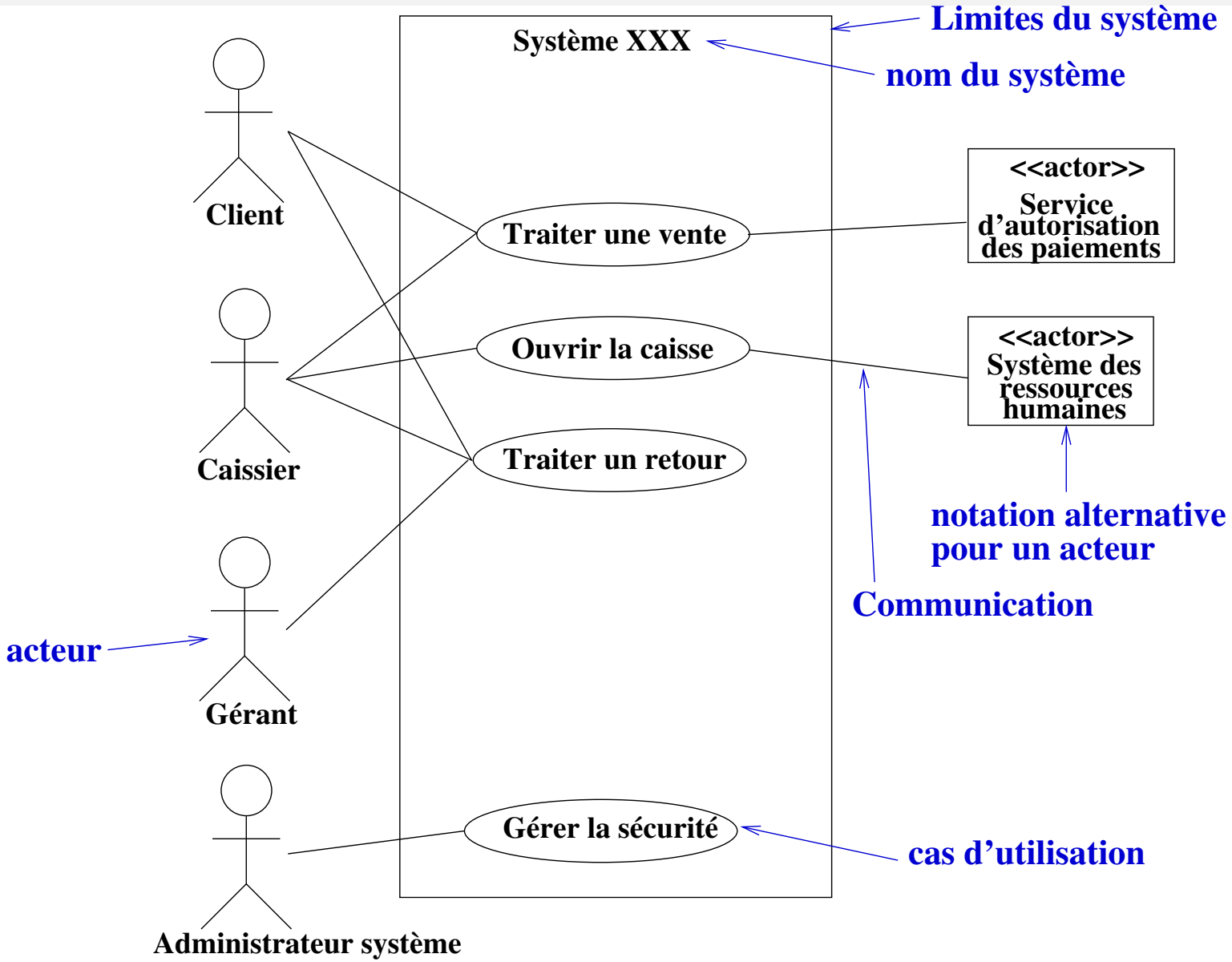
Comment décrire les cas d'utilisation ?

Modèle des cas d'utilisation :

- Diagramme des cas d'utilisation
- Descriptions textuelles des cas d'utilisation
- Diagrammes de séquence système des cas d'utilisation

Diagramme des cas d'utilisation (rappel 3IF)

Relations entre cas d'utilisation et acteurs



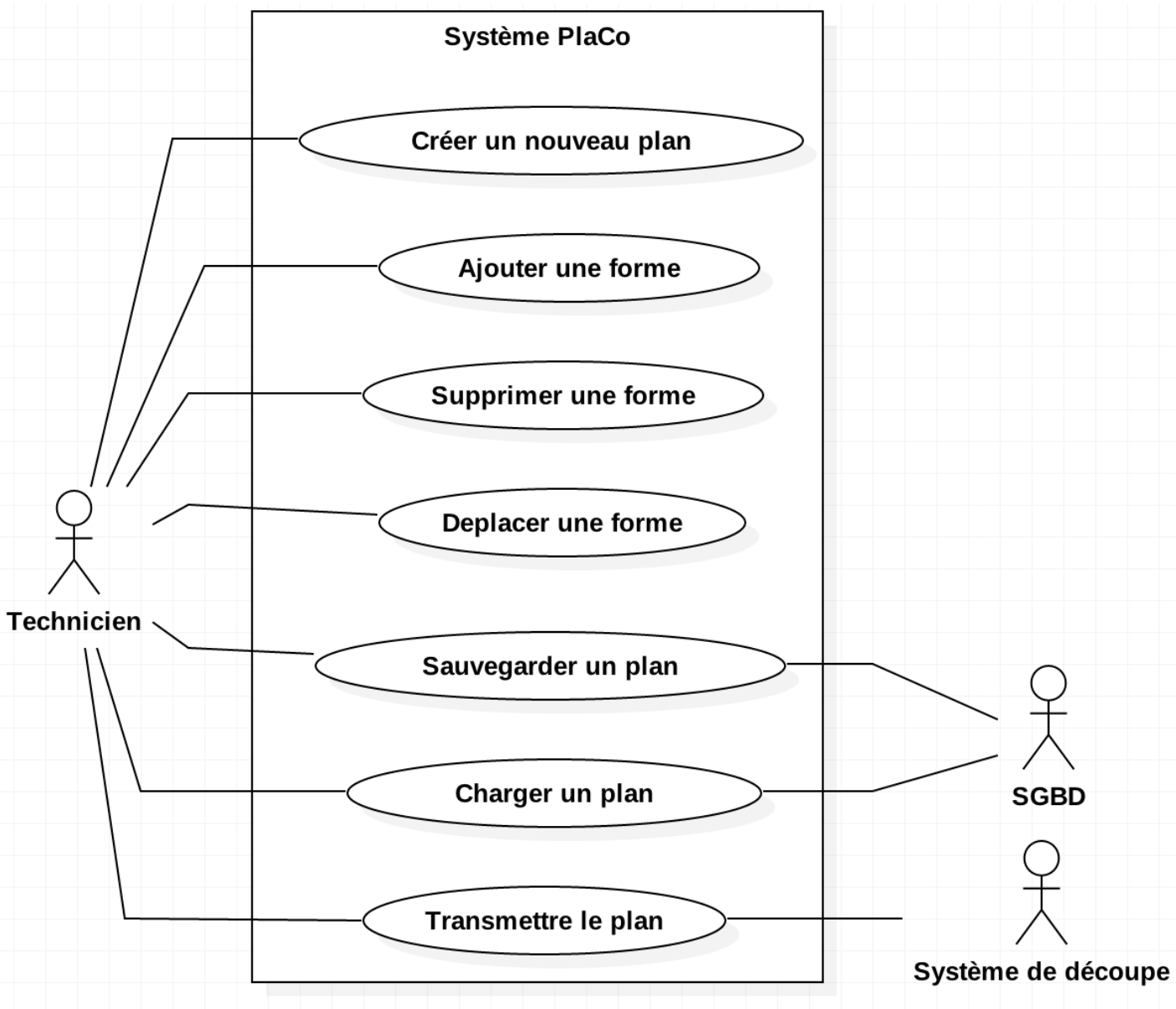
Exercice : Diagramme de cas d'utilisation de PlaCo

Une scierie, équipée d'une machine de découpe de planches au laser, veut un système pour dessiner les plans à transmettre à la machine.

- Chaque plan a une hauteur et une largeur.
- Les formes à positionner sur le plan sont des rectangles et des cercles :
  - un rectangle est défini par les coordonnées de son coin supérieur gauche, sa largeur et sa hauteur ;
  - un cercle est défini par son rayon et les coordonnées de son centre.

  Les coordonnées et les longueurs sont des valeurs entières exprimées dans une unité donnée. Les formes ne doivent pas se superposer.
- L'application doit permettre d'ajouter, supprimer et déplacer des formes sur un plan, de sauvegarder et charger des plans, et de transmettre un plan au système de découpe.

Diagramme de cas d'utilisation de PlaCo



Description textuelle d'un cas d'utilisation

Chaque cas est décrit par un ensemble de scénarios

- Un scénario de base (*main success scenario*)
- Des extensions (une pour chaque cas particulier possible)

Qu'est-ce qu'un scénario ?

Séquence d'interactions entre les acteurs et le système

Description d'un scénario

- Description abrégée : scénario de base décrit en un paragraphe
  $\leadsto$ Histoire d'un acteur qui utilise le système pour atteindre un but
- Description structurée (selon Martin Fowler) :
  - Titre : But que l'acteur principal souhaite atteindre avec ce cas
    $\leadsto$ Verbe à l'infinitif suivi d'un complément d'objet
  - Préconditions : Conditions d'activation du cas (optionnel)
  - Scénario de base : Liste d'interactions entre acteurs et système
  - Extensions : Une liste d'interactions pour chaque cas particulier

Description textuelle de "Ajouter un rectangle" (1/2)

Description abrégée :

Le technicien saisit les coordonnées des deux angles opposés du rectangle. Le rectangle est ajouté dans le plan.

Description structurée :

- Précondition : un plan est chargé
- Scénario principal :
  1. Le système demande de saisir les coordonnées d'un angle du rectangle
  2. Le technicien saisit les coordonnées d'un point $p1$
  3. Le système demande de saisir les coordonnées de l'angle opposé
  4. Le technicien saisit les coordonnées d'un point $p2$
  5. Le système ajoute le rectangle correspondant dans le plan et affiche le plan

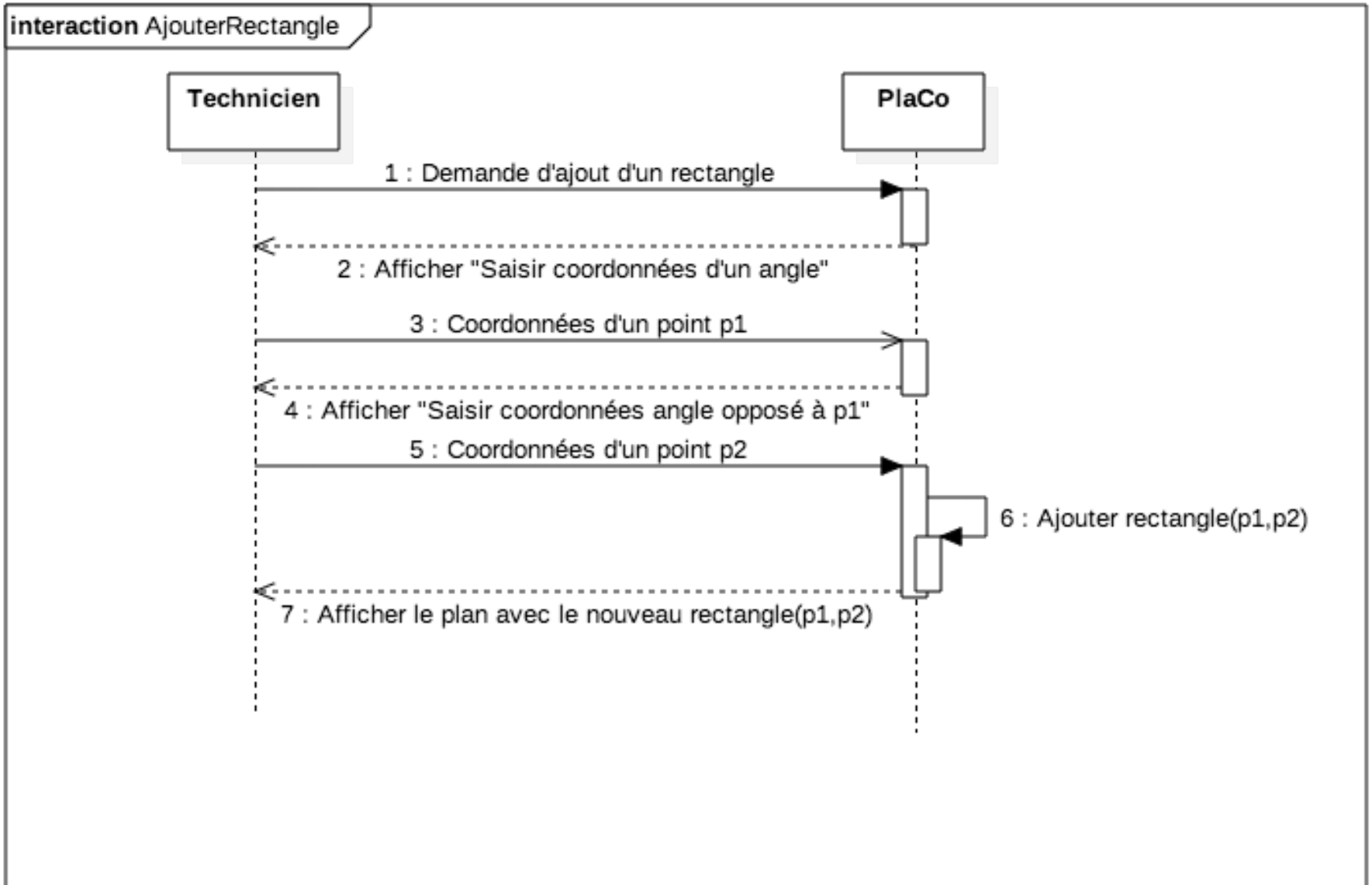
Description textuelle de "Ajouter un rectangle" (2/2)

Description structurée (suite) :

- Alternatives :
  - **2a** Le point $p1$ n'appartient pas au plan
    - Le système indique que $p1$ n'est pas valide et retourne à l'étape 1
  - **2b** Le point $p1$ appartient à une forme du plan
    - Le système indique que $p1$ n'est pas valide et retourne à l'étape 1
  - **4a** Le point $p2$ n'appartient pas au plan
    - Le système indique que $p2$ n'est pas valide et retourne à l'étape 3
  - **4b** Le rectangle défini par $(p1, p2)$ a une intersection non vide avec une forme du plan
    - Le système indique que $p2$ n'est pas valide et retourne à l'étape 3
  - **1-4a** Le technicien indique au système qu'il souhaite annuler la saisie
    - Le système annule la saisie

Diag. de séquence système d'un cas d'utilisation (rappel 3IF)

↝ Représentation graphique d'un scénario d'un cas d'utilisation



Modélisation itérative des cas d'utilisation

Le modèle des cas d'utilisation évolue au fil des phases et des itérations :

- Itération 1 / Phase d'inception :
  - La plupart des cas sont identifiés
  - Environ 10% des cas sont analysés

  $\rightsquigarrow$ Cas les plus significatifs / risqués / importants
- Itération 2 / Phase d'élaboration :
  - Environ 30% des cas sont analysés
  - Conception des cas les plus significatifs / risqués / importants
  - Implémentation de ces cas

  $\rightsquigarrow$ Premier feed-back et première estimation des coûts du projet
- Itérations suivantes de la phase d'élaboration :
  - Analyse détaillée de quelques nouveaux cas
  - Conception et implémentation de nouveaux cas
- Dernière itération de la phase d'élaboration :
  - Quasiment tous les cas sont identifiés
  - de 40 à 80% des cas sont analysés
  - Les cas les plus significatifs / risqués / importants sont implémentés

  $\rightsquigarrow$ Architecture stabilisée et projet planifié

Cas d'utilisation dans les méthodes Agiles ?

User story :

- Courte description d'une utilisation du logiciel
- Potentiellement incomplète ou imprécise

$\rightsquigarrow$ Description informelle d'un cas d'utilisation, mais pas nécessairement en termes d'interactions avec le système

Buts et Artefacts

Buts

- Comprendre le contexte du système
- Appréhender les besoins fonctionnels
- Appréhender les besoins non fonctionnels et architecturaux

Artefacts

- Modèles du domaine et du métier
- Glossaire
- Modèle des cas d'utilisation
- Exigences supplémentaires

But "Appréhender les besoins non fonctionnels"

Exigences supplémentaires :

- Certains besoins non fonctionnels sont déjà dans les cas d'utilisation
  $\rightsquigarrow$ Les regrouper pour éviter les doublons
- Lister également ceux qui ne sont pas dans les cas d'utilisation

Liste pour recenser les besoins : FURPS+

Functionality : Fonctions, capacités, sécurité

Usability : Facteurs humains, aide, documentation

Reliability : Fréquence des pannes, possibilité de récupération

Performance : Temps de réponse, débit, exactitude, ressources

Supportability : Adaptabilité, maintenance, configurabilité

+ : Facteurs complémentaires

- Langages et outils, matériel, etc
- Contraintes d'interfaçage avec des systèmes externes
- Aspects juridiques, licence
- ...

Autres artefacts de la capture et l'analyse des besoins

Vision

- Vue globale synthétique du projet
  $\rightsquigarrow$ Résumé des cas d'utilisation et exigences supplémentaires

Maquette de l'IHM

- Uniquement si IHM complexe
  $\rightsquigarrow$ Validation du client

Tableau des facteurs architecturaux

Récapitulatif des facteurs pouvant influer sur l'architecture :

- Points de variation et d'évolution
- Fiabilité et possibilités de récupération
- Performance
- ...

$\rightsquigarrow$ Evaluer les impacts, la priorité et les difficultés/risques

Plan de la partie 1

Développement logiciel itératif et agile

Conception

Objectif premier de la modélisation pendant la conception :

Comprendre et communiquer :

- Quelles sont les responsabilités des objets ?
- Quelles sont les collaborations entre objets ?
- Quels patterns de conception peut-on utiliser ?

$\rightsquigarrow$ La doc. peut être générée à partir du code (reverse engineering)

Modélisation Agile

- Modéliser en groupe
- Créer plusieurs modèles en parallèle
  $\rightsquigarrow$ Diagrammes dynamiques (interactions)
  $\rightsquigarrow$ Diagrammes statiques (classes, packages et déploiement)
- Concevoir avec les programmeurs et non pour eux !

Buts et Artefacts

Buts

- Appréhender la logique comportementale (interactions entre objets)
- Appréhender la logique structurelle

Artefacts / Livrables

- Diagrammes de séquence
- Diagrammes de classes, de packages et de déploiement

Diagrammes de séquence

↝ **Point de vue temporel sur les interactions**

Pendant la capture des besoins : Système = boite noire

↝ Séquences d'interactions entre acteurs et système

- Décrire les scénarios des cas d'utilisation

Pendant la conception : On ouvre la boite noire

↝ Interactions entre objets du système

- Réfléchir à l'affectation de responsabilités aux objets
  - Qui crée les objets ?
  - Qui permet d'accéder à un objet ?
  - Quel objet reçoit un message provenant de l'IHM ?
  - ...

  de façon à avoir un faible couplage et une forte cohésion
- Elaboration en parallèle avec les diagrammes de classes

  ↝ Contrôler la cohérence des diagrammes !

Buts et Artefacts

Buts

- Appréhender la logique comportementale (interactions entre objets)
- Appréhender la logique structurelle

Artefacts / Livrables

- Diagrammes d'interaction
- Diagrammes de classes, de packages et de déploiement

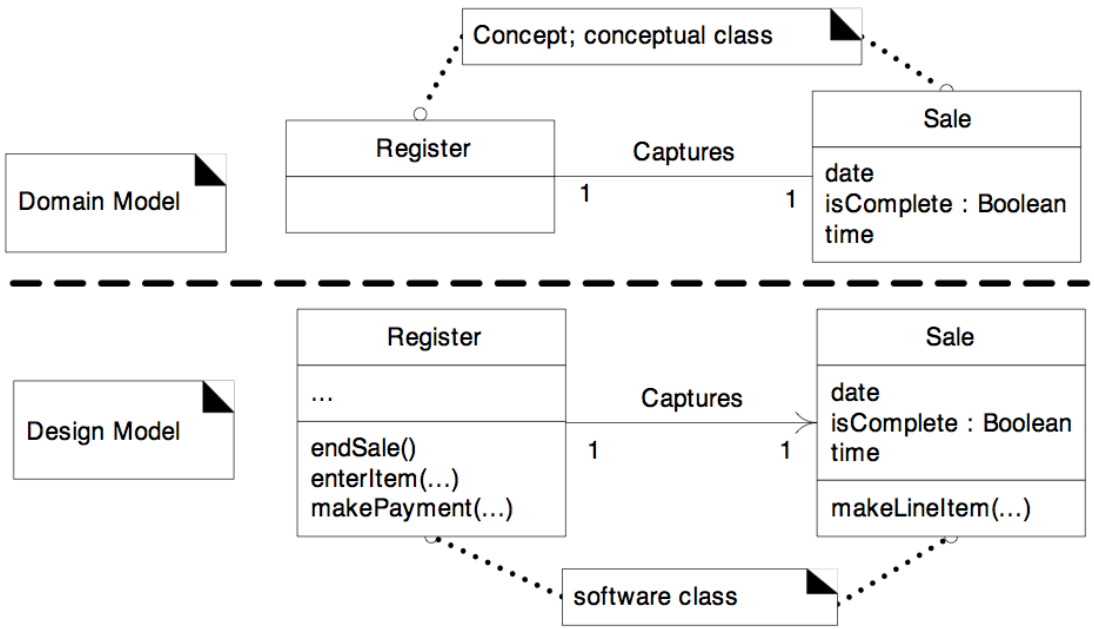
Diagrammes de classes

Pendant la capture des besoins : Modèle du domaine

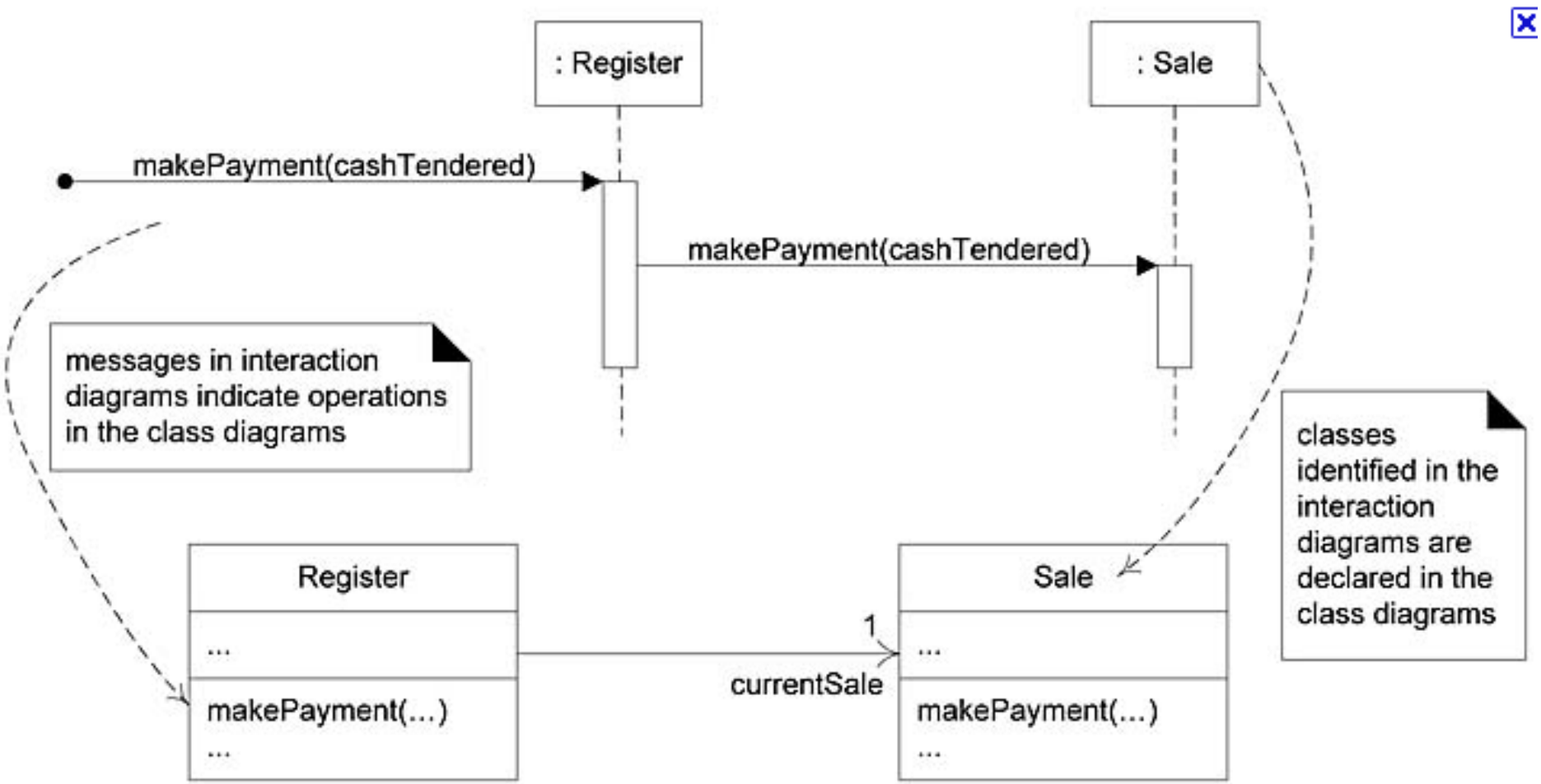
- Classes = objets du mondes réels
- Peu d'attributs, pas de méthodes, pas de visibilité

Pendant la conception : Modèle de conception

- Classes = objets logiciels
- Ajout de la visibilité, d'interfaces, de méthodes, d'attributs



Liens entre diagrammes de séquences et de classes



[Figure extraite du livre de C. Larman]

Diagrammes de packages

Qu'est-ce qu'un diagramme de packages ?

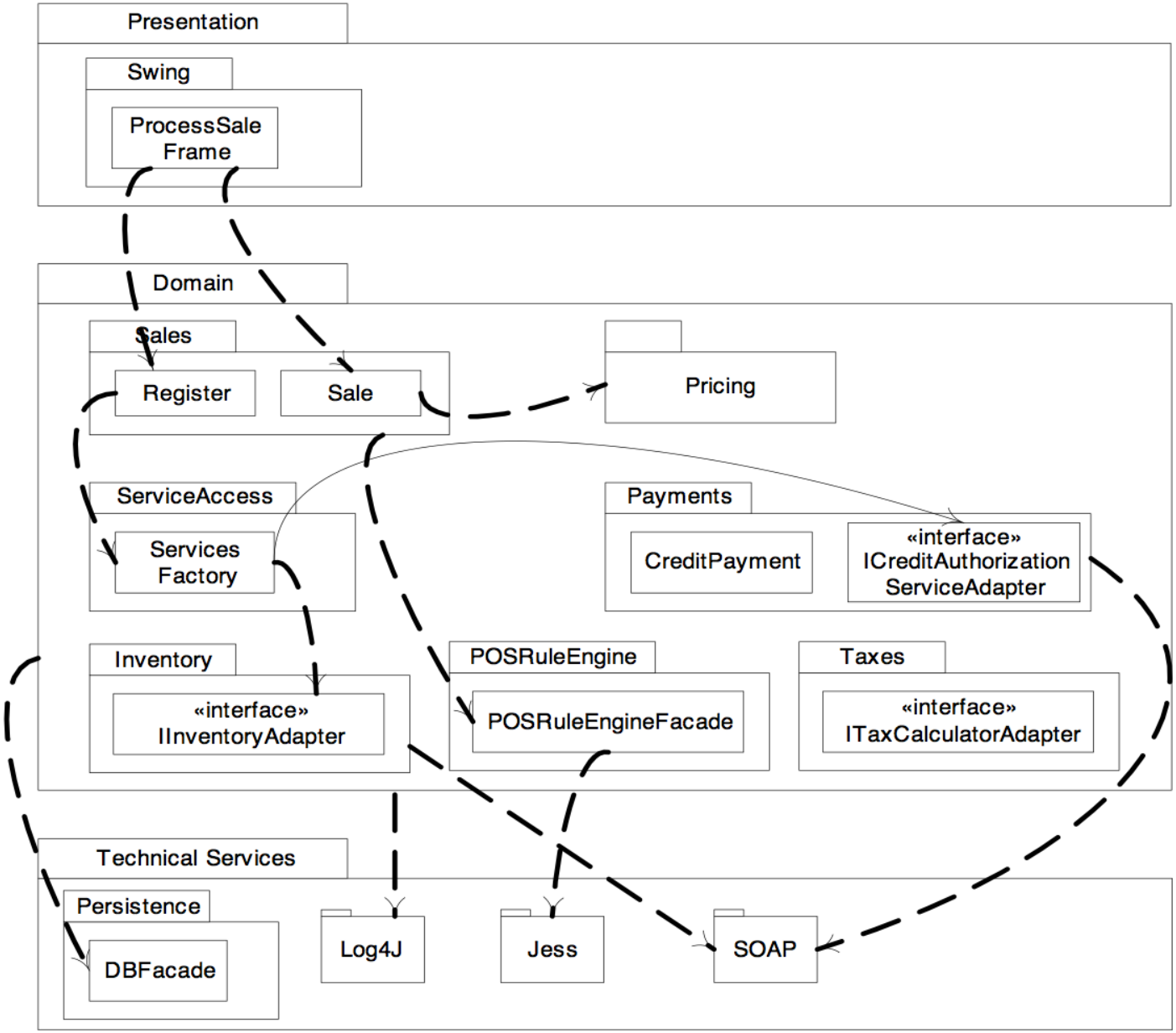
- Regroupement des classes logicielles en packages
  $\rightsquigarrow$ Point de départ pour un découpage en sous-systèmes
- Relations d'imbrication entre classes et packages ou entre packages
- Relations de dépendances entre packages

Objectifs d'un découpage en packages

- Encapsuler et décomposer la complexité
- Faciliter le travail en équipe
- Favoriser la réutilisation et l'évolutivité

Forte cohésion, Faible couplage et Protection des variations

Ex. de diagramme de packages / archi. en couches



Architecture de déploiement

Objectif

Décrire :

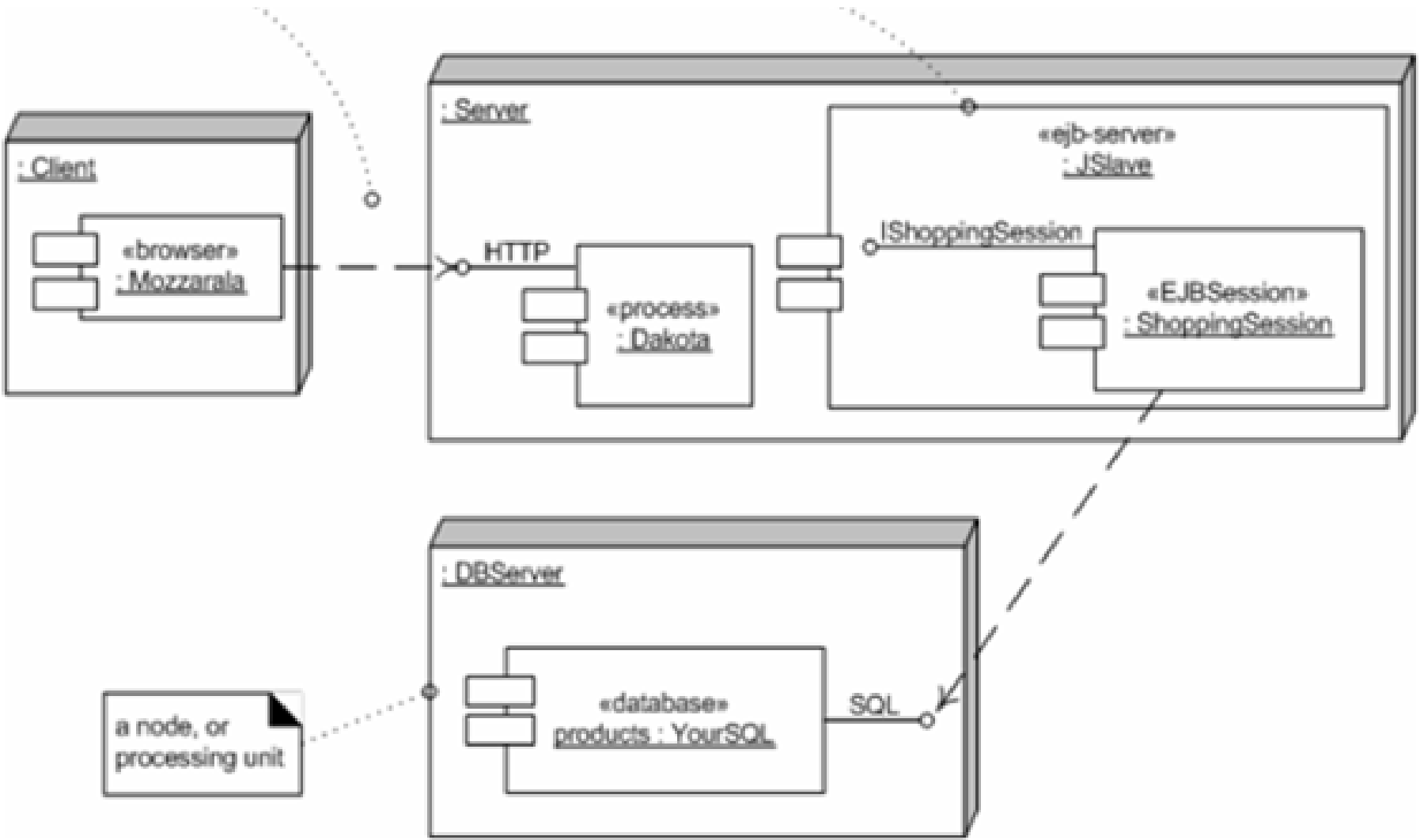
- la distribution des éléments logiciels sur les composants physiques
- la communication entre les composants physiques (réseau)

Utilisation de diagrammes de déploiement

- Nœuds physiques
  $\rightsquigarrow$ dispositifs physiques (ordinateur, téléphone, ...)
- Nœuds d'environnement d'exécution
  $\rightsquigarrow$ Ressource logicielle :
  - s'exécutant au sein d'un nœud physique
  - offrant un service pour héberger/exécuter d'autres logiciels

  (par ex. : O.S., Machine virtuelle, Navigateur Web, SGBD, ...)
- Liens
  $\rightsquigarrow$ Moyens de communication entre les nœuds

Exemple de diagramme de déploiement



De UML à la conception orientée objet

Craig Larman :

Drawing UML diagrams is a reflection of making decisions about the object design. The object design skills are what really matter, rather than knowing how to draw UML diagrams.

Fundamental object design requires knowledge of :

- *Principles of responsibility assignments*
- *Design patterns*

On y reviendra dans la deuxième partie du cours...

$\rightsquigarrow$ Illustration sur PlaCo

... et aussi dans le projet !

Plan de la partie 1

Développement logiciel itératif et agile

De la conception à la réalisation

Objectifs :

- Ecrire le code implémentant les cas d'utilisation ciblés
- Vérifier que ce code répond bien aux besoins

Ordre d'implémentation des packages et classes :

$\rightsquigarrow$ Du moins couplé au plus couplé

Génération automatique d'un squelette de code

- Diagrammes de classes
  - Définition des classes, attributs, signatures de méthodes, ...
  - Codage des associations 1-n par des collections
  - ...
- Diagrammes d'interaction :
  - Corps (partiel) des méthodes
  - Signature des constructeurs

Développement itératif et Reverse engineering

Le squelette généré automatiquement doit être complété

- Implémentation de la visibilité
  - $\rightsquigarrow$ Aptitude d'un objet o_1 à envoyer un message à un objet o_2
    - Visibilité persistante :
      - Visibilité d'attribut : o_2 est attribut de o_1
      - Visibilité globale : o_2 est attribut static public ou Singleton
    - Visibilité temporaire au sein d'une méthode p de o_1 :
      - Visibilité de paramètre : o_2 est paramètre de p
      - Visibilité locale : o_2 est variable locale de p
- Traitement des exceptions et des erreurs
- ...

En général, il doit aussi être modifié

$\rightsquigarrow$ Ajout de nouveaux attributs, méthodes, classes, ...

Utiliser des outils de reverse engineering à la fin de l'itération

$\rightsquigarrow$ Mise-à-jour des modèles de conception pour l'itération suivante

Développement piloté par les tests (Test-Driven Dev.)

$\rightsquigarrow$ Pratique popularisée par XP

Cycle de TDD :

- Ecrire le code de tests unitaires
- Exécuter les tests $\rightsquigarrow$ Echec !
- Compléter le code jusqu'à ce que les tests réussissent
  $\rightsquigarrow$ Implémentation la plus simple possible par rapport aux tests
- Retravailler le code (refactor), et re-tester

Avantages

- Les tests unitaires sont réellement écrits (!)
- Satisfaction du programmeur :
  Objectif clairement défini... défi à relever !
- Spécification du comportement des méthodes
- Assurance lors des modifications
- Automatisation et répétabilité du processus de test
  $\rightsquigarrow$ Utilisation d'outils (JUnit, CTest, ...)

Exemple d'automatisation des tests unitaires : JUnit

Principe de JUnit :

- Création d'une classe `TestXX` pour tester la classe `XX`
- Annotation des méthodes de `TestXX` :
  - `@Test` : méthode de test
  - `@Before` : méthode exécutée avant chaque méthode de test
  - `@After` : méthode exécutée après chaque méthode de test
  - `@BeforeClass` : méthode exécutée avant tous les tests
  - `@AfterClass` : méthode exécutée après tous les tests
- Vérification d'assertions dans les méthodes de test :
  - `assertEquals(x,y)` : vérifie que `x` et `y` ont la même valeur
  - `assertSame(x,y)` : vérifie que `x` et `y` pointent sur le même objet
  - `assert(not)Null(o)` : vérifie que `o == Null` (`o != Null`)
  - ...

Ce que JUnit ne fournit pas :

- Un moyen de générer un jeu de test
- Une évaluation de la couverture des tests ($\Rightarrow$ cobertura)

Quelques conseils pour l'écriture des tests...

- Couvrir tous les cas possibles : cas "moyens" (happy path tests), cas invalides, cas limites, ...
- Ecrire des tests simples... qui n'ont pas besoin d'être testés !
- Ajouter une procédure de test à chaque fois qu'un bug est découvert
- Eviter (proscrire ?) les effets de bord dans les procédures de test
- Tester le contrat et non l'implémentation

```
Pile p = new Pile();
Object o1 = new Object();
Object o2 = new Object():
p.empile(o1);
p.empile(o2);
assertSame(o2,p.depile());
assertSame(o1,p.depile());
assertTrue(p.estVide());
```

```
Pile p = new Pile();
Object o = new Object();
p.empile(o);
List l = p.getElts();
assertEquals(1,l.size());
assertSame(o,l.get(0));
p.depile();
assertTrue(p.estVide());
```

Refactoring régulier

Objectif :

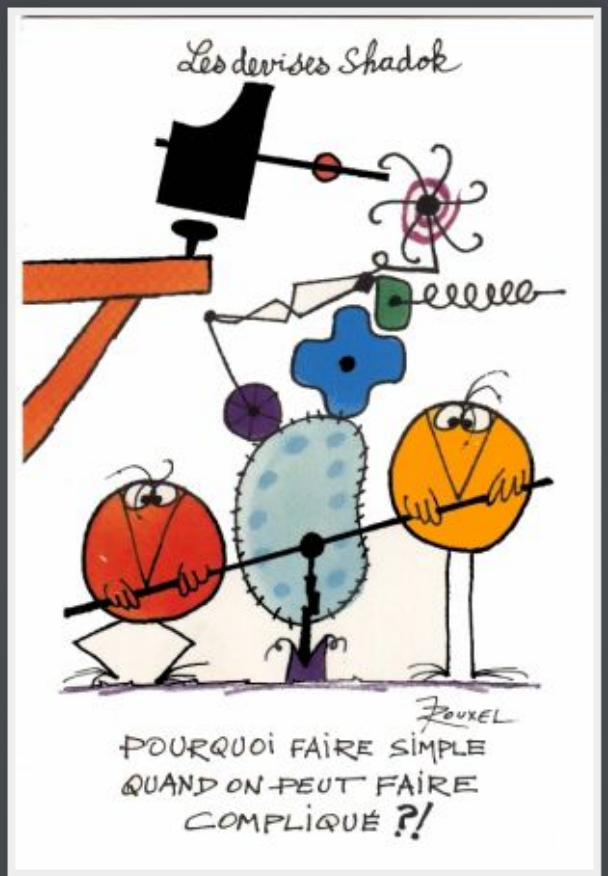
- Transformer/restructurer du code sans en modifier le comportement
  $\rightsquigarrow$ Supprimer les "code smells"
- Attention : re-exécuter les tests après chaque étape

Transformations par étapes

- Eliminer la duplication de code $\rightsquigarrow$ Création de procédures
- Améliorer la lisibilité $\rightsquigarrow$ Renommer
- Assurer le principe de responsabilité unique (single responsability)
- Raccourcir les méthodes trop longues
- Supprimer l'emploi des constantes codées en dur
- Réduire le nombre de variables d'instance d'une classe
- Renforcer l'encapsulation
- ...

cf `http://refactoring.com/catalog`

KISS



Keep It Simple, Stupid

> *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*
>
> — Martin Fowler

> *Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.*
>
> — John F. Woods

Transparents empruntés à Laurent Cottereau

Plan de la partie 1

Développement logiciel itératif et agile

Gestion de projet

Gestion des versions

- Structurer les artefacts par discipline $\rightsquigarrow$ un répertoire / discipline (sauf implémentation qui est parfois sur plusieurs répertoires)
- Utiliser un outil de gestion de versions / travail collaboratif (Git, SVN, ...)
  $\rightsquigarrow$ Créer un point de contrôle à la fin de chaque itération

Planification à deux niveaux différents :

- Plan de phase
  $\rightsquigarrow$ Macro plan visible de l'extérieur sur lequel l'équipe s'engage
  - Jalons et objectifs généraux
    $\rightsquigarrow$ Peu de jalons : fins de phases, tests "pilotes" à mi-phase
  - 1ère estimation peu fiable des jalons à la fin de l'inception
  - Estimation fiable et contractuelle à la fin de l'élaboration
    $\rightsquigarrow$ Engagement de l'équipe sur la livraison du projet
- Plan d'itération
  $\rightsquigarrow$ Micro organisation interne d'une itération
  - Le plan de l'itération n est fait à la fin de l'itération $n-1$
  - Adapter les plans d'itérations en fonction des évolutions

Plan d'itération

En fin d'itér. n, planifier l'itér. $n+1$ avec toutes les parties prenantes :
$\rightsquigarrow$ Client (*Product Owner*), Developpeurs, Architecte, Chef de projet, ...

- Déterminer la durée de l'itération (en général de 2 à 6 semaines)
- Lister les objectifs potentiels :
  nouvelles fonctionnalités / cas d'utilisation / scénarios de cas d'utilisation, traitement d'anomalies, ...
- Classer les objectifs par ordre de priorité :
  $\rightsquigarrow$ Obj. commerciaux du client / Obj. techniques de l'architecte
- Pour chaque objectif pris par ordre de priorité :
  - Etudier rapidement l'objectif
  - Estimer les tâches à faire pour l'atteindre
  - Quantifier temps nécessaire / ressources humaines disponibles
    $\rightsquigarrow$ Planning poker (http ://www.planningpoker.com/)

  Jusqu'à ce que volume de travail total = durée de l'itération

Impliquer toute l'équipe dans le processus de planification, et non juste le chef de projet

Vue globale d'une itération Agile