Méthodologie de Développement Objet

Première partie : Unified Software Development Process (USDP)

Christine Solnon

INSA de Lyon - 4IF

2014 - 2015

Contexte

Domaines d'enseignement du département IF :

- Système d'Information
- Réseaux
- Architectures matérielles
- Logiciel Système
- Méthodes et Outils Mathématiques
- Formation générale
- Développement logiciel :

En 3IF :

- C++
- Modélisation UML
- Génie logiciel

En 4IF :

- Ingénierie des IHM
- Méthodologie de dévelopmt objet
- Qualité logicielle
- Grammaires et langages

Référentiel des compétences (1/2)

Utiliser des diagrammes UML pour modéliser un objet d'étude

- Interpréter un diagramme UML donné
  ⇝ IF3-UML, IF4-DevOO, IF4-IHM
- Concevoir un diagramme UML modélisant un objet d'étude
  ⇝ IF3-UML, IF3-C++, IF3-DASI, IF4-DevOO, IF4-IHM, IF4-LG
- Vérifier la cohérence de différents diagrammes modélisant un même objet d'étude
  ⇝ IF3-UML, IF4-DevOO, IF4-LG

Concevoir l'architecture d'un logiciel orienté objet

- Structurer un logiciel en paquetages et classes faiblement couplés et fortement cohésifs
  ⇝ IF3-UML, IF3-C++, IF3-DASI, IF4-DevOO, IF4-LG
- Utiliser des Design Patterns
  ⇝ IF3-UML, IF3-C++, IF3-DASI, IF4-DevOO, IF4-LG

Référentiel des compétences (2/2)

Mettre en oeuvre une méthodologie pour concevoir, réaliser et maintenir des logiciels de qualité

- Mettre en oeuvre un processus de développement itératif tel que le “Unified Software Development Process" (USDP)
  ⇝ IF3-GL, IF4-DevOO, IF4-IHM
- Mettre en œuvre les principes du manifeste Agile
  ⇝ IF3-GL, IF4-DevOO
- Rédiger un cahier des charges logiciel
  ⇝ IF3-C++, IF3-GL, IF4-DevOO, IF4-QL

Implémenter de bons logiciels

- Mettre en œuvre à bon escient les mécanismes offerts par les langages de programmation orientés objet : héritage, généricité, surcharge, polymorphisme, ...
  ⇝ IF3-C++, IF3-GL, IF3-DASI, IF4-DevOO

Organisation

Cours

- du 23/09 au 9/10 : 6 cours “DevOO”
  - Partie 1 (3 cours) : USDP
  - Partie 2 (1 cours) : Méthodes Agiles par S. Sorlin (Esker)
  - Partie 3 (1 cours) : Ingénierie des modèles
  - Partie 4 (1 cours) : Présentation du projet
- du 22/10 au 07/10 : 6 cours “Ingénierie des IHM" par Léa Laporte

Mise-en-pratique : Projet “Gestion de tournées de livraisons"

- 5 séances de 4h $\rightsquigarrow$ Développement objet
- 3 séances de 4h $\rightsquigarrow$ IHM

Inspiré du projet de R&D *Optimod'Lyon* piloté par le Grand Lyon

Evaluation

- DS le 18 décembre + Note de projet

Quelques livres à emprunter à DOC'INSA

- Le processus unifié de développement logiciel
  Ivar Jacobson, Grady Booch, James Rumbaugh
  ⇝ *Description d'USDP par ses concepteurs*
- UML 2 et les design patterns
  Craig Larman
  ⇝ *Mise en œuvre d'USDP dans un esprit Agile*
  ⇝ *Bonne introduction aux design patterns*
- Modélisation Objet avec UML
  Pierre-Alain Muller, Nathalie Gaertner
  ⇝ *Bon rappel sur UML pour l'analyse et la conception OO*
  ⇝ *Brève présentation d'USDP*
- ...et plein d'autres !

Plan du cours

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 50000 projets depuis 2004)

	2004	2006	2008	2010	2012
Succès	29%	35%	32%	37%	39%
Echec	18%	19%	24%	21%	18%
Mitigé	53%	46%	44%	42%	43%

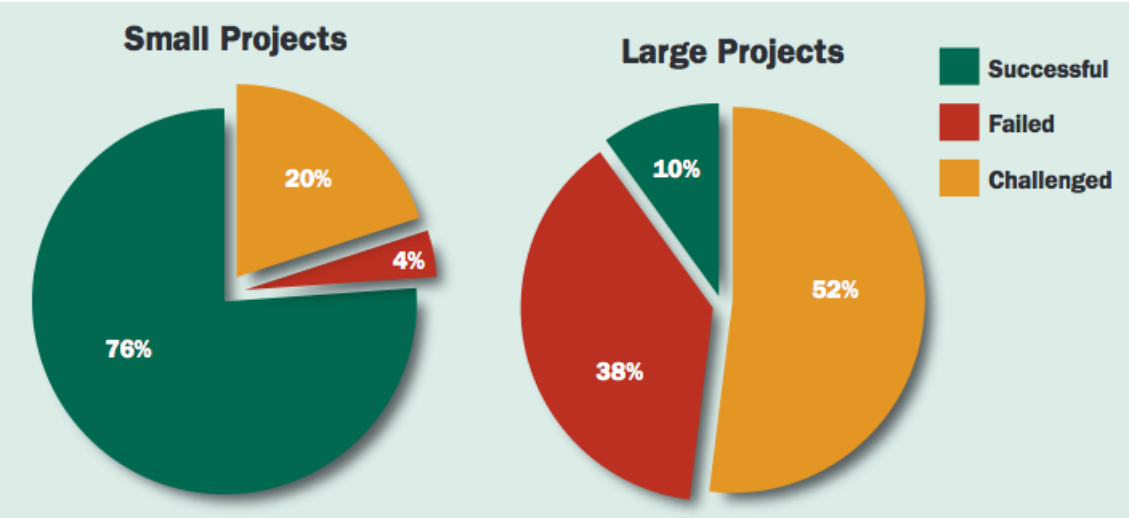
- Succès : livré à temps, sans surcoût et avec toutes les fonctionnalités
- Echec : abandonné en cours de route
- Mitigé : livré avec moins de fonctionnalités + dépassement coût ou tps

	2004	2006	2008	2010	2012
Dépassement temps	84%	72%	79%	71%	74%
Dépassement coût	56%	47%	54%	46%	59%
Fonctionnalités livrées	64%	68%	67%	74%	69%

⇝ **On progresse un peu... mais 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%

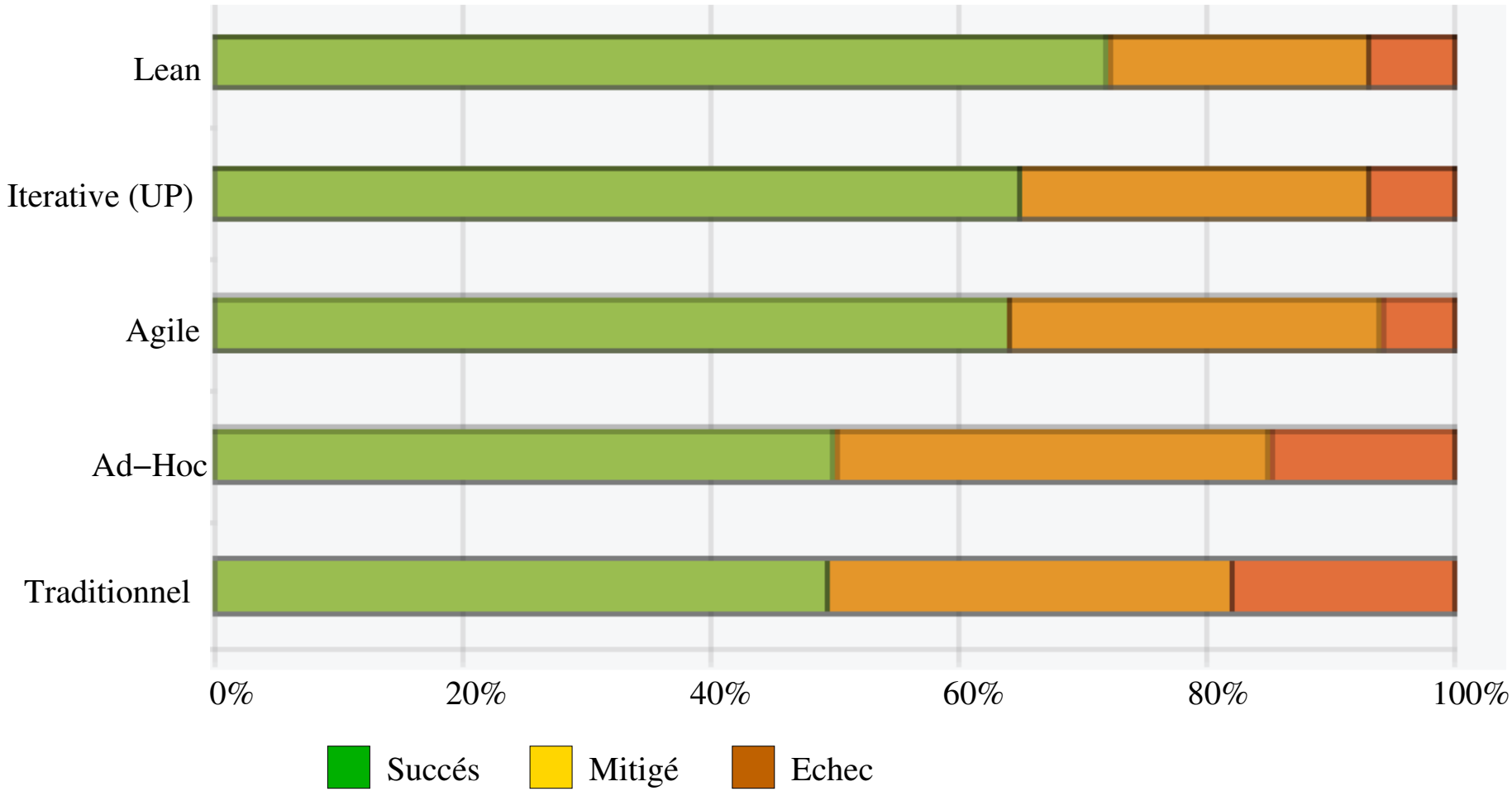
Extrait des conclusions de l'étude :

"Research at the Standish Group indicates that smaller time frames, with delivery of software components early and often, will increase the success rate. Shorter time frames result in an iterative process of design, prototype, develop, test and deploy small elements. This process is known as growing software as opposed to the old concept of developping software. Growing software engages the user earlier."

Unified Software Development Process (USDP / UP)

- Processus itératif populaire pour le développement orienté objet
- Processus flexible et ouvert $\rightsquigarrow$ Agile et XP compatible !

Etude de S. Ambler sur 173 projets en 2013



Plan du cours

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
- Analystes et 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, ...
- L'analyste/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

Le manifeste Agile (2001) 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 du cours

Vue globale de la vie d'un logiciel avec USDP

La vie d'un logiciel est composée de cycles

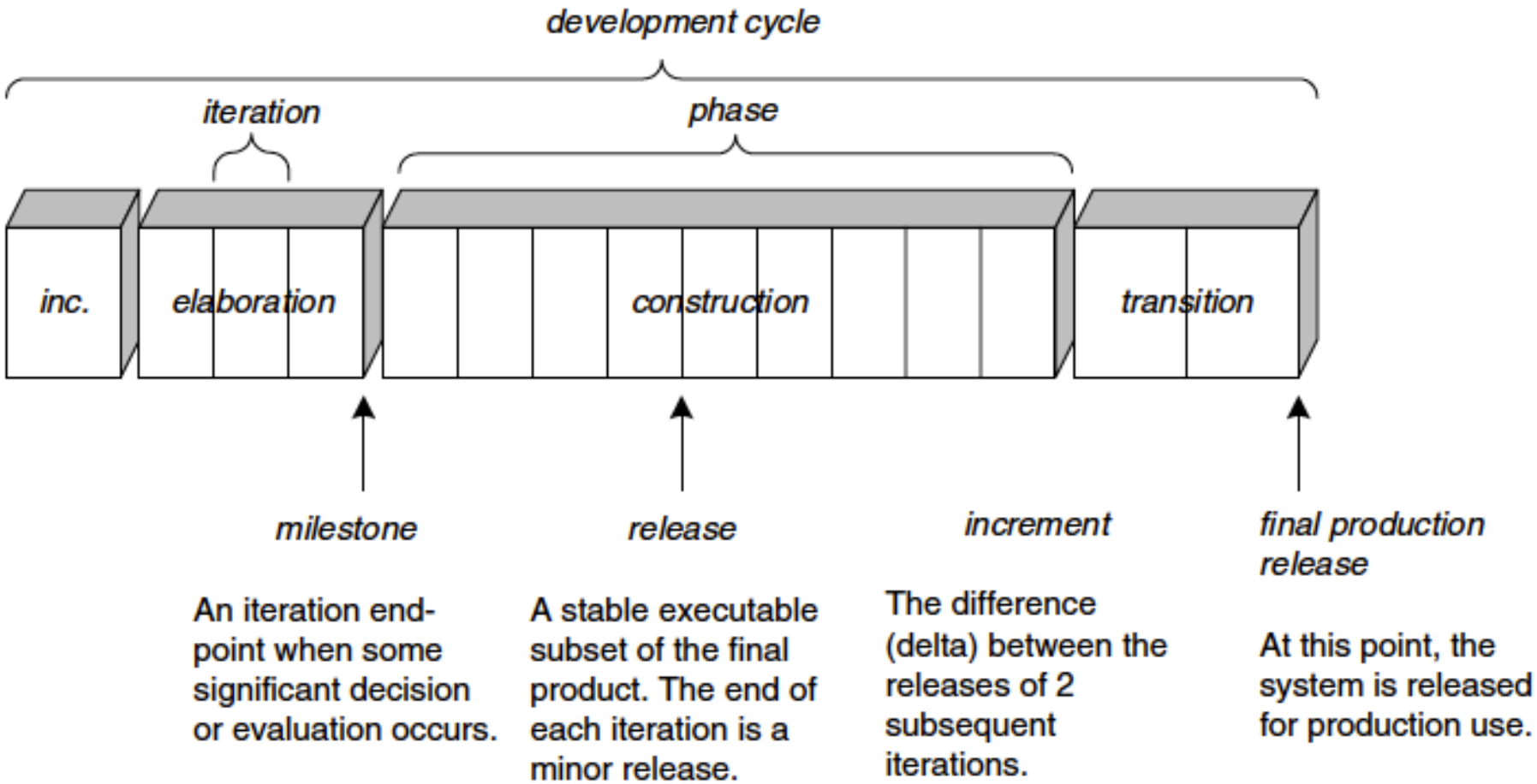
- 1 cycle $\Rightarrow$ 1 nouvelle version du logiciel
- Chaque cycle est composé de 4 phases :
  - Etude préliminaire (*Inception*)
  - Elaboration
  - Construction
  - Transition

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 USDP



[figure extraite du livre de C. Larman]

Plan du cours

Phase 1 : Etude préliminaire (*inception*)

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

  ⇝ 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

⇝ 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 ?

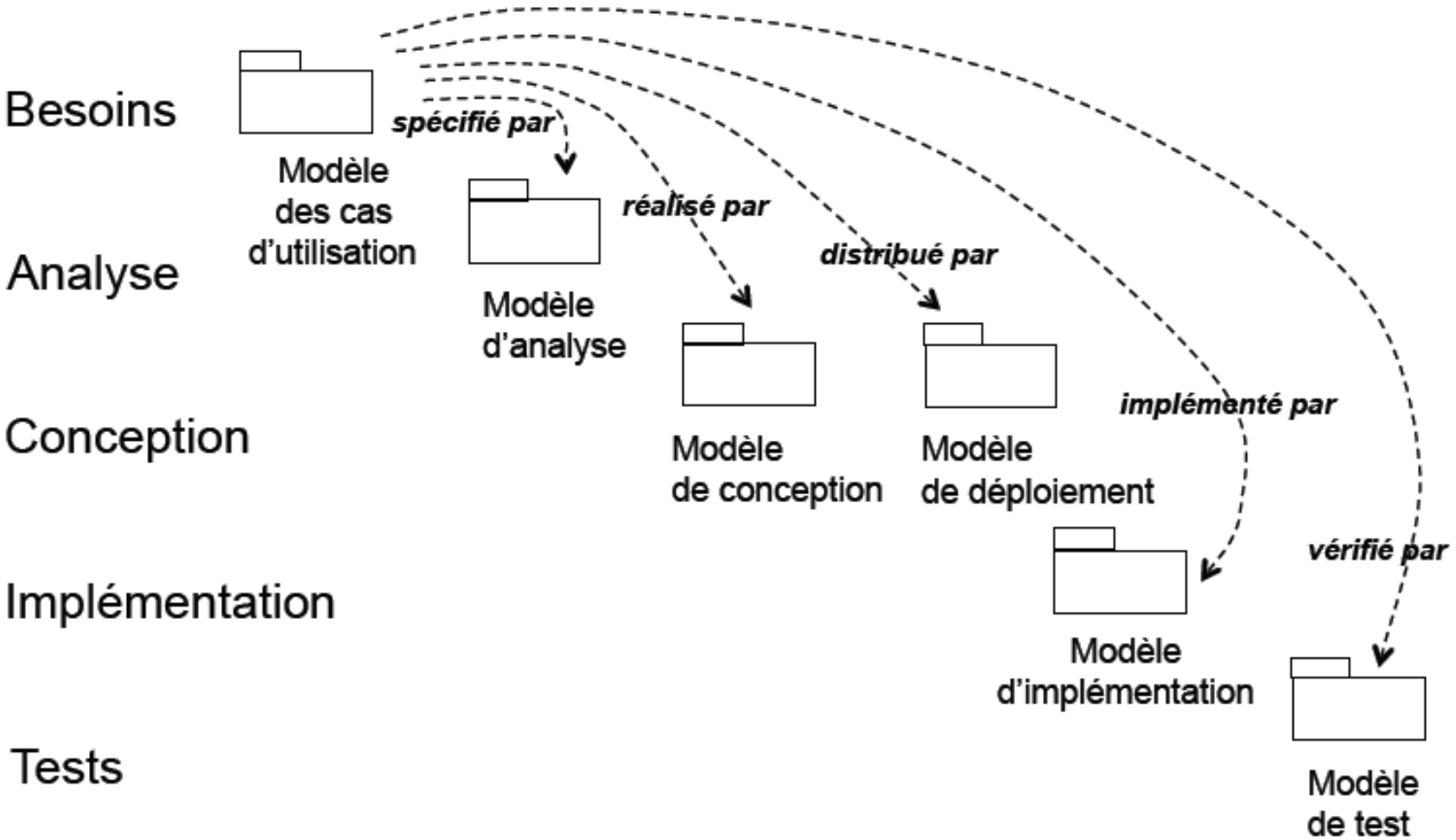
Plan du cours

Processus guidé par les cas d'utilisation

Les cas d'utilisation :

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

Cas d'utilisation / activités



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

Processus centré sur l'architecture

Architecture :

- Vue des aspects les plus significatifs du système
  - $\rightsquigarrow$ Abstraction des principaux modèles du système
- Conçue et stabilisée lors des premières itérations
  - $\rightsquigarrow$ Traiter en premier les cas d'utilisation "pertinents" :
    - les plus risqués / critiques
    - les plus importants pour le client
    - les plus représentatifs du système

Processus itératif et incrémental

Itération = mini projet donnant lieu à un incrément

Avantages :

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

Ce qu'on va voir dans la suite du cours...

- Description des différentes activités d'une itération
  - Que faut-il décrire ?
  - Sous quelle forme ?
  - Comment faire cela ?

  ⇝ Description technique de la méthode
- Description des différentes phases
  - Quel est le but à atteindre ?
  - Quels sont les livrables ?
  - Quel niveau d'abstraction ?

  ⇝ Description du déroulement du projet

Plan du cours

Vue globale d'une itération "générique"

Mini cascade qui enchaîne différentes activités :

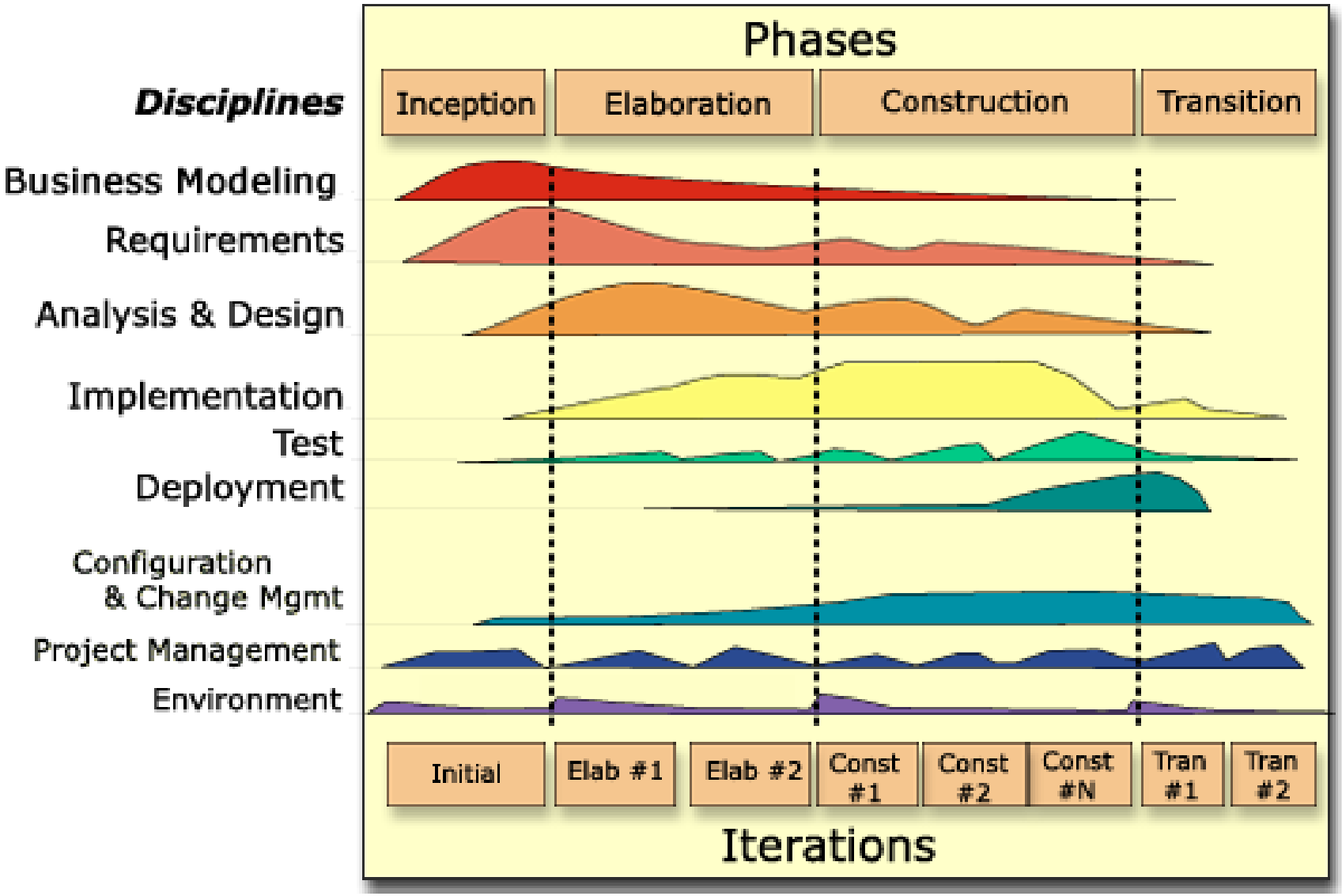
- Capture et analyse des besoins
  ⇝ Modéliser le système vu de l'extérieur
- Conception
  ⇝ Modéliser le système vu de l'intérieur
- Réalisation et tests
  ⇝ Implémenter le système
  ⇝ Valider l'implémentation par rapport aux besoins

⇝ 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 du cours

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

- 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
$\rightsquigarrow$ 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"

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

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

Glossaire

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

Au travail !

Extrait du projet donné en 2011 : Gestion des bagages d'un aéroport

L'application a pour but de gérer les bagages dans un aéroport depuis le guichet de l'enregistrement de départ jusque dans l'avion, ou bien de l'arrivée de l'avion jusqu'au carrousel de récupération des bagages. Les bagages sont transportés par des chariots autonomes circulant sur des rails reliés par des embranchements. Chaque bagage est muni d'une puce RFID permettant à des capteurs de communiquer sa position au système. Les postes de supervision affichent des vues sur l'ensemble du système de transport avec la position des chariots et des bagages, l'état des éléments (panne, batterie déchargée...).
(...)
A l'enregistrement, chaque bagage est posé sur un tapis roulant qui l'achemine jusque sur un chariot. Le chariot circule sur des rails de façon autonome jusqu'au toboggan de destination (à chaque embranchement, il choisit son chemin).
(...)
A l'arrivée d'un avion, les bagages sont acheminés par un train jusqu'à un tapis roulant sur lequel ils sont déposés un à un par un opérateur. Le tapis roulant achemine chaque bagage jusque sur un chariot qui circule ensuite de façon autonome jusqu'au carroussel de récupération des bagages.
(...)

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

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 (instances)
  $\rightsquigarrow$ Scénario de base et ses variantes ($\rightsquigarrow$ cas particuliers)

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 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

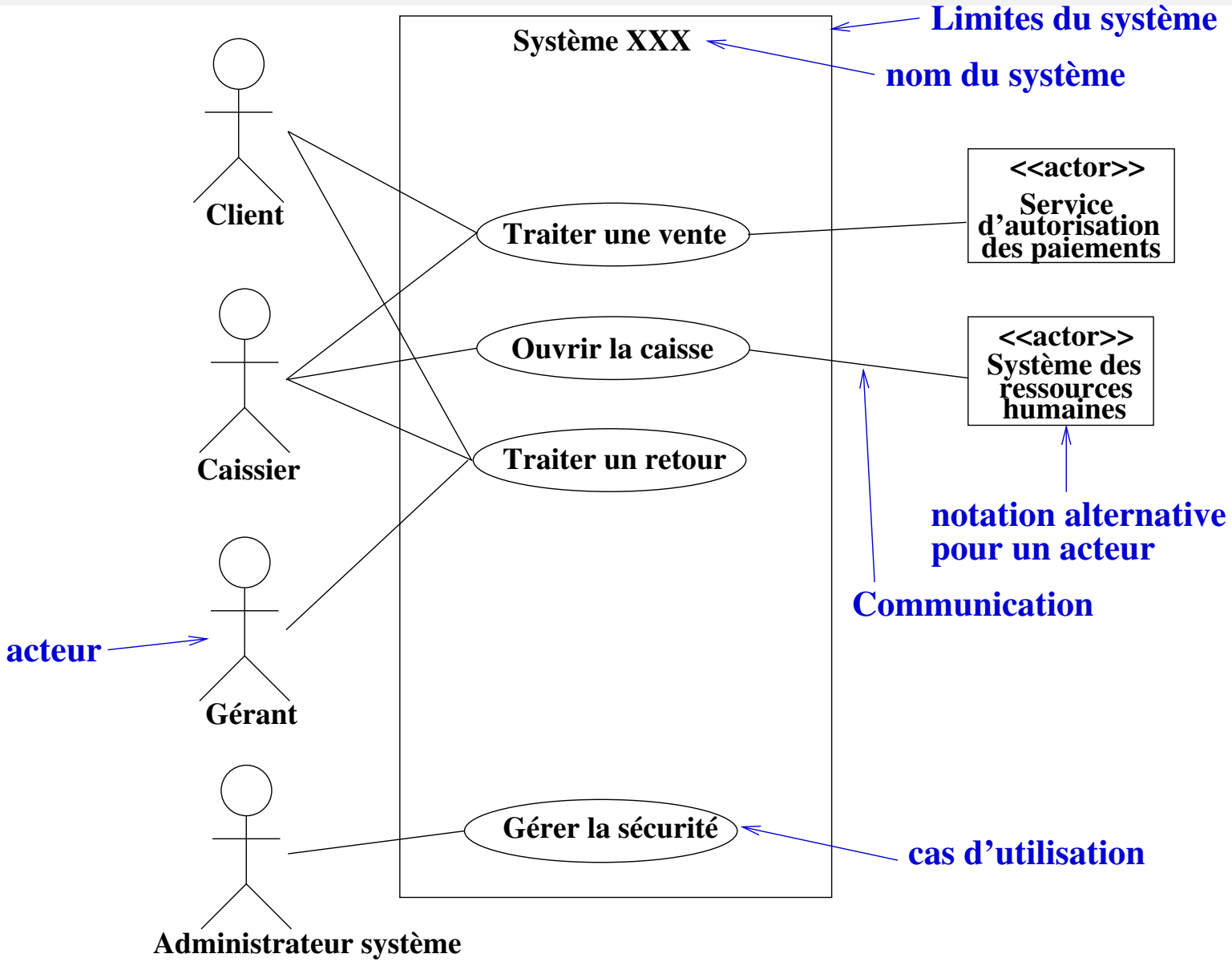
Comment décrire les cas d'utilisation ?

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)

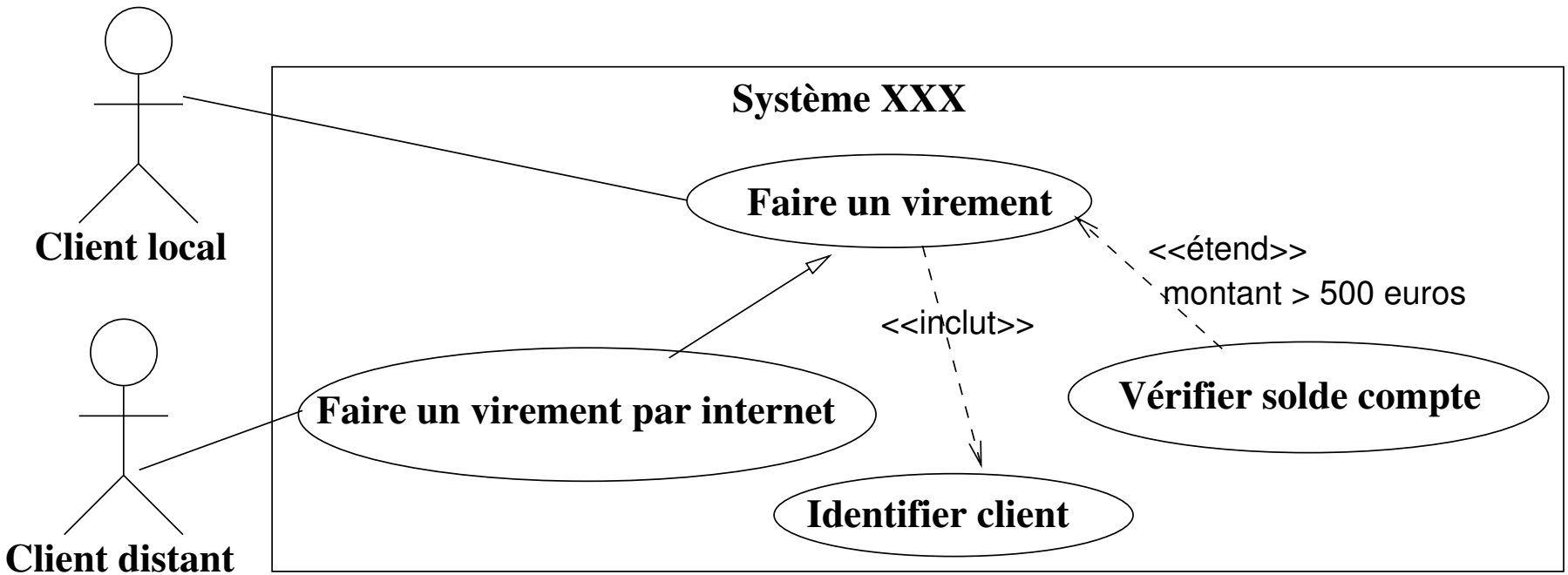
Relations entre cas d'utilisation et acteurs



Relations entre cas d'utilisation (rappel 3IF)

- Généralisation : ——▷
- Inclusion : - - - - - «include» - - - - - >
- Extension : - - - - - «extend» - - - - - > (préciser la condition)

⇝ A utiliser avec la plus grande modération



Description textuelle abrégée d'un cas d'utilisation

Scénario de base décrit en un paragraphe

$\rightsquigarrow$ Histoire d'un acteur qui utilise le système pour atteindre un but

Exemple : Description abrégée de "Traiter une vente"

Le caissier utilise le système pour enregistrer les articles qu'un client souhaite acheter. Le système affiche le détail des articles et le montant total des achats, hors taxe et avec taxe. Le client fournit les informations nécessaires pour le règlement. Le système valide et enregistre ces informations, puis met à jour les quantités en stock et imprime le ticket de caisse destiné au client.

Description textuelle structurée d'un cas d'utilisation

Martin Fowler :

There is no standard way to write the content of a use case, and different formats work well in different cases.

Structuration proposée par Martin Fowler :

- Titre : But que l'acteur principal souhaite atteindre avec ce cas d'utilisation

  $\rightsquigarrow$ Verbe à l'infinitif suivi d'un complément d'objet

- Scénario de base : Liste numérotée d'interactions entre les acteurs et le système
- Extensions : Une liste numérotée d'interactions pour chaque cas particulier possible

Exemple de description structurée d'un cas d'utilisation

- Titre = Traiter une vente
- Scénario de base =
  1. Le Caissier indique au système qu'il commence une nouvelle saisie
  2. Le Caissier entre le code d'un article et une quantité
  3. Le Système affiche le prix de l'article et le total en cours HT et TTC
     - *Répétition des étapes 2 et 3 jusqu'à ce que tous les articles soient saisis*
  4. Le Système affiche le total final HT et TTC
  5. Le Caissier sélectionne un mode de paiement
  6. ...
- Extensions :

2a. Le code de l'article est invalide

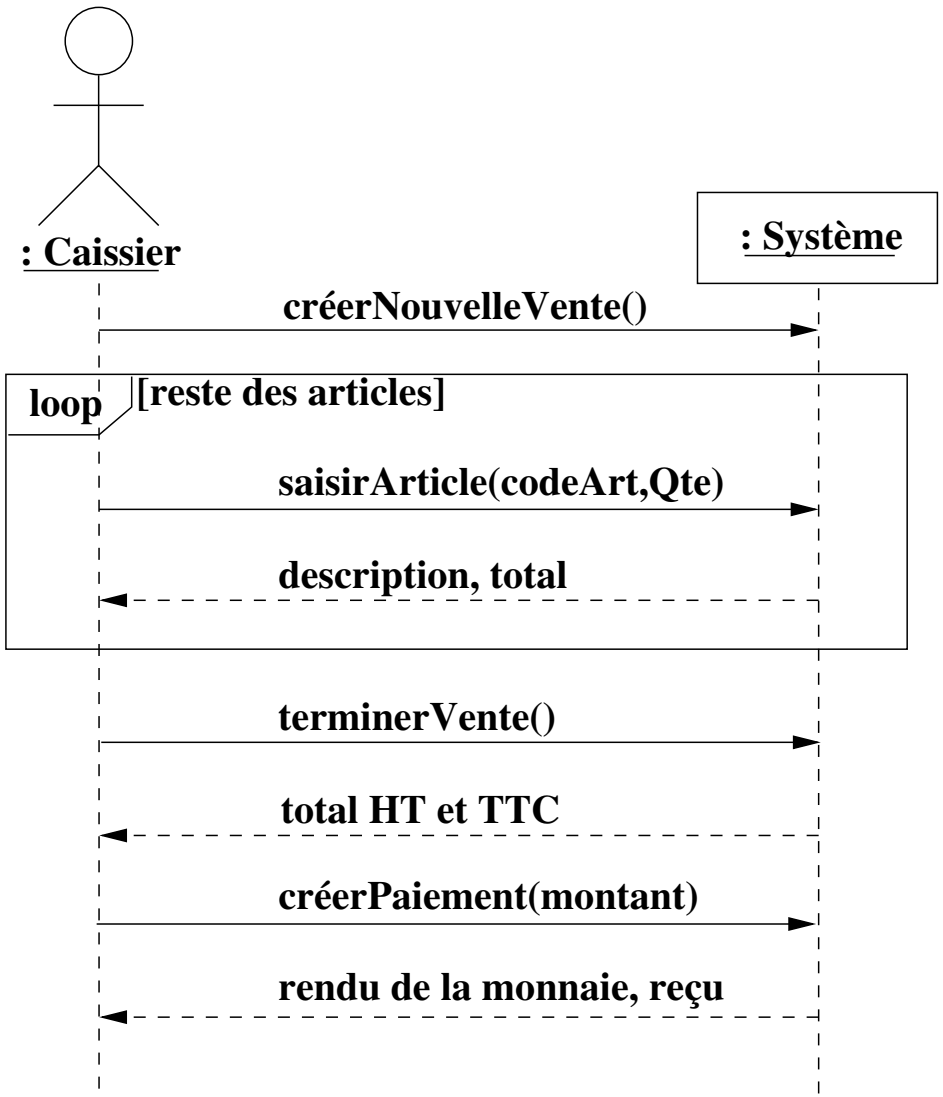
1. Le Système signale l'erreur et rejette la saisie
2. S'il existe un code lisible par un être humain, alors ...
3. Sinon si le prix est sur l'article, alors ...
4. ...

2b. La quantité saisie est négative

...

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 ?

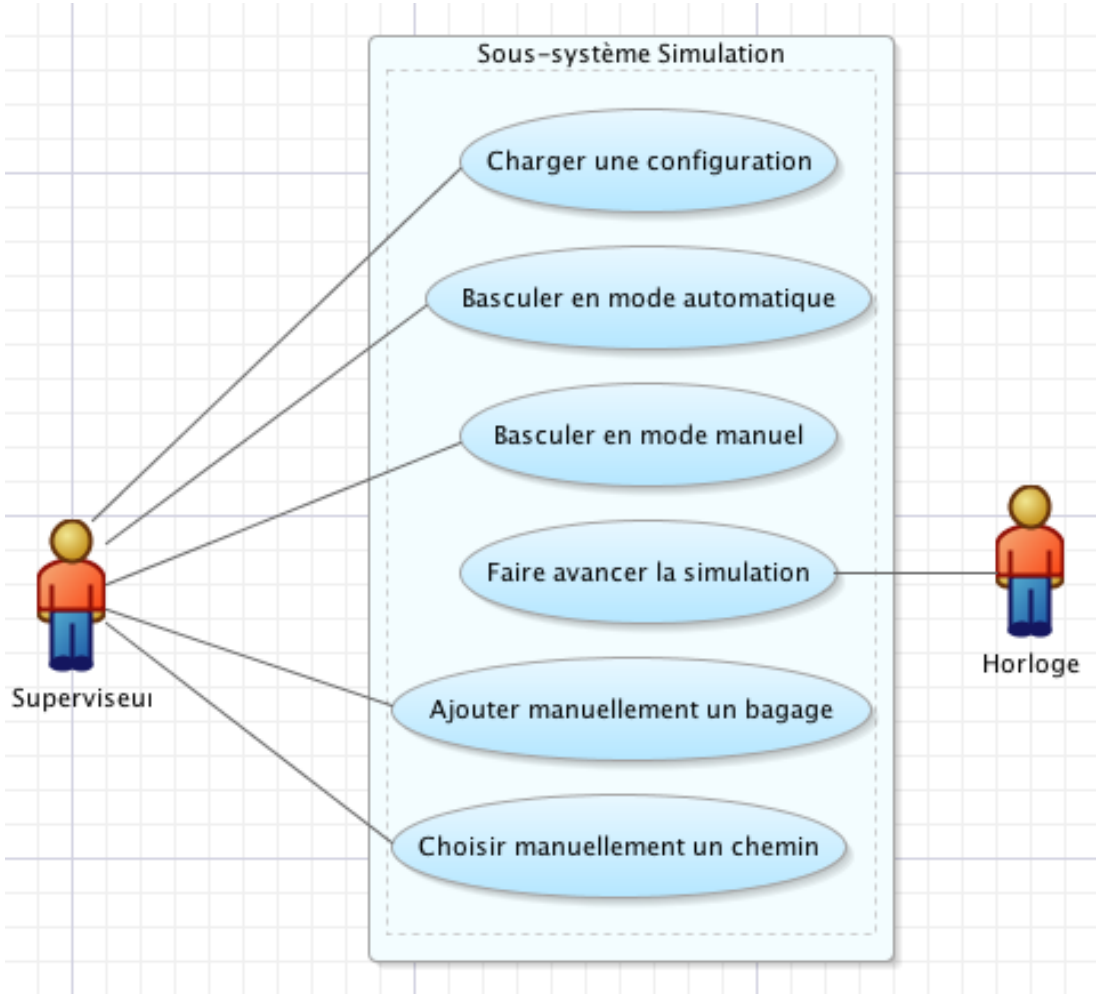
User story :

- 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

Au travail !

Extrait du projet donné en 2011 : Gestion des bagages d'un aéroport



- Scénarios et diagrammes de séquence système des cas :
  - Ajouter manuellement un bagage
  - Faire avancer la simulation

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
  ⇝ 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
  ⇝ Résumé des cas d'utilisation et exigences supplémentaires

Maquette de l'IHM

- Uniquement si IHM complexe
  ⇝ 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
- ...

⇝ Evaluer les impacts, la priorité et les difficultés/risques

Plan du cours

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
- Documenter l'architecture

Artefacts

- Diagrammes d'interaction
- Diagrammes de classes, de packages et de déploiement
- Document d'architecture du logiciel (N+1 vues)

Diagrammes d'interaction (rappels de 3IF)

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

Pendant la capture des besoins :

⇝ Interactions entre acteurs et système (système = boite noire)

- Décrire les scénarios des cas d'utilisation

Pendant la conception :

⇝ Interactions entre objets (à l'intérieur 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 !

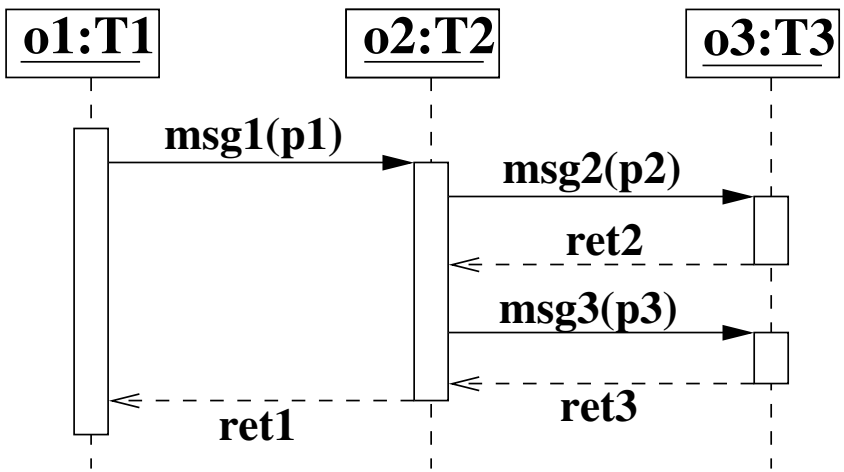
Diagrammes de séquence vs diagrammes de communication (rappels de 3IF)

Diagrammes de séquence :

Structuration en termes de

- temps ⇝ axe vertical
- objets ⇝ axe horizontal

Exemple :

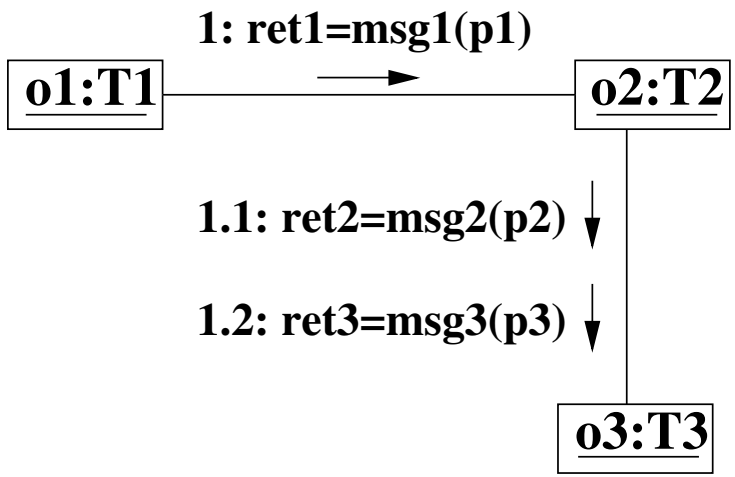


Diagrammes de communication :

Structuration en multigraphe

- Numérotation des arcs pour modéliser l'ordre des interactions

Exemple :



Buts et Artefacts

Buts

- Appréhender la logique comportementale (interactions entre objets)
- Appréhender la logique structurelle
- Documenter l'architecture

Artefacts

- Diagrammes d'interaction
- Diagrammes de classes, de packages et de déploiement
- Document d'architecture du logiciel (N+1 vues)

Diagrammes de classes (rappel de 3IF)

Un diagramme de classes est un graphe :

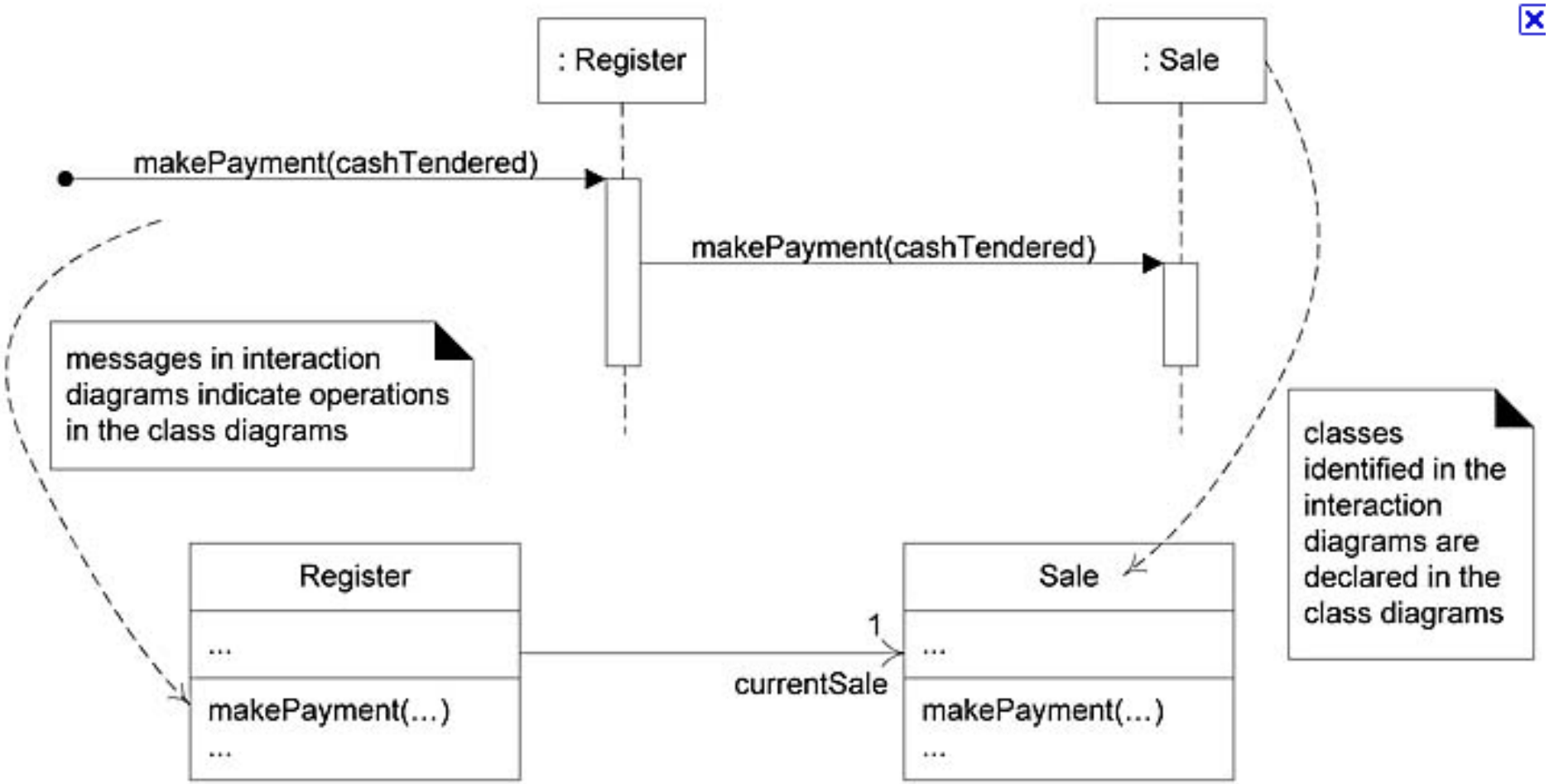
- Nœud du graphe = Classe (abstraction d'un ensemble d'objets)
- Arc du graphe = Relation entre des classes :
  - Relation d'association : A ——— B ou A ———> B
    $\rightsquigarrow$ Abstraction d'un d'ensemble de liens entre objets
  - Relation de généralisation / spécialisation : A ———▷ B
    $\rightsquigarrow$ Factorisation de propriétés communes à plusieurs classes
  - Relation de dépendance : A - - - - - - - - - - > B
    $\rightsquigarrow$ La modification de B peut avoir un impact sur A

Très nombreuses utilisations, à différents niveaux :

- Pendant la capture des besoins : Modèle du domaine
  $\rightsquigarrow$ Classes correspondant à des objets du domaine
- Pendant la conception/implémentation : Modèle de conception
  $\rightsquigarrow$ Classes correspondant à des objets logiciels

Réduire le décalage des représentations...

Liens entre diagrammes de classes et d'interaction



[Figure extraite du livre de C. Larman]

Principes et patrons de conception OO (rappels de 3IF)

Objectif : Concevoir de bons modèles orientés objet

⇝ Faciles à comprendre, à mettre en œuvre, à maintenir et à réutiliser

Principes de conception GRASP de Craig Larman

- GRASP = General Responsability Assignment Software Patterns
- Principes de base pour affecter des responsabilités aux objets :
  - Qui doit savoir ?
  - Qui doit faire ?

Patrons de conception (Design patterns) du GoF

- GoF = Gang of Four : E. Gamma, R. Helm, R. Johnson, J. Vlissides
- Catalogue de patrons
  ⇝ Description nommée de pb récurrents et de solutions standards
  - Réutiliser des solutions avérées
  - Faciliter le dialogue

  ⇝ Vocabulaire commun entre concepteurs et réalisateurs

Principes GRASP (rappels de 3IF)

Faible couplage :

Affecter les responsabilités de façon à éviter les couplages inutiles

Forte cohésion :

Faciliter la compréhension, gestion et réutilisation des objets en affectant leurs propriétés pour qu'ils aient une forte cohésion

Polymorphisme :

Manipuler des instances de classes différentes de façon uniforme

Protection des variations :

Identifier les points de variation ou d'évolution et créer une interface stable autour

Indirection :

Eviter le couplage entre différentes entités en ajoutant des objets intermédiaires

Fabrication pure :

Créer des objets logiciels ne correspondant pas à des concepts du domaine pour assurer Forte cohésion et Faible couplage

23 Patterns GoF (rappels de 3IF)

Patterns de création

- Abstract factory : Création de familles d'objets
- Singleton : Assurer qu'une classe a une seule instance accessible globalement
- Factory method : Méthode chargée de créer des instances

et aussi : Prototype, Builder

Patterns structuraux

- Adapter : Fournir une interface stable à un composant dont l'interface peut varier
- Decorator : Attacher dynamiquement des responsabilités à un objet
- Composite : Représenter des hiérarchies et manipuler de façon uniforme composants et composés

et aussi : Bridge, Facade, Flyweight, Proxy

Patterns comportementaux

- Iterator : Accès aux éléments d'un agrégat indépendamment de l'implémentation
- Observer : Faire savoir à des objets qu'un objet observable a été modifié
- Command : Séparer le code à l'origine d'une action, du code de l'action

et aussi : Visitor, Strategy, Chain of responsiblity, Interpreter, Mediator, Memento, State, Template method

Diagrammes de packages et Architecture logique (rappel 3IF)

Qu'est-ce qu'une architecture logique ?

- Regroupement des classes logicielles en packages
  $\rightsquigarrow$ Point de départ pour un découpage en sous-systèmes

Objectifs d'un découpage en packages

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

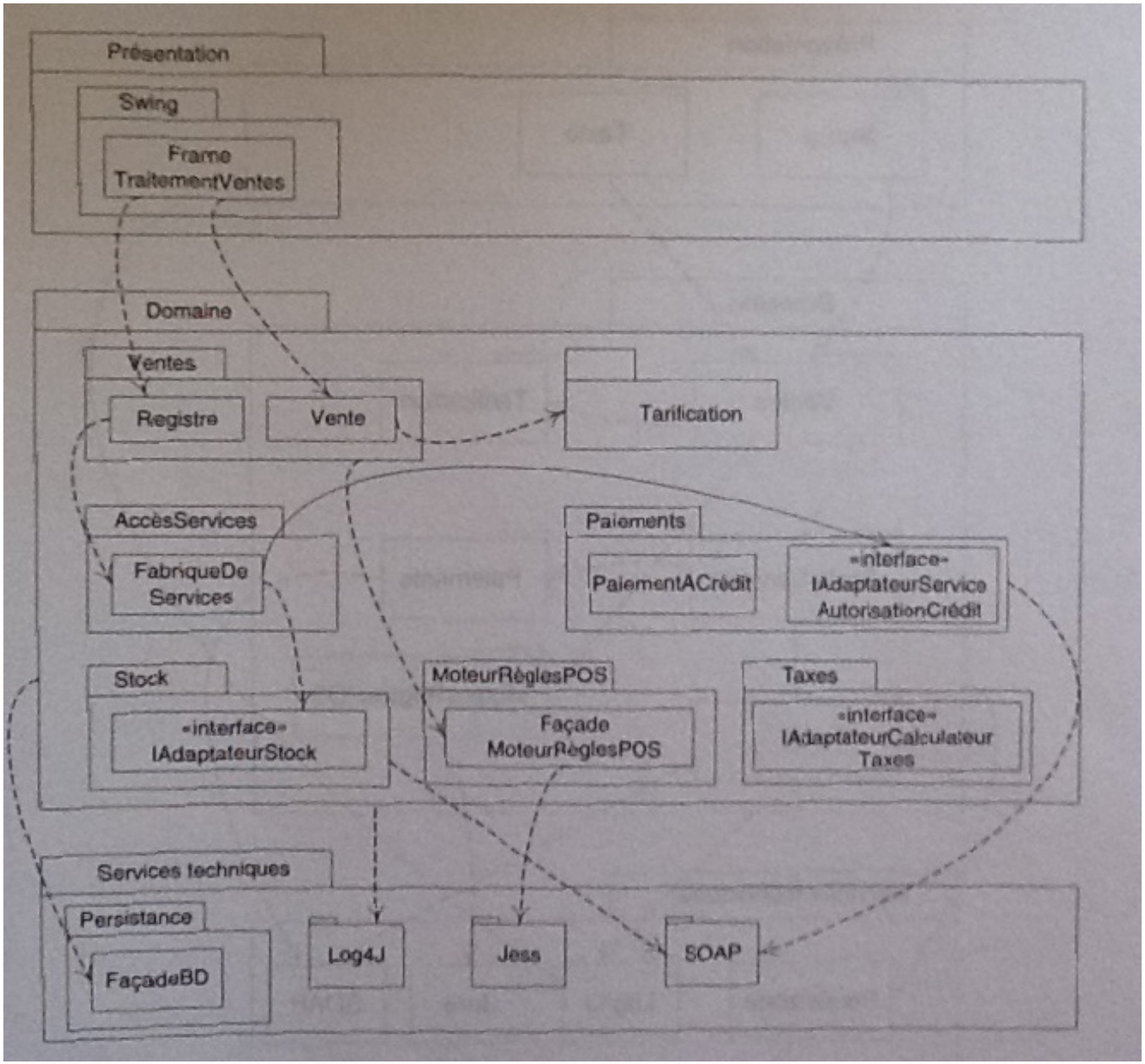
Forte cohésion, Faible couplage et Protection des variations

Utilisation d'UML pour décrire une architecture logique

Diagramme de packages :

- Relations d'imbrication entre classes et packages ou entre packages
- Dépendances entre packages

Exemple d'architecture logique en couches



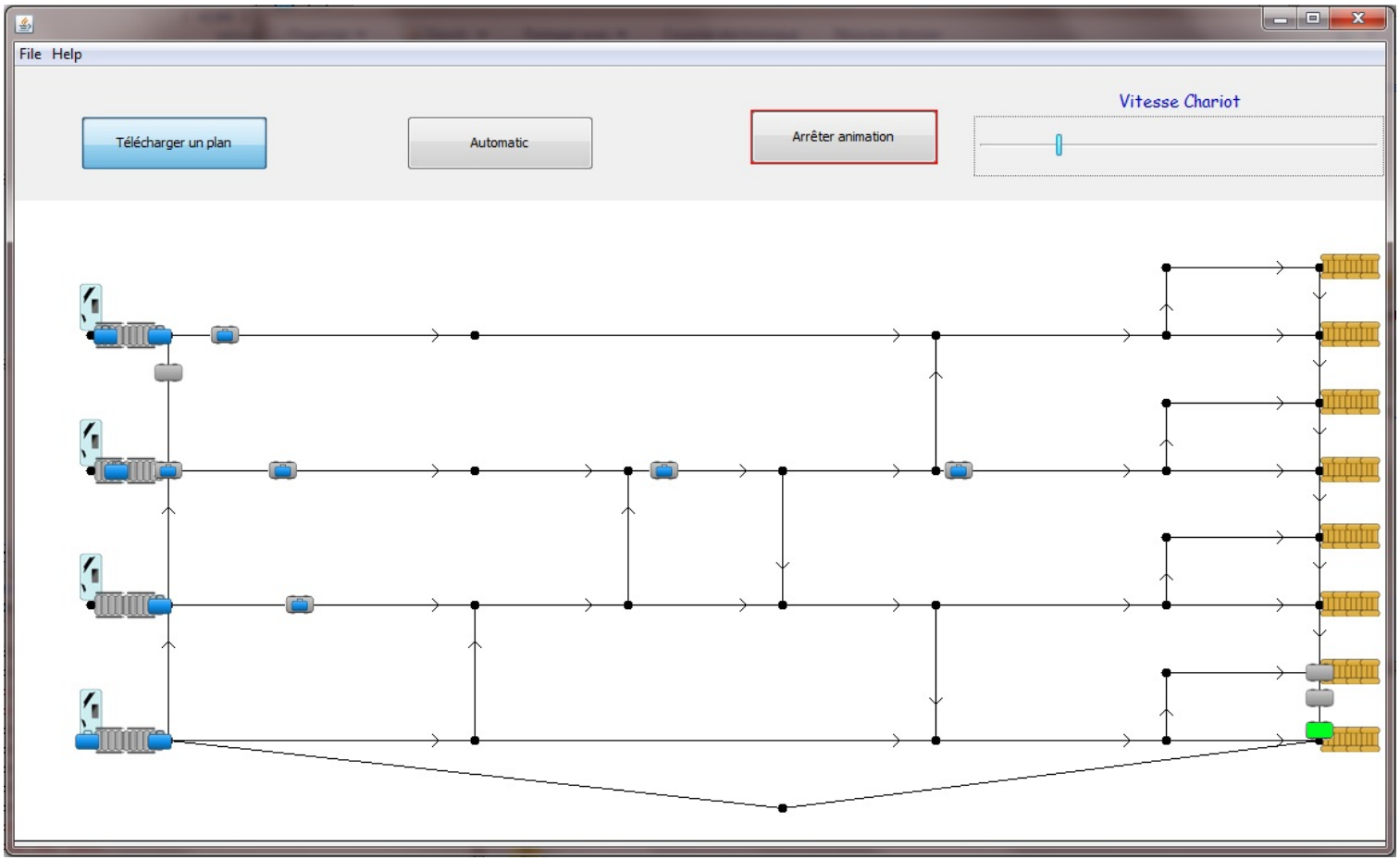
Patterns d'architectures logiques

Exemples de patterns architecturaux

- Layers (stricte ou lâche)
  $\rightsquigarrow$ Présentation, Application, Domaine, ServicesTech., ...
- Pipes and filters
- Multitiers
- Model-View-Controller
- Peer-to-peer
- Blackboard
- ...

Au travail !

Exemple de simulation pour le projet de gestion des bagages d'un aéroport



- Diagramme de séquence modélisant les interactions entre objets pour réaliser le cas "Faire avancer la simulation"
- Diagrammes de classes et de packages correspondants

Architecture de déploiement

Objectif

Décrire :

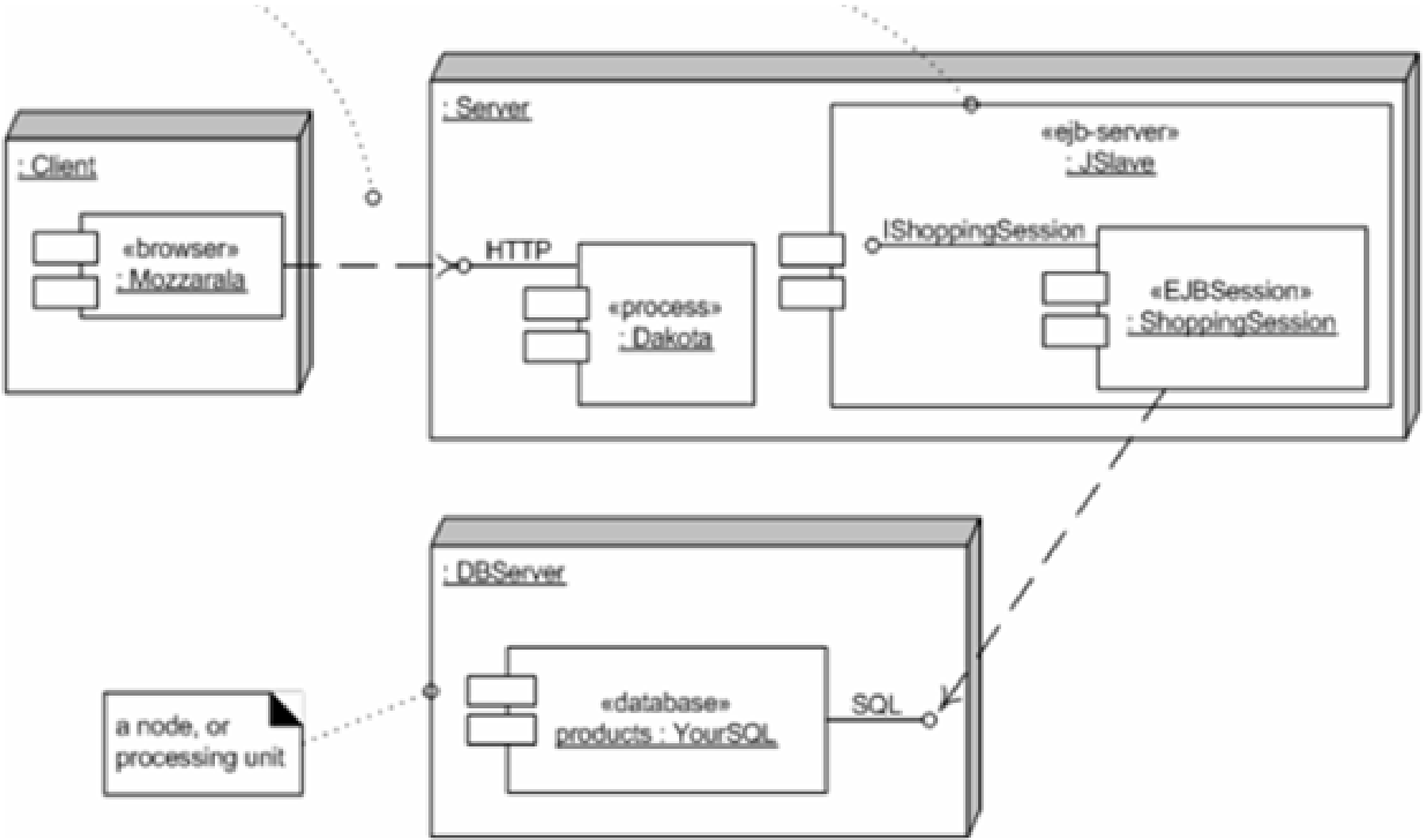
- la distribution des éléments logiciels sur les composants physiques
- la communication entre les composants physiques (réseau)

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



Buts et Artefacts

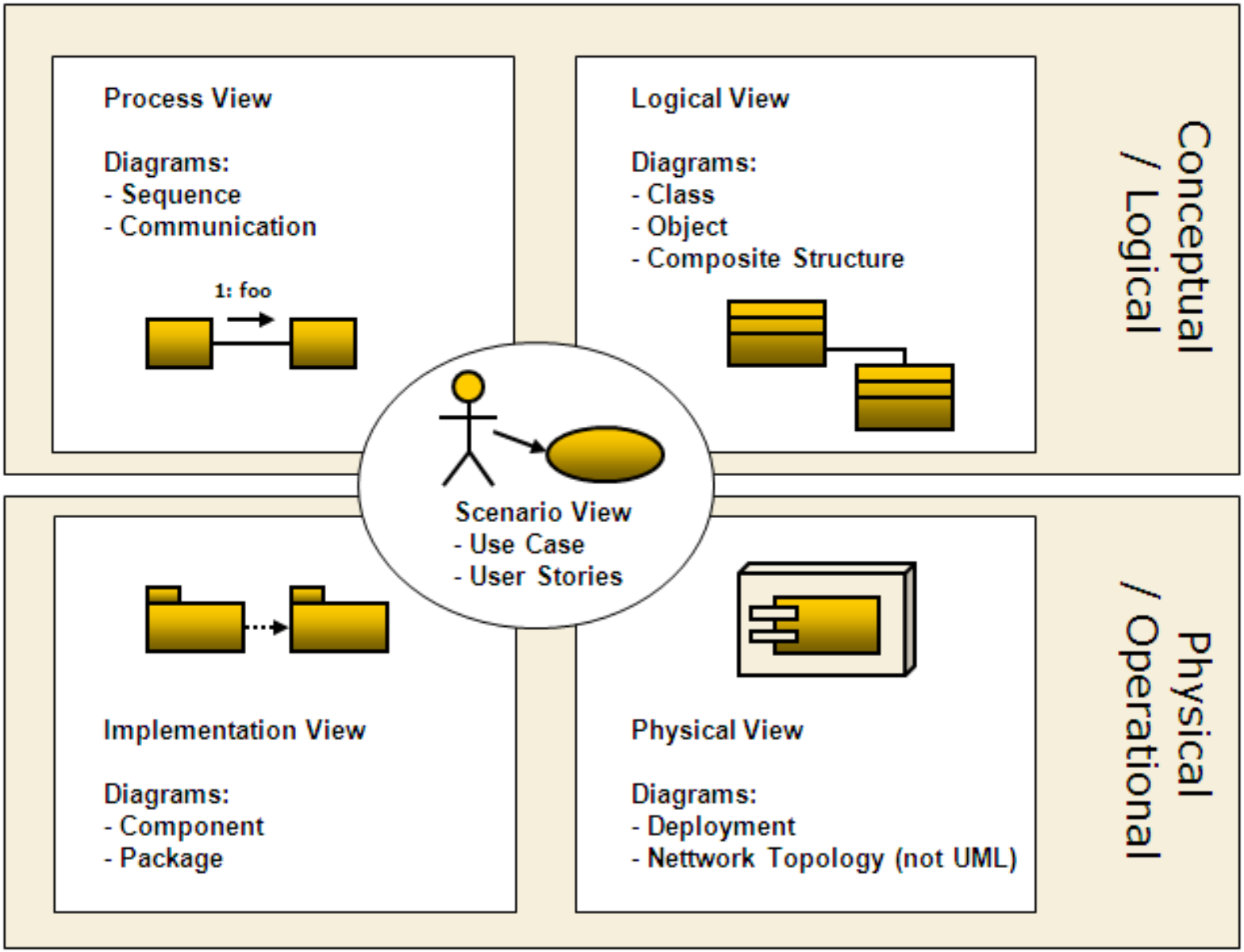
Buts

- Appréhender la logique comportementale (interactions entre objets)
- Appréhender la logique structurelle
- Documenter l'architecture

Artefacts

- Diagrammes d'interaction
- Diagrammes de classes, de packages et de déploiement
- Document d'architecture du logiciel (N+1 vues)

4+1 vues architecturales [P. Kruchten 95]



[Figure extraite de wiki.cantara.no/display/dev/4+plus+1+View+Model]

Documenter l'architecture

Objectif : Décrire les grandes lignes de l'architecture

⇝ Comprendre rapidement les idées essentielles du système

Structure du Document d'Architecture du Logiciel

- **Représentation architecturale** : Comment l'archi. est décrite dans ce doc.
- **Facteurs architecturaux** : Référence au Tableau des facteurs
- **Décisions architecturales** : Ensemble de mémos techniques
  - Nom du problème
  - Résumé de la solution
  - Facteurs architecturaux concernés
  - Solution
  - Motivation
  - Problèmes non résolus
  - Autres solutions envisagées
- **N+1 vues** : Logique, Processus, Implémentation, Physique, ..., + Cas d'util.
  Pour chaque vue :
  - Décrire les modèles les plus importants
  - Motiver et commenter les choix

Plan du cours

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
  - ⇝ Aptitude d'un objet *a* à envoyer un message à un objet *b*
    - Visibilité persistante :
      - Visibilité d'attribut : *b* est attribut de *a*
      - Visibilité globale : *b* est un Singleton
    - Visibilité temporaire :
      - Visibilité de paramètre : *b* est paramètre d'une méthode de *a*
      - Visibilité locale : *b* est variable locale d'une méthode de *a*
- Traitement des exceptions et des erreurs
- ...

En général, il doit aussi être modifié

⇝ Ajout de nouveaux attributs, méthodes, classes, ...

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

⇝ Mise-à-jour des modèles de conception pour l'itération suivante

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

⇝ Pratique popularisée par XP

Cycle de TDD :

- Ecrire le code de tests unitaires
- Exécuter les tests ⇝ Echec !
- Compléter le code jusqu'à ce que les tests réussissent
  ⇝ 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
  ⇝ 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);
assertEquals(o2,o.depile());
assertEquals(o1,o.depile());
assertTrue(p.estVide());
```

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

Refactoring régulier

Objectif :

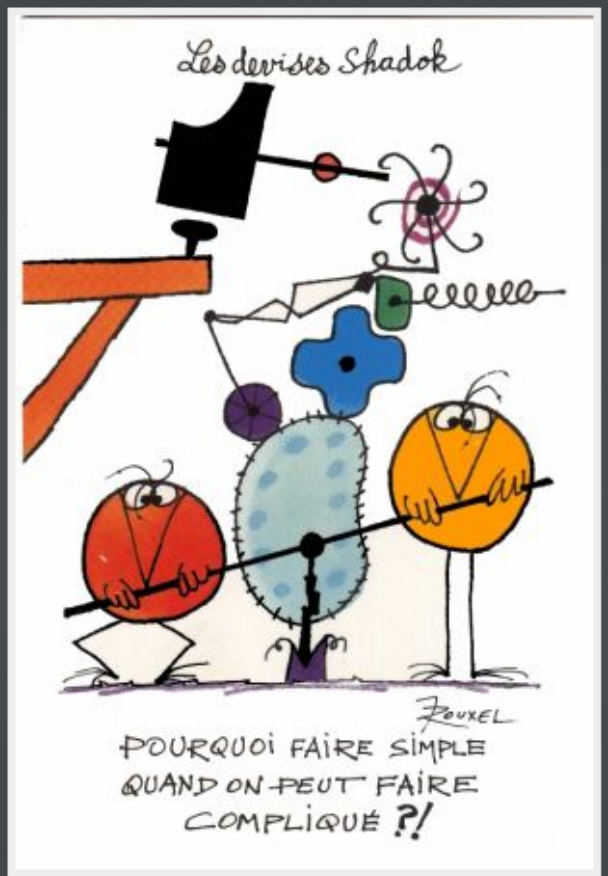
- Transformer/restructurer du code sans en modifier le comportement
  $\rightsquigarrow$ Supprimer les "code smells"
- Attention : re-exécuter les tests après chaque étape

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 du cours

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 :
$\rightsquigarrow$ Clients, 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/)

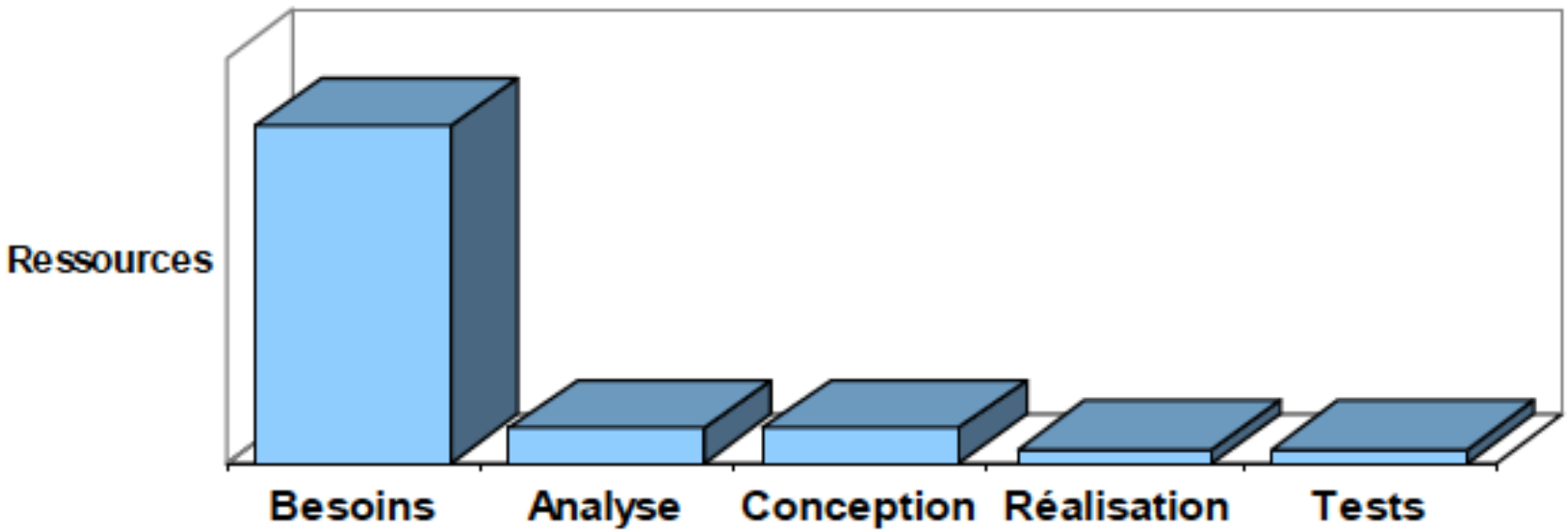
  Jusqu'à ce que volume de travail total = durée de l'itération

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

Vue globale d'une itération Agile

Plan du cours

Phase 1 : Etude préliminaire (*Inception*)



Objectif : lancer le projet

- Établir les contours du système et spécifier sa portée
- Estimer les risques les plus importants

⇝ Phase très courte (1 itération)

A la fin de cette phase on décide de continuer ou non

[Figure empruntée à J.-L. Sourrouille]

Activités principales de l'étude préliminaire

Capture et analyse des besoins

- Comprendre le contexte du système ($\sim$ 50-70% du contexte)
- Identifier les cas d'utilisation ($\sim$ 80%)
- Identifier les besoins non fonctionnels ($\sim$ 80%)
- Détailler et analyser les cas les plus importants ($\sim$ 10%)

$\rightsquigarrow$ Mieux comprendre le système à réaliser
$\rightsquigarrow$ Guider le choix de l'architecture

Analyse et Conception Orientées Objet

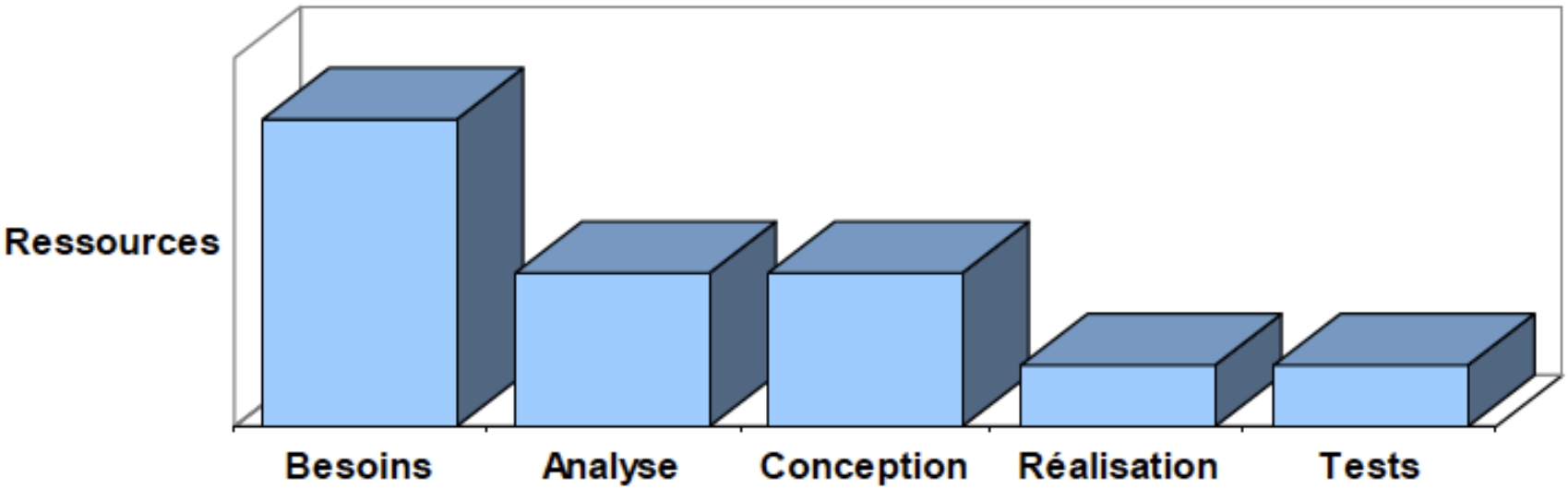
- Ebaucher la conception architecturale
  $\rightsquigarrow$ Architectures logique et de déploiement
- Examen des aspects importants et à plus haut risque

Livrables (potentiels) de l'étude préliminaire

- Liste des besoins potentiels
- Première version du modèle du domaine / métier
- Liste ordonnée de cas d'utilisation (description abrégée)
- Description détaillée de quelques cas
- Esquisse des architectures logique et de déploiement
- Liste ordonnée de risques
- Première estimation (peu fiable) des coûts et délais
- Planification de la première itération de l'élaboration
- ...

Plan du cours

Phase 2 : Elaboration



Objectif : Cerner le projet

- Identifier et stabiliser les besoins
- Atténuer ou éliminer les risques majeurs
- Planifier les phases du projet et estimer les coûts
- Établir les fondations de l'architecture
- Réaliser un squelette du système

$\rightsquigarrow$ Phase courte : quelques itérations de quelques semaines

Activités principales de l'élaboration

Capture et analyse des besoins

- Compléter la liste des cas d'utilisation
  ⇝ Description abrégée de tous
  ⇝ Description détaillée / Diag. de séquences de 40 à 80%
- Faire un prototype de l'interface utilisateur (éventuellement)

Analyse et Conception Orientées Objet

- Stabiliser la conception architecturale
  ⇝ Architectures logique et de déploiement
- Effectuer la conception des cas sélectionnés

Réalisation et test

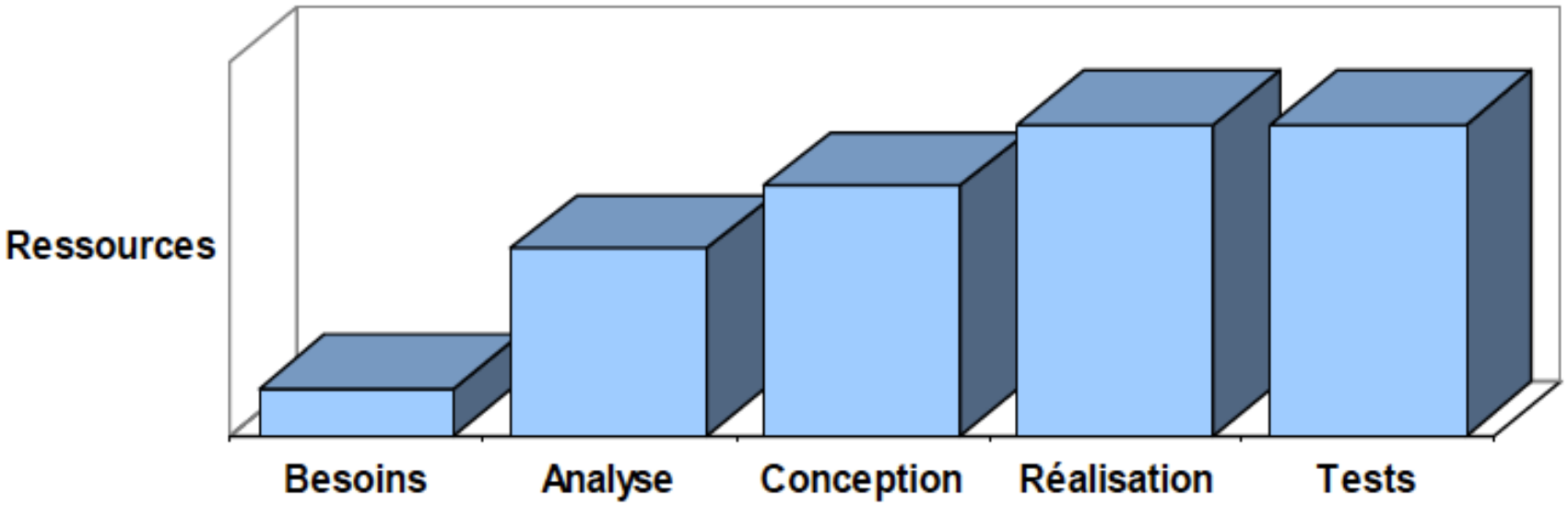
- Limités au squelette de l'architecture
  ⇝ Valider les choix

Livrables de l'élaboration

- Modèle du domaine / métier quasi complet
- Modèle des cas d'utilisation quasi complet
- Modèles de conception (diagrammes de classes et d'interactions, diagramme de déploiement, ...) partiels
- Architecture de base exécutable
- Implémentation de quelques fonctionnalités
- Document d'architecture du logiciel / modèles en cours
- Planification des phases
- Évaluation du coût du projet
- ...

Plan du cours

Phase 3 : Construction



Objectif : Réaliser une version bêta

[Figure empruntée à J.-L. Sourrouille]

Activités principales de la construction

Capture et Analyse des besoins

- Spécifier l'interface utilisateur (cf. cours IHM)
- Terminer l'analyse de tous les cas d'utilisation

Analyse et Conception Orientées Objet

- L'architecture est fixée
- Concevoir les packages et classes pour réaliser les cas
  $\rightsquigarrow$ Selon l'ordre de priorité

Réalisation et Tests

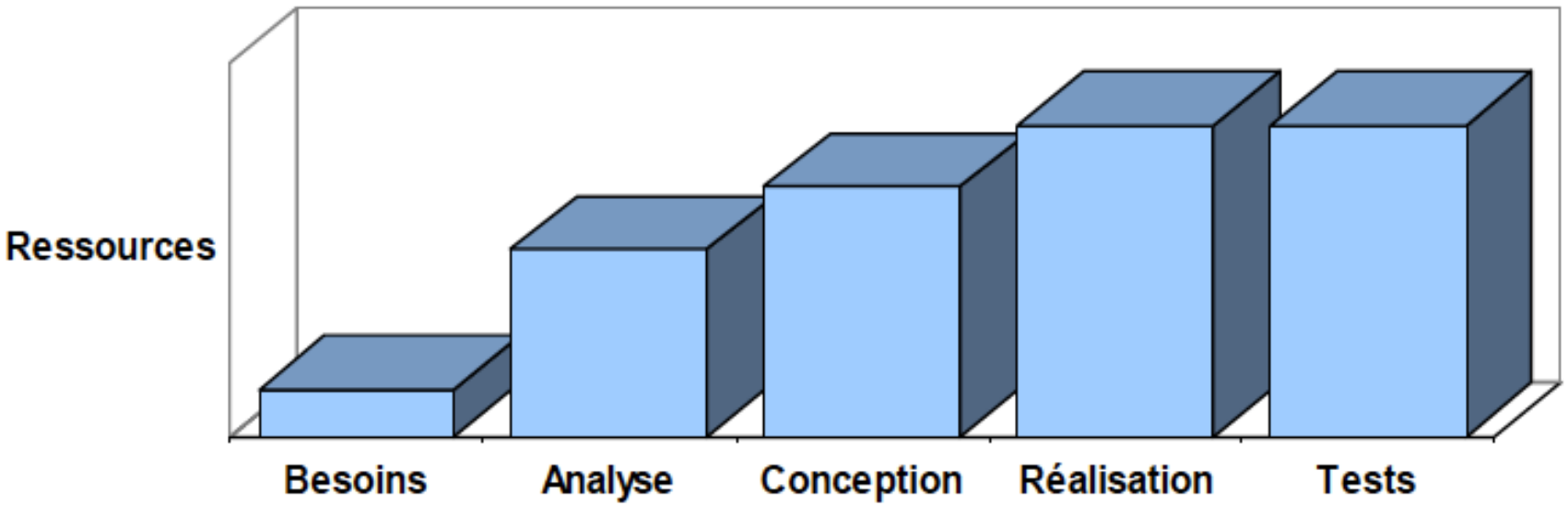
- Réaliser et tester les cas, intégrer les incréments

Livrables de la construction

- Un exécutable
- Tous les documents et les modèles du système
- Document d'architecture du logiciel à jour
- Un manuel utilisateur suffisamment détaillé pour les tests
- ...

Plan du cours

Phase 4 : Transition



Objectif : Mise en service chez l'utilisateur

- Test de la bêta-version, correction des erreurs
- Préparer la formation, la commercialisation

[Figure empruntée à J.-L. Sourrouille]

Activités principales de la transition

- Préparer la version bêta à tester
- Installer la version sur le site, convertir et faire migrer les données nécessaires...
- Gérer le retour des sites
  - ⇝ Le système fait-il ce qui était attendu ?
  - ⇝ Erreurs découvertes ?
  - ⇝ ...
- Adapter le produit corrigé aux contextes utilisateurs (installation...)
- Terminer les livrables du projet (modèles, documents...)
- Déterminer la fin du projet
  ⇝ Reporter la correction des erreurs trop importantes
- ...

⇝ Planifier le prochain cycle de développement ! ?

Livrables de la transition

- L'exécutable et son programme d'installation
- Les documents légaux : contrat, licences, garanties...
- Un jeu complet de documents de développement à jour
- Les manuels utilisateur, administrateur et opérateur et le matériel d'enseignement
- Les références pour le support utilisateur (site Web...)
- ...