

Génie Logiciel Avancé

Cours 4 — Le modèle à objets

Mo Foughali
foughali@irif.fr

Laboratoire IRIF, Université de Paris

2021–2022

URL <https://moodle.u-paris.fr/course/view.php?id=10699>
Copyright © 2021–2022 Mo Foughali
© 2016–2019 Stefano Zacchiroli
License Creative Commons Attribution-ShareAlike 4.0 International License
<https://creativecommons.org/licenses/by-sa/4.0/>



Sommaire

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 Spécification à l'aide d'UML
 - Vues de cas d'utilisation
 - Vues d'architecture
 - Vues dynamiques
 - Vues statiques
- 4 Modélisation C4
- 5 Synthèse

Outline

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 Spécification à l'aide d'UML
 - Vues de cas d'utilisation
 - Vues d'architecture
 - Vues dynamiques
 - Vues statiques
- 4 Modélisation C4
- 5 Synthèse

Le modèle à objets

Note

Ce cours suppose quelques connaissances en programmation orientée objet.

- Lorsque l'on développe un système ayant une **contrepartie physique**, une association de la forme **1 objet physique** \leftrightarrow **1 composant logiciel** peut être tentante.
- Elle peut faciliter le raisonnement et, surtout, la validation d'une spécification vis-à-vis des besoins
 - ▶ Est-ce que je conçois le bon logiciel ?
- Cette correspondance facilite également la **discussion avec un non-expert**
 - ▶ on peut utiliser un composant logiciel par son nom devant un client non informaticien.

Principes généraux de GL... dans le modèle à objets

Définition (Objet)

Un objet est formé d'un **état** et d'un ensemble de **comportements** modélisés comme des réactions à des **messages**.

Un objet a une **identité**. Sa **durée de vie** est limitée. Il joue un ou plusieurs **rôles** dans le système.

Principes :

Modularité La logique interne de l'objet est décorrélée de son utilisation.

Encapsulation La seule façon d'influer sur l'état d'un objet est de lui envoyer des messages.

Abstraction Les objets sont généralement classifiés suivant une relation de généralisation.

Les forces du modèle à objets

- En plus des apports mentionnés plus tôt, les objets facilitent un **raffinement progressif** du modèle logique à l'implémentation.
- En effet, les concepts importants du système sont souvent modélisés par des **classes abstraites** dont les sous-classes fournissent des concrétisations.
- De plus, les objets améliorent la **réutilisabilité** grâce à leur relative indépendance vis-à-vis du contexte d'utilisation.
- Enfin, l'**extension *a posteriori*** d'un composant est autorisée par le mécanisme d'héritage.
 - ▶ Cette extension n'est pas intrusive : elle ne nécessite pas de reprendre à zéro le raisonnement sur le système dans sa globalité (*separation of concerns*).

Les faiblesses du modèle à objets

Malgré son utilisation très répandue, le modèle à objet n'est pas la solution ultime au problème de la définition de composants logiciel réutilisables, corrects et robustes.

Faiblesses du modèle à objets

- ❶ La **non-transparence observationnelle** : à cause de son état interne, la réaction d'un objet à un message n'est pas toujours la même. Ceci rend difficile le **raisonnement sur les objets**.
- ❷ Le mécanisme d'**héritage** ne reflète pas la **même intention** en fonction du niveau d'abstraction auquel on se place. En effet, dans un modèle logique, l'héritage sert à refléter la relation de **généralisation/spécialisation**. Plus on se rapproche d'une spécification technique et plus cette relation est un mécanisme de **réutilisation de code**.

Les faiblesses du modèle à objets (cont.)

Malgré son utilisation très répandue, le modèle à objet n'est pas la solution ultime au problème de la définition de composants logiciel réutilisables, corrects et robustes.

Faiblesses du modèle à objets (cont.)

- ③ La notion de **message** n'est pas de première classe, ce qui rend compliquée l'expression de mécanismes calculatoires de la forme *pour tout message, ...*
 - ▶ solution partielle : *aspect oriented programming*
- ④ On aimerait parfois raisonner sur le système comme un **monde clos** en interdisant certaines extensions futures dangereuses
 - ▶ solution partielle : *final classes*

Outline

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 Spécification à l'aide d'UML
 - Vues de cas d'utilisation
 - Vues d'architecture
 - Vues dynamiques
 - Vues statiques
- 4 Modélisation C4
- 5 Synthèse

Le Rational Unified Process (RUP)

Le **Rational Unified Process (RUP)** développé par IBM est une famille de processus de développement logiciel.

- Ce sont des processus **itératifs et incrémentaux**, centrés sur le modèle à objets et sur UML (voir plus loin dans ce cours)
- Les validations de chaque phase s'appuient sur des **cas d'utilisation**.
- Le système est décrit comme la somme de multiples vues.
- Son architecture est le soucis permanent : le RUP préconise le développement préliminaire d'une **architecture exécutable**, c'est-à-dire une version du système avec un nombre très limité de fonctionnalités mais dont le "squelette" est fixé.

Les différentes variantes du RUP

- **Unified Process** (UP) est la version la plus documentée du RUP
 - ▶ C'est une version générique adaptable aux besoins particuliers.
- **Agile Unified Process** (AUP) ajoute un caractère *évolutif* à RUP
 - ▶ On s'appuie sur une **haute qualification des développeurs** pour limiter le plus possible la production de documents préliminaires au développement.
 - ▶ Les cas d'utilisation sont représentés par des **tests exécutables**. Un prototype est développé très rapidement et dirigés par la validation de ces tests.
 - ▶ La spécification est construite en même temps que le logiciel, une fois que celui-ci est confronté sous une forme exécutable aux utilisations des clients.

Perspectives du RUP

① Perspective **dynamique**

- ▶ les **phases** du processus et leur ordre temporel

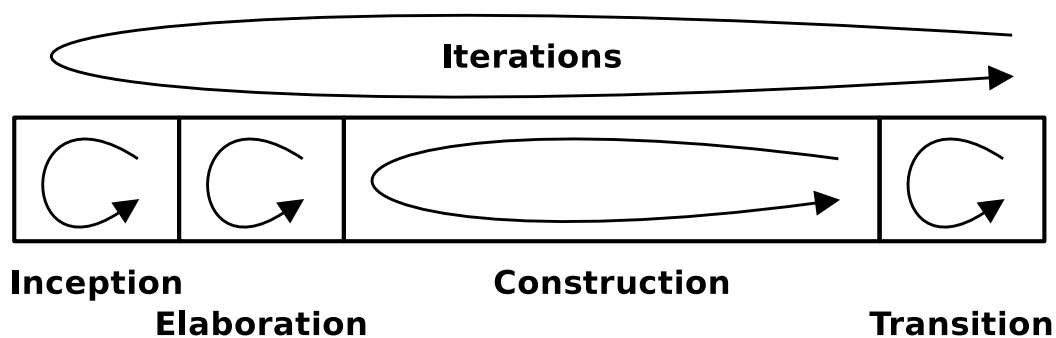
② Perspective **statique**

- ▶ les **work-flows** qui correspondent aux activités de développeurs et d'autres acteurs

③ Perspective **pratique**

- ▶ les bonnes pratiques (**best practices**) à utiliser pendant toute la durée du processus

Perspective dynamique — les phases du RUP

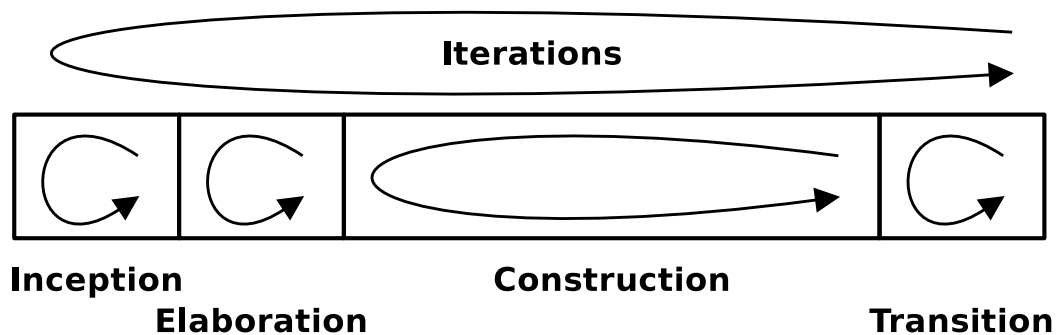


- ① initialisation (*inception*)
- ② élaboration
- ③ construction
- ④ transition

Itération

- chaque phase peut être itérée plusieurs fois avant de passer à la phase successive (**micro-itération**)
- l'ensemble de phases est typiquement itéré plusieurs fois, comme dans tous les modèles itératifs (**macro-itération**)

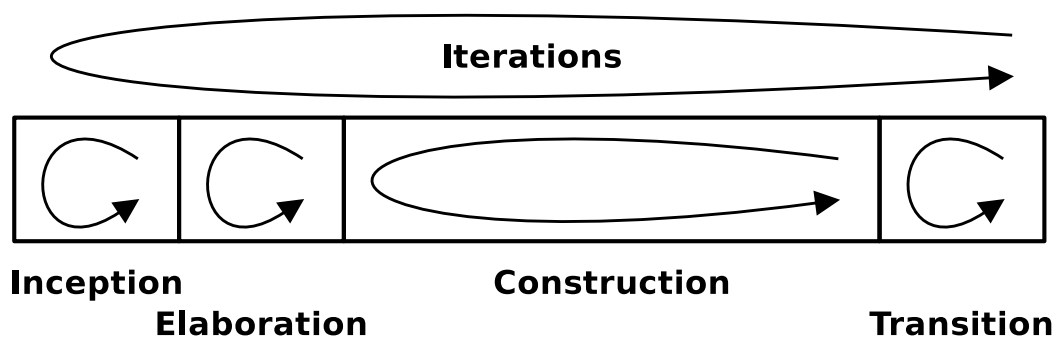
Phase initialisation (*inception*)



Cette phase correspond à l'**étude de faisabilité** classique du génie logiciel.

- établir un *business case* pour le système
- il peut amener à l'abandon du projet (donc il est plus intense vers le début du projet, pour minimiser les risques)

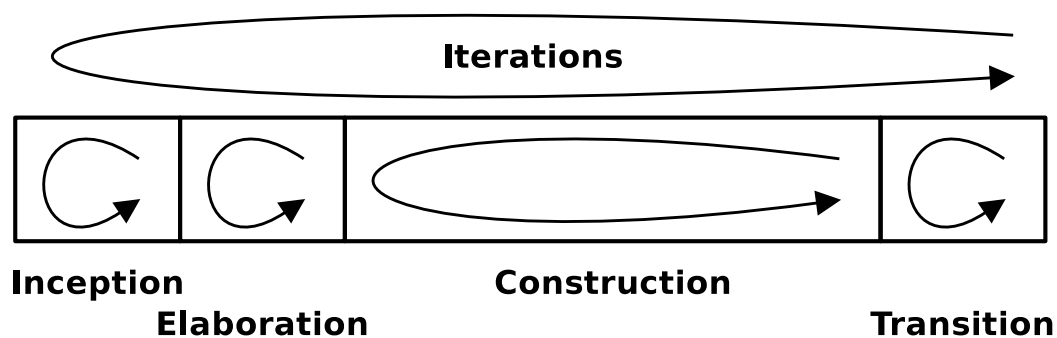
Phase élaboration



Il s'agit de l'**analyse des besoins**.

- Celle-ci fait un usage intensif des **cas d'utilisation** (et donc de scénarios) pour raffiner la compréhension du problème posé et expliciter les spécificités du domaine.
- Des **prototypes** (parmi lesquels on trouve l'architecture exécutable) sont développés pour évaluer concrètement des points techniques risqués.

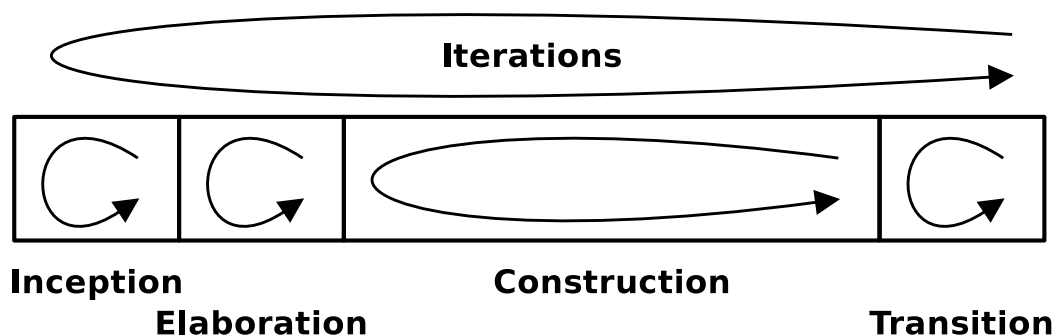
Phase construction



Cette phase correspond à la **conception** et à l'**implémentation**.

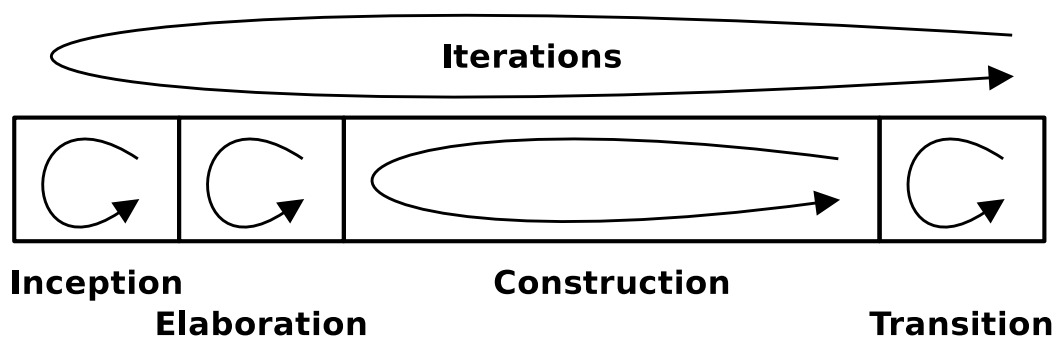
- Elle est **répétée plusieurs fois** pour une progression incrémentale aboutissant à diverses versions du système, résolvant les problèmes techniques à hauts risques en priorité.

Phase construction (cont.)



- Dans une conception orientée objet, il est parfois difficile de bien distinguer la **spécification/conception** de l'**implémentation**.
 - ▶ Certains spécialistes préconisent la définition de deux modèles disjoints : un modèle logique et un modèle d'implémentation (voir : *model-driven engineering*).
 - ▶ Cette distinction est importante car il doit toujours exister une spécification, s'appuyant sur le cahier des charges, servant de référence aux implémentations.
 - ▶ À partir des modèles logiques, le *model-driven engineering* préconise la génération automatique de code.

Phase transition



- La phase de transition de ce processus correspond à l'activité de **déploiement** et marque le début de la **maintenance** du logiciel
- Il s'agit de vérifier la mise en place du système auprès des utilisateurs (production de manuel d'utilisation, formation, ...) et de préparer ses futures évolutions.
- Cette phase a été ignorée par plusieurs modèles de développement antérieurs au RUP.

Perspective statique — les *work-flows* du RUP

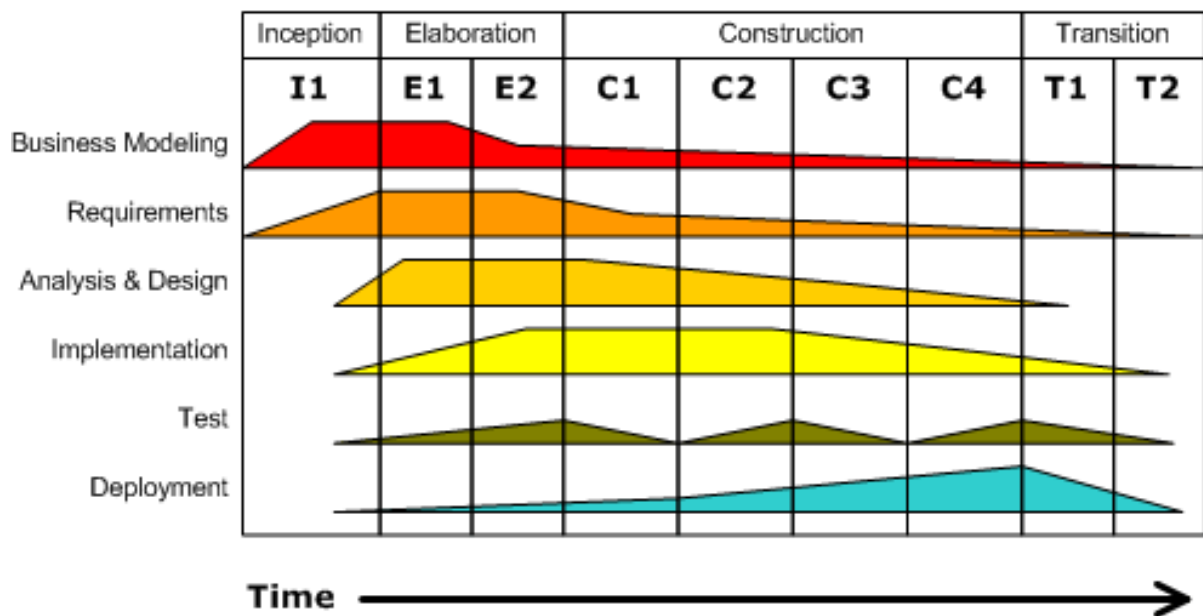
Core *work-flows*

- *business modeling*
 - ▶ i.e., modélisation du contexte du logiciel, point de vue business
- *requirements*
- *analysis and design*
- *implementation*
- *testing*
- *deployment*

Support *work-flows*

- *configuration and change management*
- *project management*
- *environment*
 - ▶ i.e., gestion des outils (de développement ou autres) nécessaires dans les différentes phases et *work-flows*

Interaction entre phases et *work-flows*



<http://en.wikipedia.org/wiki/File:Development-iterative.gif>

- les lignes correspondent aux *work-flows*
- les colonnes correspondent aux phases
- la hauteur détermine l'intensité de *work-flows* dans chaque micro-itération
- on répète tout le diagramme pour chaque macro-itération

Perspective pratique

Bonnes pratiques encouragées par RUP :

- ① développement itératif
- ② gestion explicite de requis
- ③ utilisation d'une architecture à composants (component-based software)
- ④ modélisation semi-formelle et visuelle du logiciel avec UML
- ⑤ assurance qualité
- ⑥ gestion du changement (logiciel et spécification)

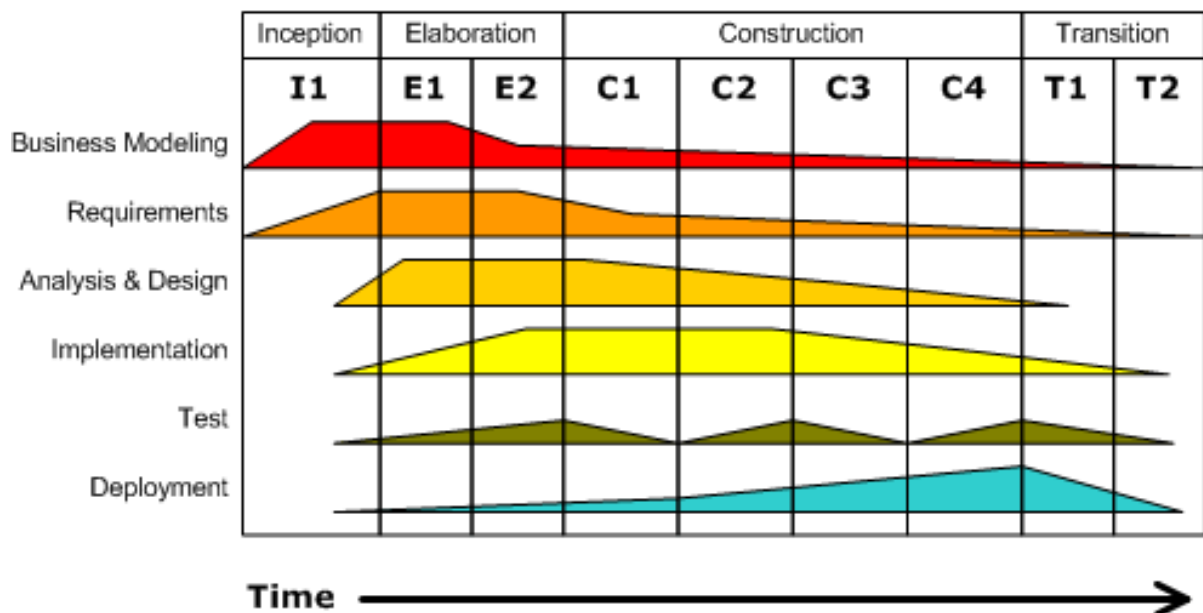
Commentaire :

- (3), (5), et (6) sont encouragées explicitement dans RUP
- le choix d'UML dans (4) est discutable et est à comparer avec d'autres langages de modélisation

Innovations du RUP

- Séparation entre phases et *work-flows*
 - ▶ chaque *work-flow* est inter-phase, comme dans la réalité d'un projet de développement
- *Work-flow* explicite pour le **déploiement** du logiciel

Ce cours



- Nous allons nous intéresser à l'utilisation d'UML pour les trois premières phases.

Outline

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 **Spécification à l'aide d'UML**
 - Vues de cas d'utilisation
 - Vues d'architecture
 - Vues dynamiques
 - Vues statiques
- 4 Modélisation C4
- 5 Synthèse

Présentation d'UML

- UML est l'acronyme de *Unified Modeling Language*.
- UML est un ensemble de notations.
 - ▶ Ces notations sont en majorité des formats de diagrammes.
- UML est standardisé par l'*Object Management Group* (OMG).
- UML est la notation la plus utilisée par l'industrie logicielle.
- La dernière version de la spécification d'UML est toujours disponible à :

<http://www.omg.org/spec/UML/Current>
(version courante : 2.5.1, Décembre 2017)

- ▶ Nous ne pourrions pas l'étudier en détail.
- ▶ Vous devez vous y référer pour écrire des spécifications qui nécessitent une conformité formelle à UML.

Critiques d'UML

Avantages

- Plusieurs modèles sont réunis : objet, orienté donnée, flot de données.
- Il existe de nombreux outils pour produire des diagrammes UML.
- C'est le résultat d'un consensus entre plusieurs « écoles » de modélisation.

Inconvénients

- La sémantique d'UML n'est pas encore fixée.
 - ▶ Toutefois, des experts essaient de définir *Precise UML*, un sous-ensemble formalisé d'UML.
- C'est seulement depuis la version 2.0 que la syntaxe est standardisée.
- Les notations sont parfois redondantes.

Différentes vues sur un système

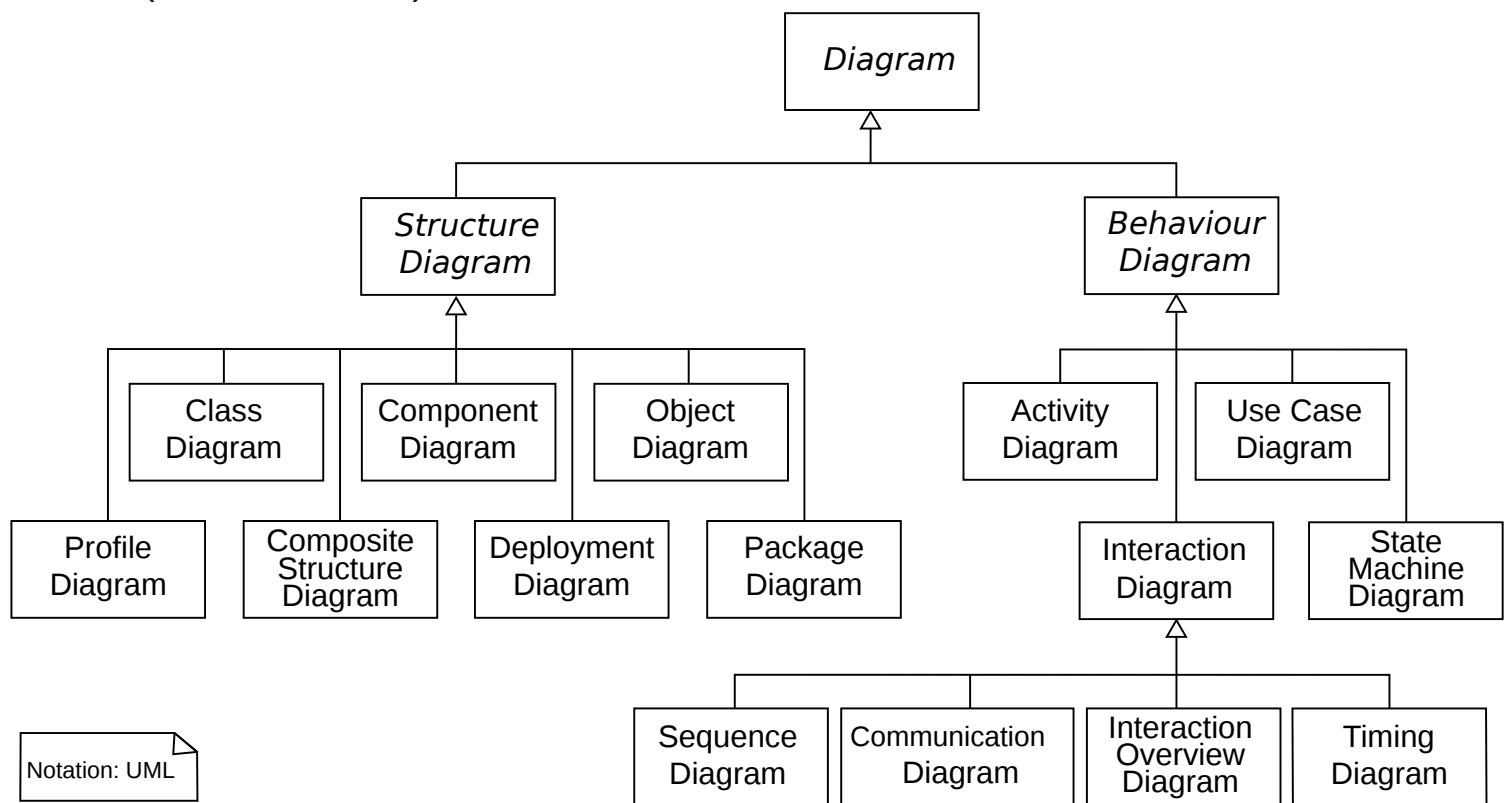
UML fournit des diagrammes pour plusieurs types de **vues** sur un logiciel.

Dans ce cours on regardera les vues suivantes :

- ① les vues de **cas d'utilisation** ;
- ② les vues d'**architecture** ;
- ③ les vues **dynamiques** ;
- ④ les vues **statiques**.

Une abondance de diagrammes ...

Les diagrammes d'UML 2.x... en syntaxe UML 2.x pour les diagrammes de classe (voir plus loin) !



http://en.wikipedia.org/wiki/File:UML_diagrams_overview.svg

Outline

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 **Spécification à l'aide d'UML**
 - Vues de cas d'utilisation
 - Vues d'architecture
 - Vues dynamiques
 - Vues statiques
- 4 Modélisation C4
- 5 Synthèse

Les cas d'utilisation

Définition (cas d'utilisation — rappel)

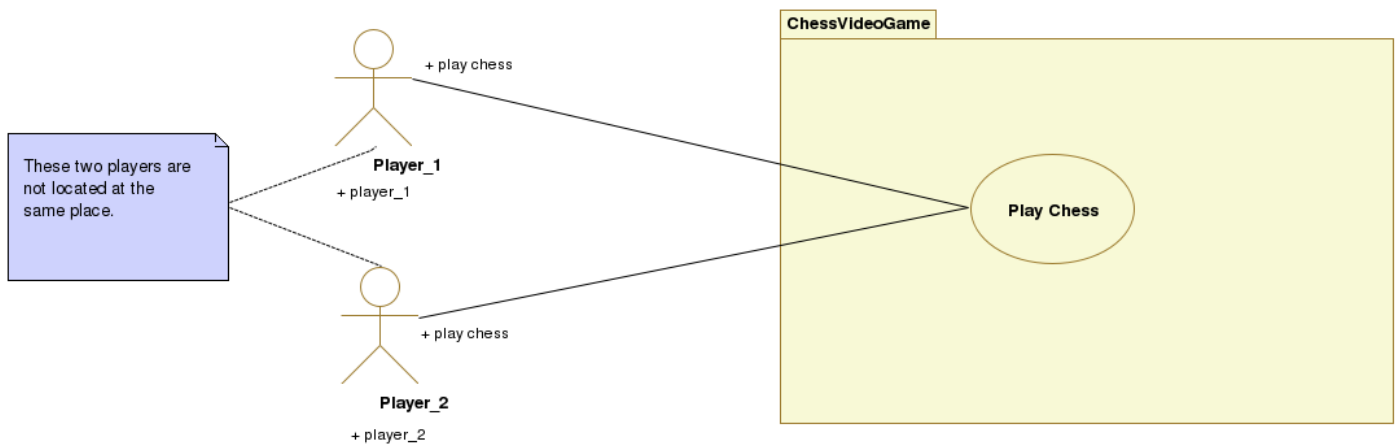
Un **cas d'utilisation** est la représentation d'une **interaction** entre le système et des acteurs en vue de la réalisation d'un objectif.

- On applique ici le principe de *separation of concerns*
 - ▶ on se focalise sur une **certaine utilisation** du système en oubliant le reste.
- En plus de réduire temporairement la complexité du système, cette unité de description est intéressante car elle est accessible aux **clients non experts**.
- Lorsque l'on suit RUP, les cas d'utilisation sont décrits par deux notations :
 - ▶ les spécifications en **langage structuré**
 - ▶ les **diagrammes de cas d'utilisation** d'UML
- Dans les méthodes agiles on représente les cas d'utilisation par des *programmes exécutables* pour pouvoir vérifier leur satisfiabilité automatiquement.

“Grammaire” des diagrammes de cas d'utilisation

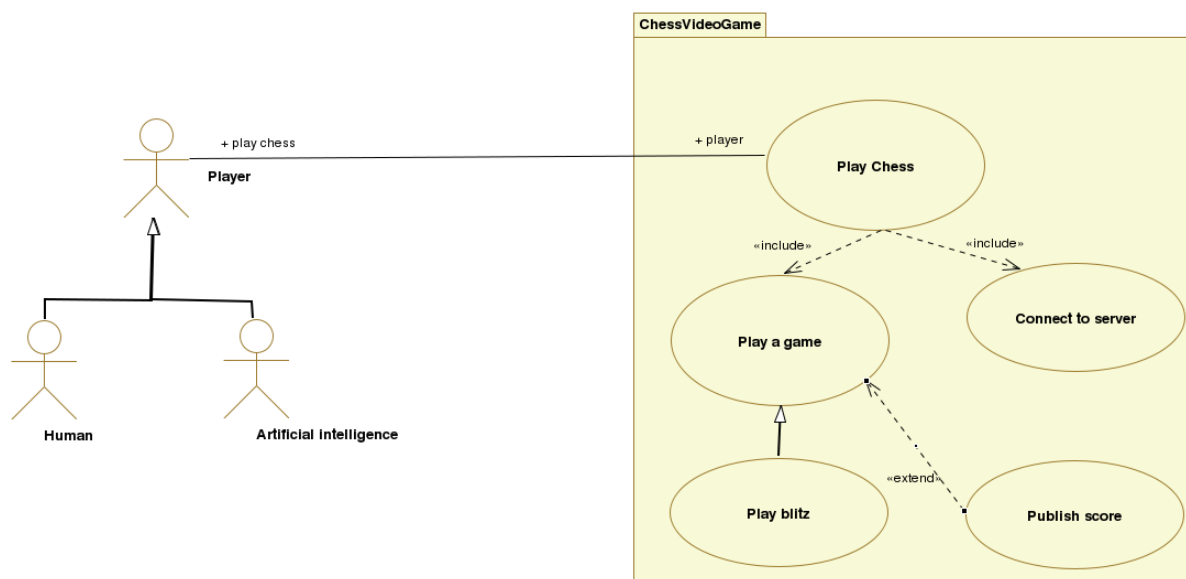
- On représente un **acteur** par un personnage schématisé.
 - ▶ Attention, cependant, un acteur n'est pas forcément un utilisateur ! (autres systèmes, API, etc.)
- Le **système** est inclus dans un rectangle (sa frontière), éventuellement étiqueté.
- Les **interactions** entre le système et les acteurs sont représentées par des lignes.
- Les **cas d'utilisation** sont des verbes à l'infinitif entourés par des ellipses.
- On représente des **relations** logiques (voir plus loin) :
 - ① entre les acteurs
 - ② entre le cas d'utilisation

Exemple : cas d'utilisation



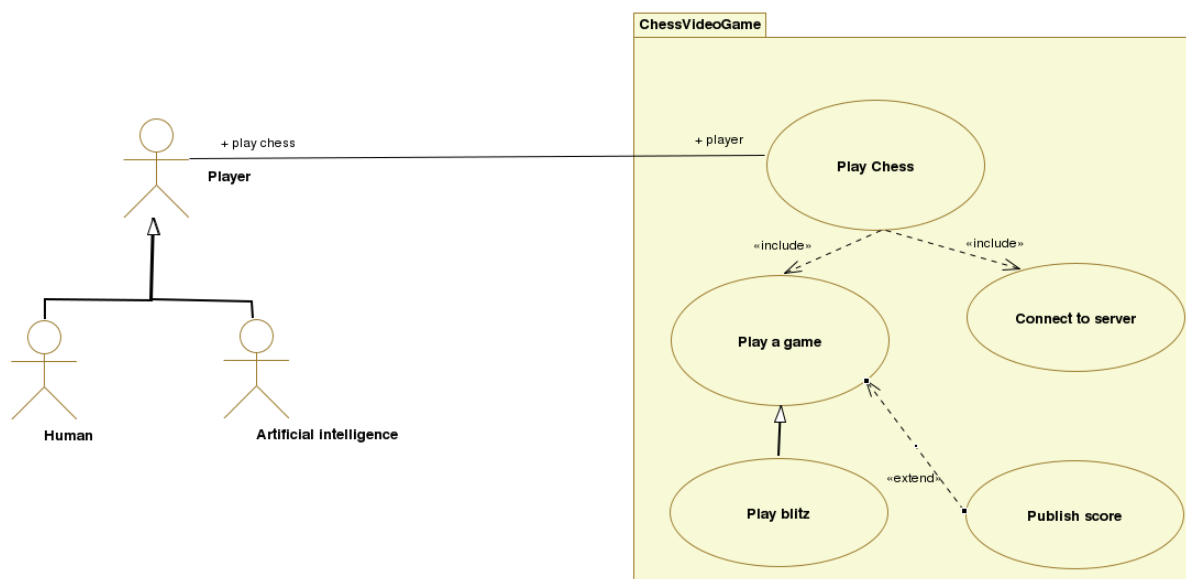
- En général, tout diagramme UML peut être annoté par un complément textuel d'information attaché à ces entités visuelles.

Exemple : cas d'utilisation (cont.)



- Une relation d'héritage permet de classifier les acteurs.
- Si l'acteur \mathcal{A}_2 hérite de l'acteur \mathcal{A}_1 alors tous les scénarii de \mathcal{A}_1 sont accessibles à \mathcal{A}_2 .

Exemple : cas d'utilisation (cont.)



- On retrouve ici les relations «étend» et «inclus» classiques entre cas d'utilisation.

Rôle des cas d'utilisation dans RUP

Ils jouent un rôle central.

- ① Ils servent de matériau de base à la phase d'élaboration (i.e., l'analyse de besoins).
 - ▶ Classifier les cas d'utilisation, en termes logiques, de priorité ou de risque, permet d'organiser l'analyse qui suit.
- ② De nombreuses vues dynamiques sont des raffinements des cas d'utilisation.
 - ▶ Ces raffinements précisent le vocabulaire et les mécanismes mis en jeu.
- ③ Si une nouvelle utilisation du système apparaît pendant l'élaboration, il est systématiquement inséré dans la base des cas d'utilisation.
- ④ La partie validation de la phase transition consiste souvent à formuler une version vérifiable/exécutable des cas d'utilisation et à y confronter le système.
- ⑤ Enfin, le manuel d'utilisation du système s'appuie très largement sur cette base de connaissance.

Activités liées à l'explicitation des cas d'utilisation

À titre indicatif, voici une succession d'activités pouvant mener à l'**obtention des cas d'utilisation** :

- ① Identification des **acteurs principaux**.
 - ▶ Les acteurs à satisfaire en priorité.
 - ▶ Les entités externes vitales au système.
- ② Identification des **cas d'utilisation principaux**.
 - ▶ On omet les situations exceptionnelles.
 - ▶ On obtient une description intentionnelle (centrée sur les objectifs).
 - ▶ On met à jour les termes et concepts incontournables du système.

Activités liées à l'explicitation des cas d'utilisation

À titre indicatif, voici une succession d'activités pouvant mener à l'**obtention des cas d'utilisation** :

- ③ Identification des **acteurs secondaires**.
 - ▶ Des acteurs qui interviennent dans les cas d'utilisation découverts.
- ④ Identification des **cas d'utilisations secondaires**.
 - ▶ Par raffinement des cas d'utilisation principaux.
- ⑤ Factorisation des **redondances**.
- ⑥ Définition du **vocabulaire** du domaine.
 - ▶ Les cas d'utilisation soulèvent des questions sur le sens des termes employés par les acteurs.

Critique des cas d'utilisation

Malgré les qualités citées plus tôt, la centralisation autour des cas d'utilisation peut avoir des faiblesses :

Taille l'énumération des cas d'utilisation et de leurs variations peut induire une combinatoire assez importante.

Conception le point de vue « utilisateur » n'est pas forcément la bonne façon d'aborder un problème.

- Par exemple, les utilisateurs peuvent avoir une vue incomplète du problème ou être des instances (inconscientes) de problèmes plus généraux.

Imprécision il est très difficile d'avoir un discours précis en s'exprimant seulement à l'aide de cas d'utilisation.

- Formaliser rapidement les concepts ou processus primordiaux permet d'en saisir les subtilités.

Outline

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 Spécification à l'aide d'UML**
 - Vues de cas d'utilisation
 - Vues d'architecture**
 - Vues dynamiques
 - Vues statiques
- 4 Modélisation C4
- 5 Synthèse

Vues d'architecture

- L'architecture est une vue d'ensemble du système.
- C'est un point de conception à haut risque.
- Il s'agit de **partitionner le système en sous-systèmes**.

- Un bon partitionnement établit :
 - ▶ une **faible dépendance** entre les sous-systèmes ;
 - ▶ affecte un **rôle clair** et distinct à chaque sous-système ;
 - ▶ permet de couvrir l'ensemble des cas d'utilisation.

Exemple : paquets

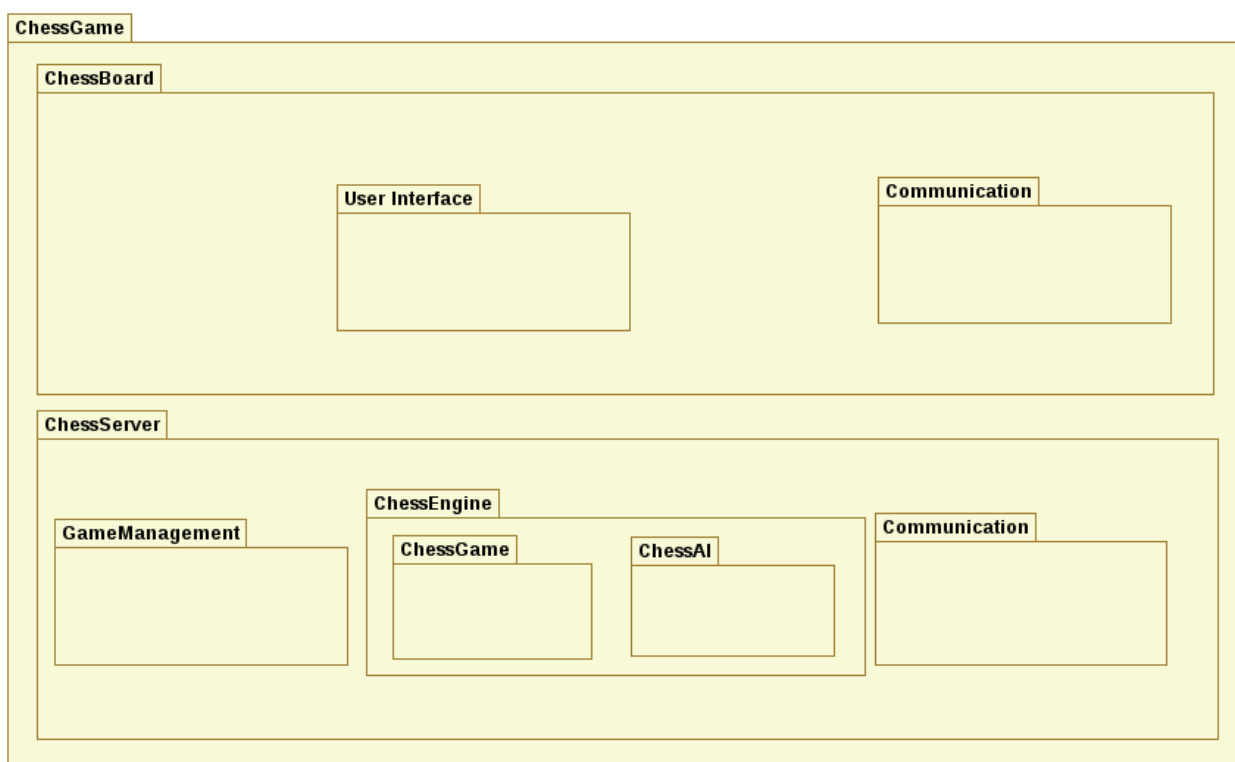


Figure – Diagramme de paquets

Diagramme des paquets : contenu

Un diagramme de paquets illustre les relations entre les différents paquets (sous-systèmes). Chaque paquet est une représentation haut-niveau d'un ensemble de classes avec leurs associations (diagramme de classes, voir plus loin).

Diagramme des paquets : relations

Les relations suivantes peuvent être définies entre sous-systèmes dans des diagrammes des paquets :

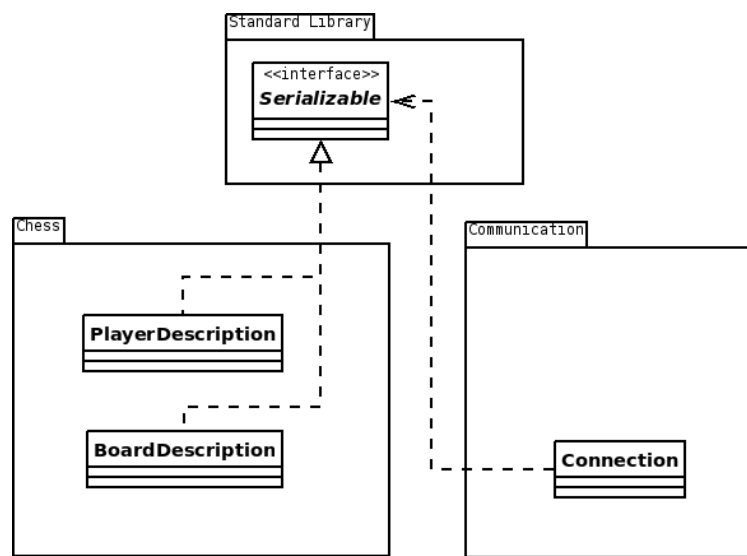
dépendance l'“utilisation” (vague. . .) d'un sous-système par un autre

import *utiliser des fonctionnalités provenant d'un autre package
(implementation counterpart : e.g. import in Java)*

merge *une sorte d'héritage entre paquets.*

La dualité import/merge (diagramme de paquets) est très similaire à la dualité extend/héritage (diagramme de cas d'utilisation)

Exemple : paquets avec classes et dépendances

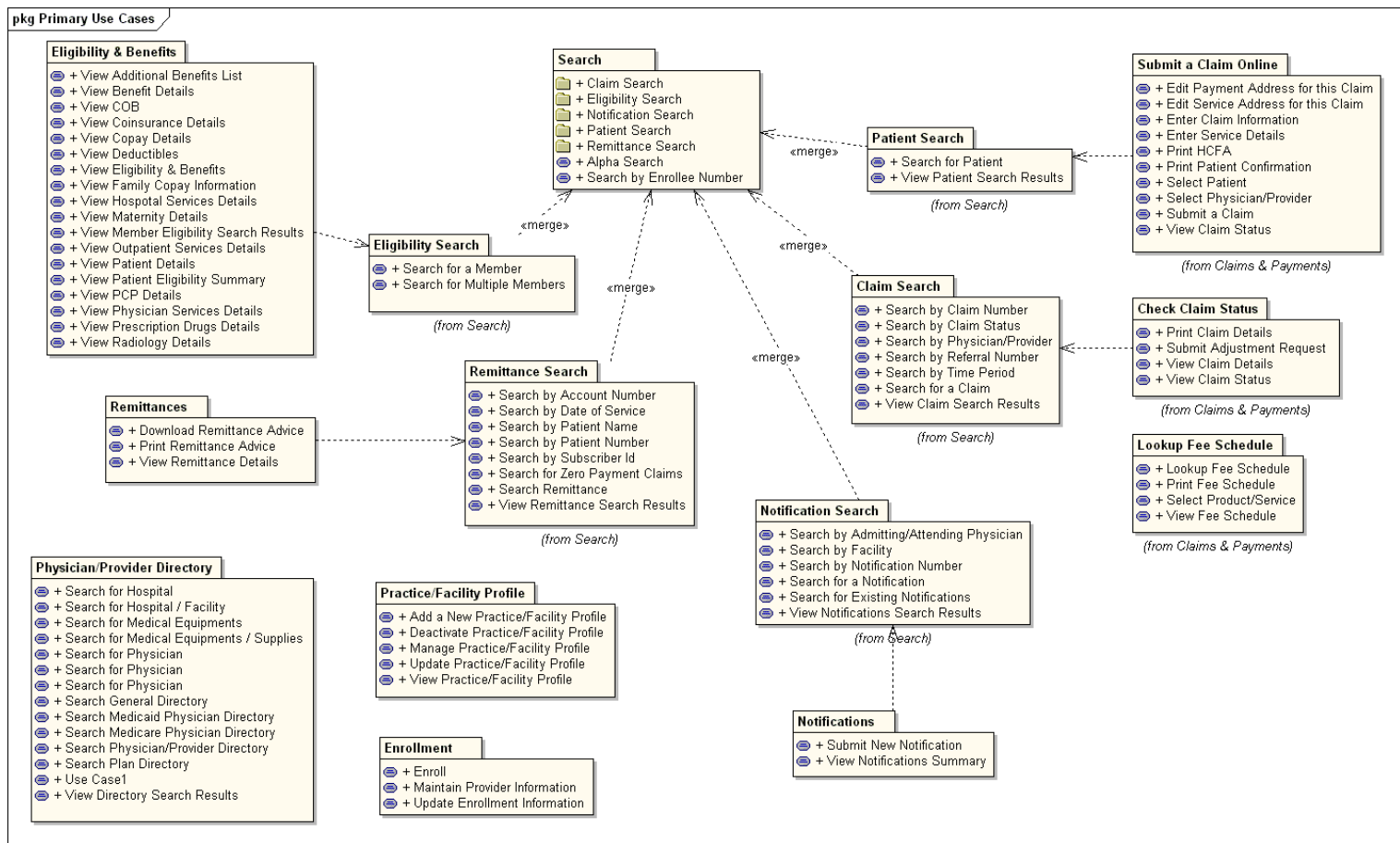


- Les « boîtes » qui apparaissent ici sont des classes importantes du système que l'on commence à classer en termes de leur appartenance à un sous-système donné.
- Les relations entre ces classes (voir plus loin) induisent des **dépendances** entre les sous-systèmes.

Effets de dépendances

- Lorsqu'un sous-système dépend d'un autre, on doit commencer par établir l'interface de ce dernier.

Exemple : avec cas d'utilisations et relations



http://en.wikipedia.org/wiki/File:Package_Diagram.PNG

Diagramme de déploiement

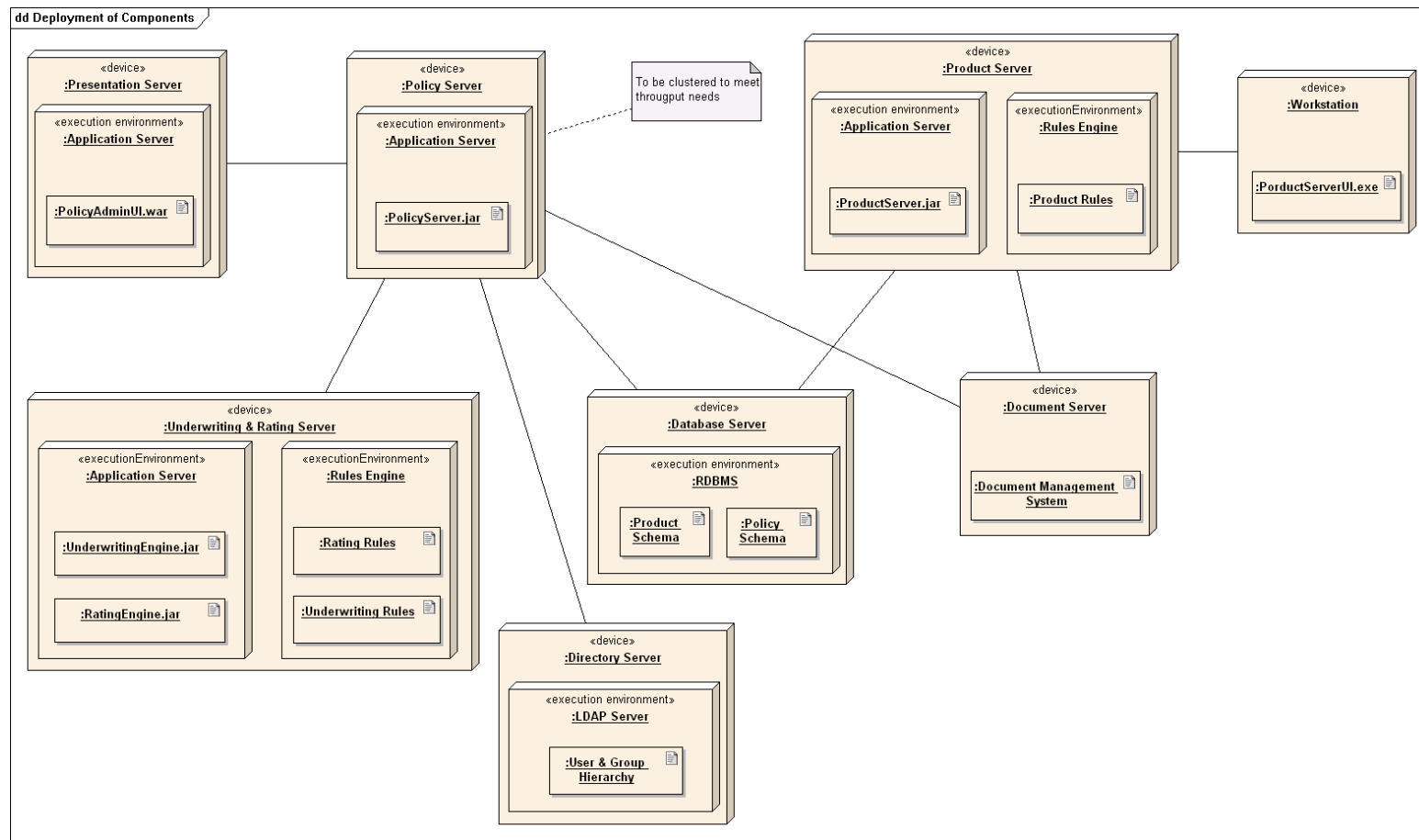
Définition (Diagramme de déploiement)

Un diagramme de déploiement (*deployment diagram*) montre le plan de **déploiement** d'un système, quand le système sera complet.

Dans un diagramme de déploiement, une association (*mapping*) entre artefacts et noeuds est établie

- les **artefacts** sont des entités “physiques” produites ou utilisées par le processus de développement logiciel (p.ex., fichiers, code objets, bases de données, documents, etc.)
- les **noeuds** sont des ressources de calcul (computational resources), sur lesquelles les artefacts peuvent être déployés (p.ex., serveurs matériels, portables), ou des environnements d'exécution (p.ex., serveurs logiciels, machines virtuelles, *containers*, etc.)
 - ▶ en général, les noeuds sont organisés dans une **hiérarchie** de noeuds

Exemple : déploiement



http://en.wikipedia.org/wiki/File:Deployment_Diagram.PNG

Outline

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 Spécification à l'aide d'UML**
 - Vues de cas d'utilisation
 - Vues d'architecture
 - Vues dynamiques**
 - Vues statiques
- 4 Modélisation C4
- 5 Synthèse

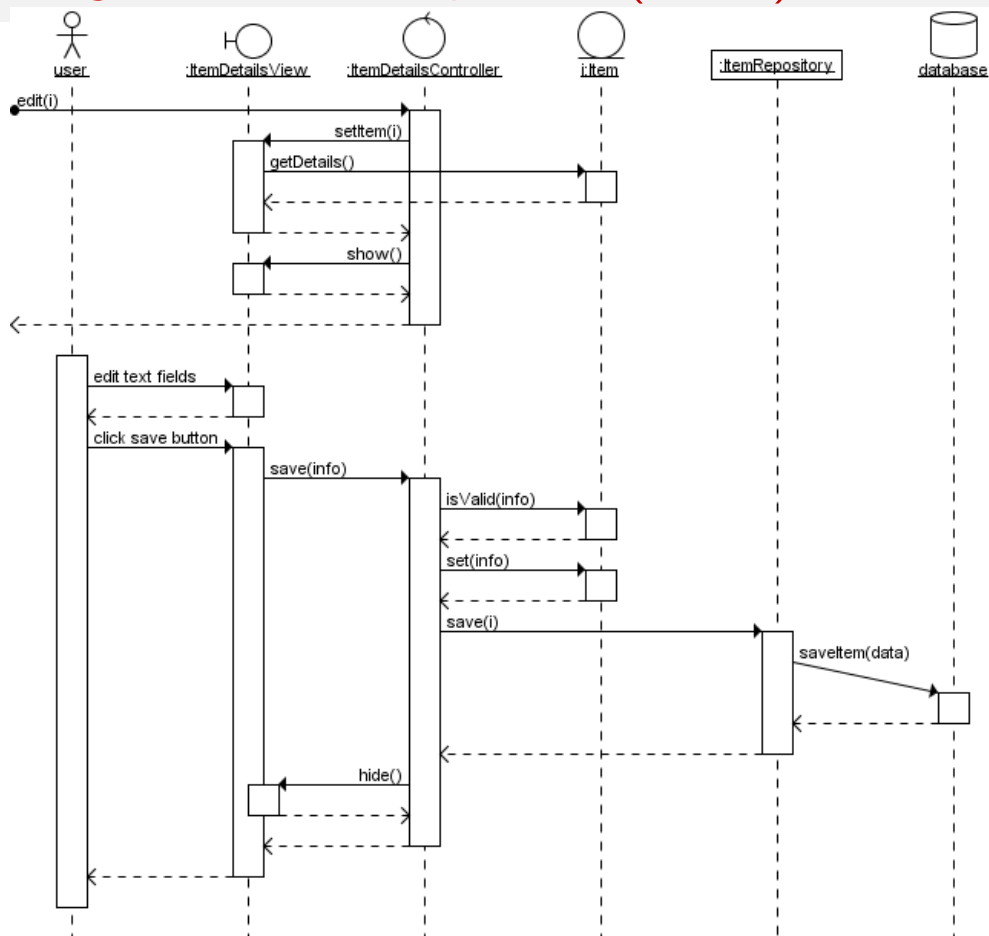
Vues dynamiques

- Les vues dynamiques décrivent le **comportement du système** (*behaviour*).
- Elles permettent de
 - ① préciser les cas d'utilisation sous la forme d'**interaction entre objets**
 - ② de **décrire l'état des objets** de façon abstraite en termes de **réactions** vis-à-vis de leur environnement et des messages qui leur sont envoyés.

Diagramme de séquence

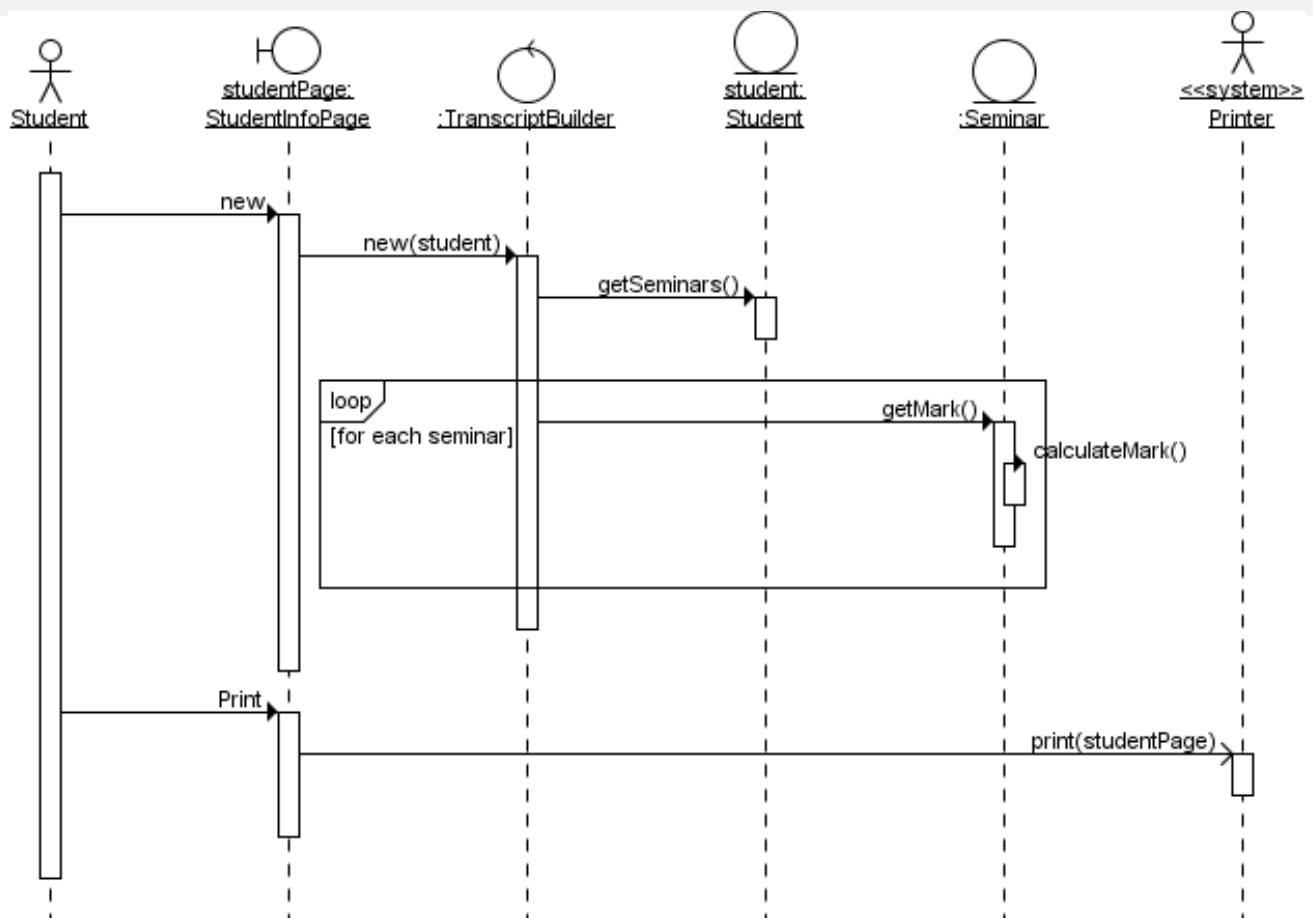
- Un diagramme de séquence présente les interactions entre les objets comme une **succession de message**
- On peut y dénoter des contraintes de réponses **synchrones** ou **asynchrones**, des états bloquants, des protocoles requête/réponse ou *fire and forget*, ...
- La **ligne du temps** est bien définie sur l'axe verticale du diagramme
- Cette notation met l'accent sur le **protocole de communication** entre les objets.

Exemple : diagramme de séquence (cont.)



<http://www.tracemodeler.com/gallery/>

Exemple : diagramme de séquence (cont.)



<http://www.tracemodeler.com/gallery/>

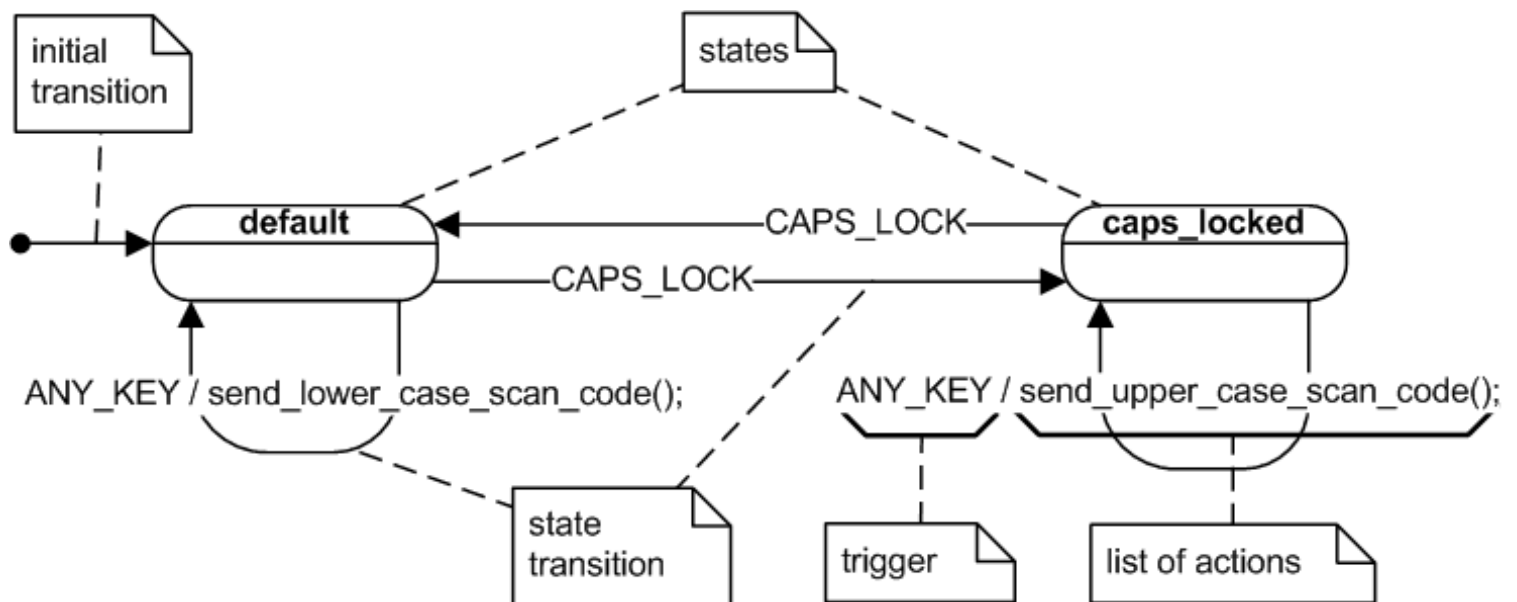
Diagramme d'état

- Les diagrammes d'état (*UML state machine* ou *UML statechart*) représentent l'**évolution de l'état** du système (ou d'un sous-système) sous la forme d'une machine à états finis (Finite-State Machine FSM).
- Une transition de cet FSM est suivie en **réaction à un événement**.
- Elle peut être conditionnée par des **contraintes** exprimées sur le système.

Intuition

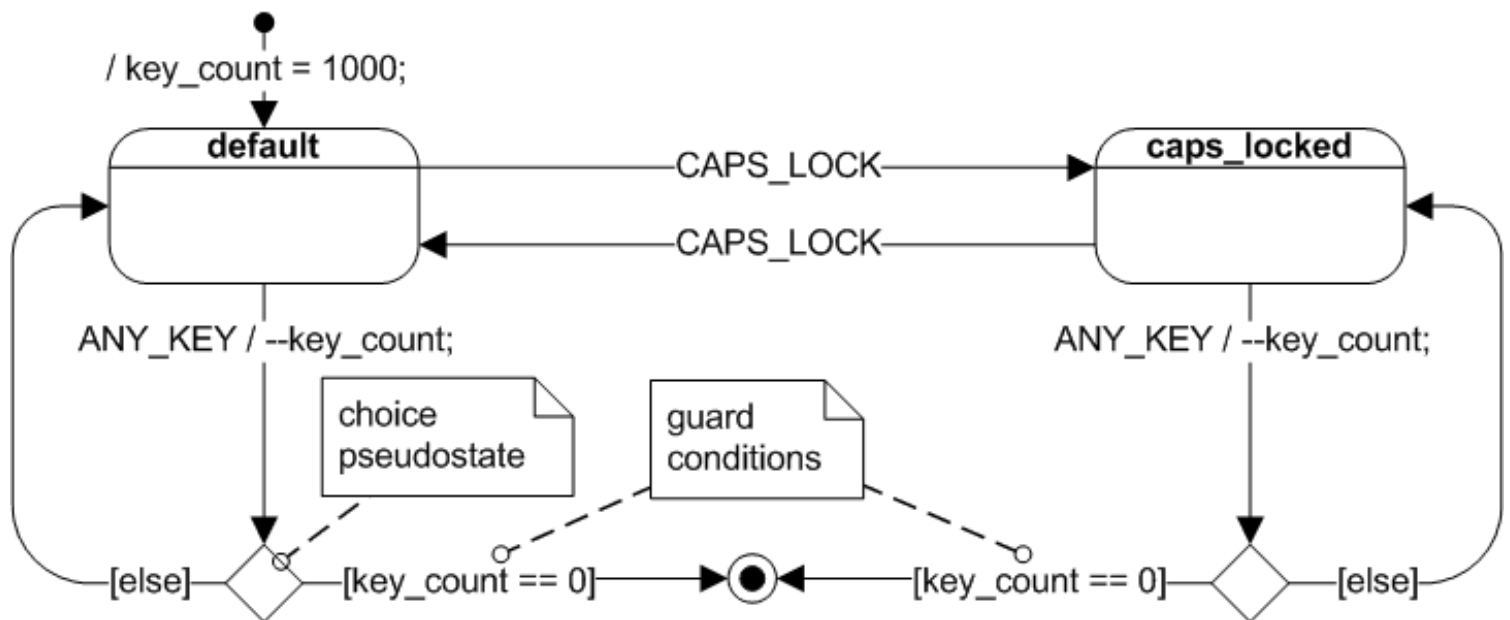
UML statechart \simeq FSM
+ gardes & opérations (FSM étendue)
+ héritage entre les états
+ ...

Exemple : diagramme d'état




http://en.wikipedia.org/wiki/File:UML_state_machine_Fig1.png

Exemple : diagramme d'état (cont.)



http://en.wikipedia.org/wiki/File:UML_state_machine_Fig2.png

Diagramme d'activité

- un diagramme d'activité modélise un **processus business** (ou *work-flow*), ses choix, et comment ils interagissent avec le contexte dans lequel le système sera déployé
 - ▶ plusieurs **systèmes** et sous-systèmes, faisant partie ou pas du système en cours de développement, peuvent apparaître dans un diagramme d'activité
- les diagrammes d'activité permettent d'exprimer **choix**, **concurrency**/synchronisation, et **itérations**
- une sémantique (en termes de réseaux de Petri) a été proposée pour les diagrammes d'activité. P.ex. :
 -  **Harald Störrle and Jan Hendrik Hausmann**
Towards a formal semantics of UML 2.0 activities.
Software Engineering 2005
<http://subs.emis.de/LNI/Proceedings/Proceedings64/article3638.html>
- originellement encodés comme diagrammes d'état dans UML 1.x, ont été séparés à partir de UML 2.x

“Grammaire” des diagrammes d'activité

rectangles arrondis activités

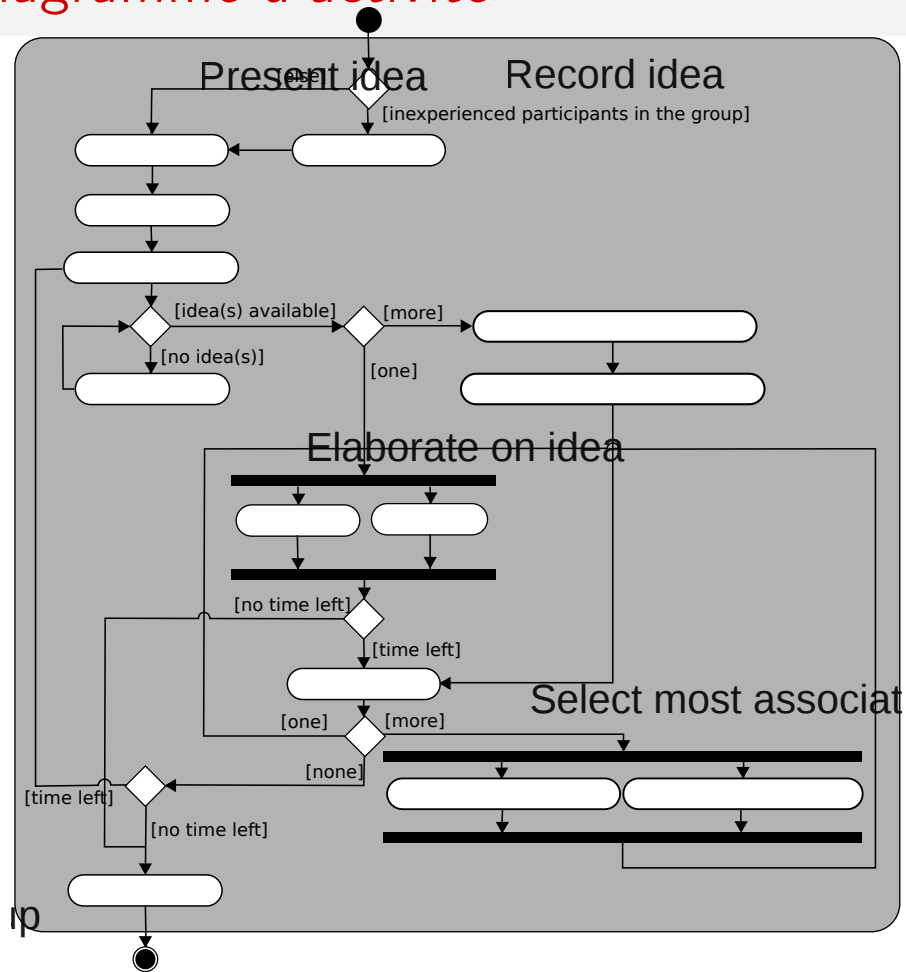
losanges choix/décisions

barres concurrence (*fork*) et synchronisation (*join*)

cercles noirs état initial

cercles noirs contournés état final

Exemple : diagramme d'activité



http://en.wikipedia.org/wiki/File:Activity_conducting.svg

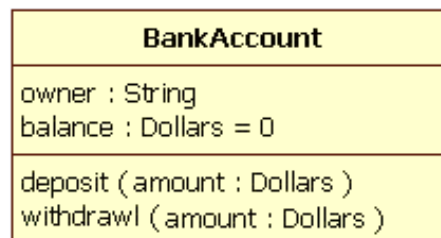
Outline

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 Spécification à l'aide d'UML**
 - Vues de cas d'utilisation
 - Vues d'architecture
 - Vues dynamiques
 - **Vues statiques**
- 4 Modélisation C4
- 5 Synthèse

Vues statiques

- Les vues statiques établissent la structure du système, à un niveau de détail plus fin que celui du diagramme de paquets.
- Il s'agit d'énumérer les différentes **classes d'objets** et **leur relations**.

Exemple : diagramme de (une) classe



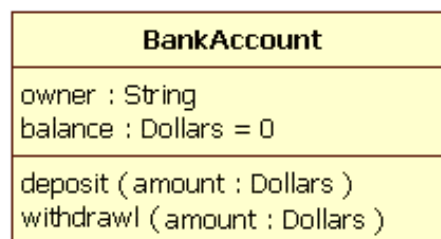
<http://en.wikipedia.org/wiki/File:BankAccount.jpg>

Figure – Diagramme de classe, réduit à une classe

on retrouve, pour chaque classe :

- ❶ **nom** de la classe (unique dans le paquet)
- ❷ **attributs** avec types (et valeur initial)
- ❸ **méthodes** avec noms et types (d'entrée et sortie)

Exemple : diagramme de (une) classe



<http://en.wikipedia.org/wiki/File:BankAccount.jpg>

Figure – Diagramme de classe, réduit à une classe

méthodes et attributs peuvent être annotés avec leur **visibilité** (*scope*) ; pour cela, UML offre des **préfixes** standardisés :

- + public (*default*)
- # protected
- private
- ~ package

Relation entre classes

Les classes peuvent être mise en **relation**.

UML propose les relations suivantes :

association un lien sémantique entre deux classes

- associations simples
- agrégations
- compositions

relations “class-level” un lien entre des entités à différents niveaux

- généralisation/spécialisation
- réalisation

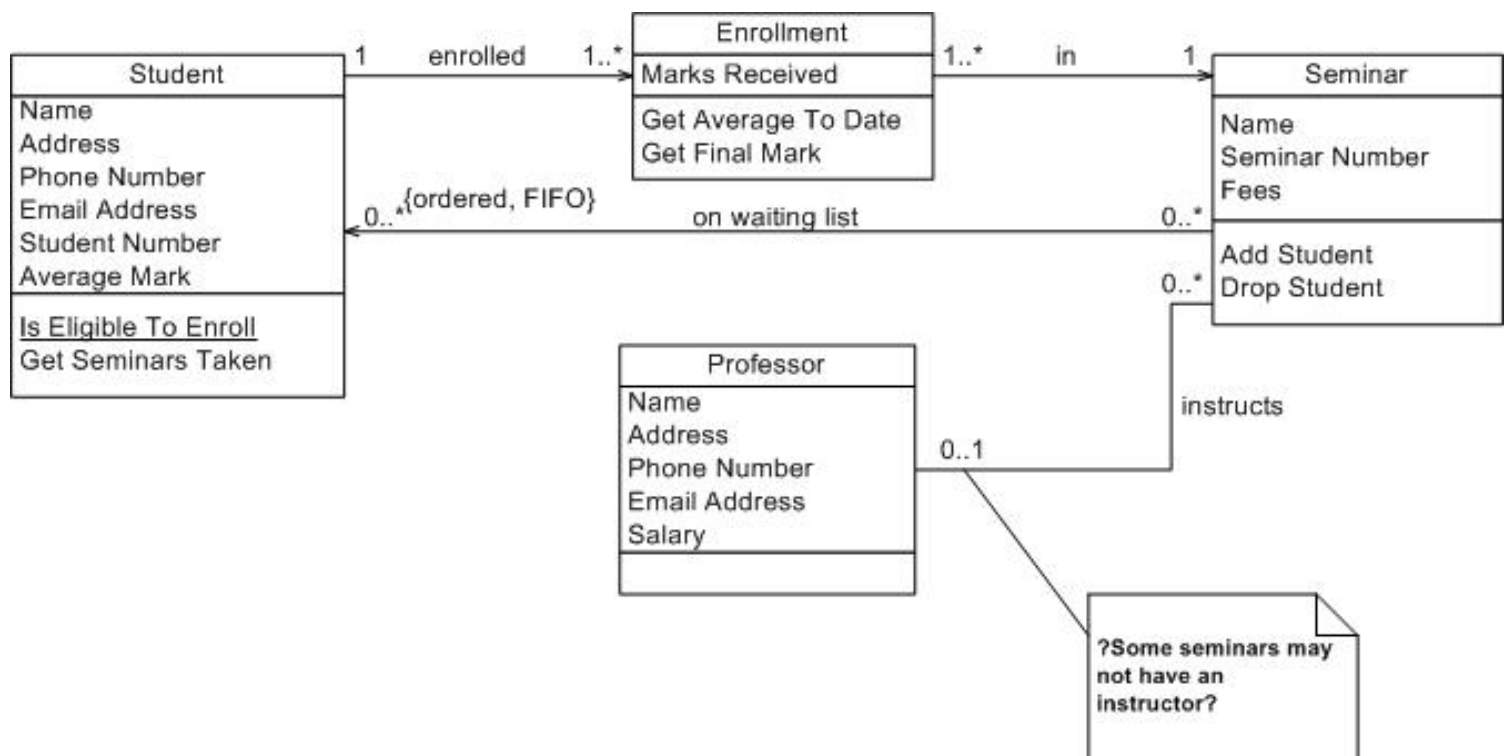
Relation d'association

- Il s'agit de la notion mathématique de relation.
- Une relation a une arité à gauche et à droite.
- Chaque objet impliqué a un rôle dans la relation.

Exemples

- Un scénario **est joué** par un joueur dans un partie.
- Une action **est applicable sur** plusieurs objets d'une scène.
- Des objets **sont nécessaires pour** autoriser une action.

Exemple : diagramme de classe avec association



Un association est formée par :

- un **nom** ;
- des **multiplicités** à gauche et à droite ;
- des **rôles** affectés à chaque objet ;
- une **direction** (bidirectionnelle par défaut).

Syntaxe des multiplicités

Un entier “ n ” n objets interviennent dans la relation.

L'étoile “ $*$ ” plusieurs objets interviennent.

Le segment “ $n..*$ ” au moins n objets interviennent.

Le segment “ $n..m$ ” au moins n et au plus m objets interviennent.

Agrégation/Composition

- L'**agrégation** est une relation d'appartenance (*has a*)
 - ▶ L'agrégation est forcément binaire et unidirectionnelle
 - ▶ Exemples (*container*) :
 - ★ Les pièces d'un échiquier lui appartiennent.
 - ★ Les joueurs d'une partie appartiennent à la partie.
 - La **composition** est une relation d'agrégation qui établit une relation de vie ou de mort d'un objet vis-à-vis d'un autre (*owns a*)
 - ▶ Exemples
 - ★ Si l'échiquier est détruit alors ses pièces aussi.
- vs
- ★ Si une partie est terminée, les joueurs peuvent en jouer une autre. (Ils survivent à la partie.)

La composition est une relation assez subtile et souvent tributaire de certains choix d'implémentation. Il est souvent préférable de ne pas l'utiliser (sinon posez-vous la question : vos classes sont-elles réutilisables?).

Exemple : compositions et agrégations

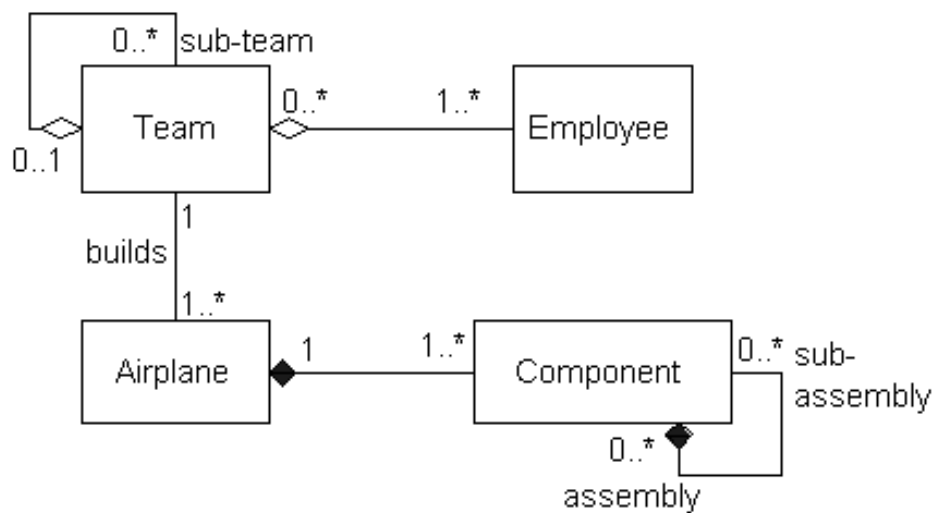


Figure – diagramme de classe avec compositions et agrégations.

- Le losange vide signifie «est agrégé à».
- Le losange plein signifie «est composé de».

Le losange est une direction implicite de l'association !

Objectifs de la généralisation/spécialisation

Spécialisation

- Ajout d'une fonctionnalité.
- Focalisation sur un aspect spécifique à une classe.

Généralisation

- Factorisation de critères communs.
- Abstraction des détails.

Analogie avec la relation d'inclusion ensembliste (*is a*).

Définition (Classe abstraite)

Une classe est **abstraite** si elle n'est jamais vouée à être instanciée.

Être abstraite capture la notion de “concept”.

Exemple : généralisation

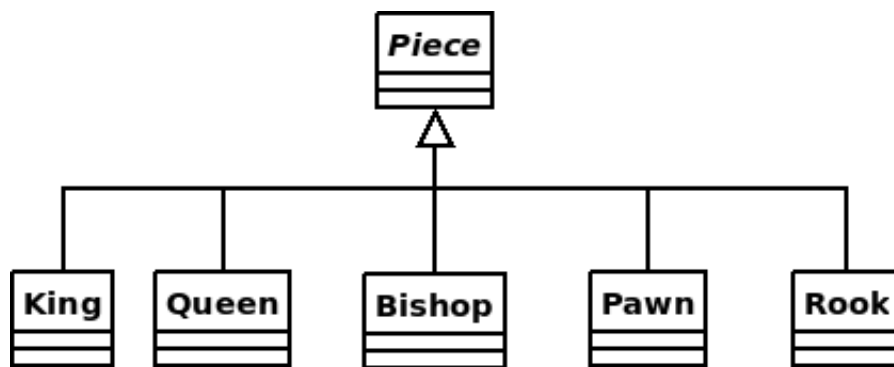
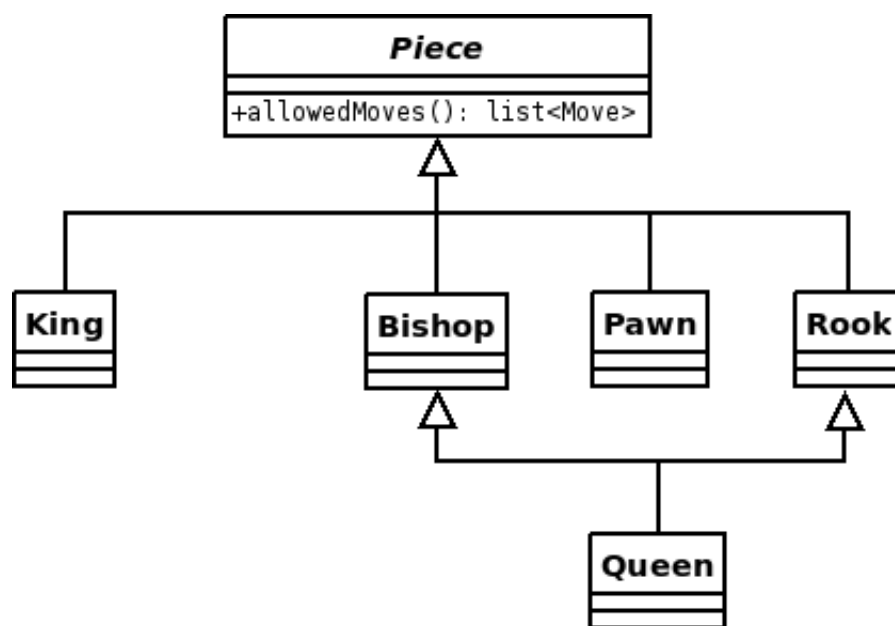


Figure – Diagramme de classe (avec relation de généralisation).

- «Piece» est une **super-classe** de «Queen», elle **généralise** cette dernière.
- «Queen» est une **sous-classe** de «Piece», elle **spécialise** cette dernière.

On exprime ici une relation d'abstraction entre composants.

Exemple : mauvaise généralisation



- La relation suivante n'est pas correcte puisqu'une reine **n'est pas un cas particulier** (*is a*) de tour et de fou (even worse, n'est pas une tour et un fou!).
- Il ne faut pas confondre généralisation et réutilisation de code.

Annotations de la relation de généralisation

On peut annoter la relation de généralisation par :

- **complete** on ne peut plus rajouter une nouvelle sous-classe.
incomplete on pourra rajouter une nouvelle sous-classe dans le futur.
- **disjoint** les sous-classes ne pourront pas être les parents de futures sous-classes.
overlap les sous-classes pourront être utilisées comme super-classes dans le futur.

Intuition : complete/incomplete s'occupent de la "extensibilité horizontale" de la généralisation ; disjoint/overlap de sa "extensibilité verticale"

Outline

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 Spécification à l'aide d'UML
 - Vues de cas d'utilisation
 - Vues d'architecture
 - Vues dynamiques
 - Vues statiques
- 4 **Modélisation C4**
- 5 Synthèse

UML ou pas ?

UML est considéré comme un langage très lourd, qui fait beaucoup (trop ?) des choses.

UML est aussi utilisé le plus souvent pour la représentation des architectures logicielles. *Si* cela est votre seul besoin, le reste de UML ne (vous) sert pas à grand chose.

Dans ce cas, d'autres langages graphiques, plus légers que UML, existent pour la modélisation d'architecture logicielles. Comme p.ex., la **modélisation C4**¹ (**C4 model** en anglais), introduit par Simon Brown entre 2006 et 2011.

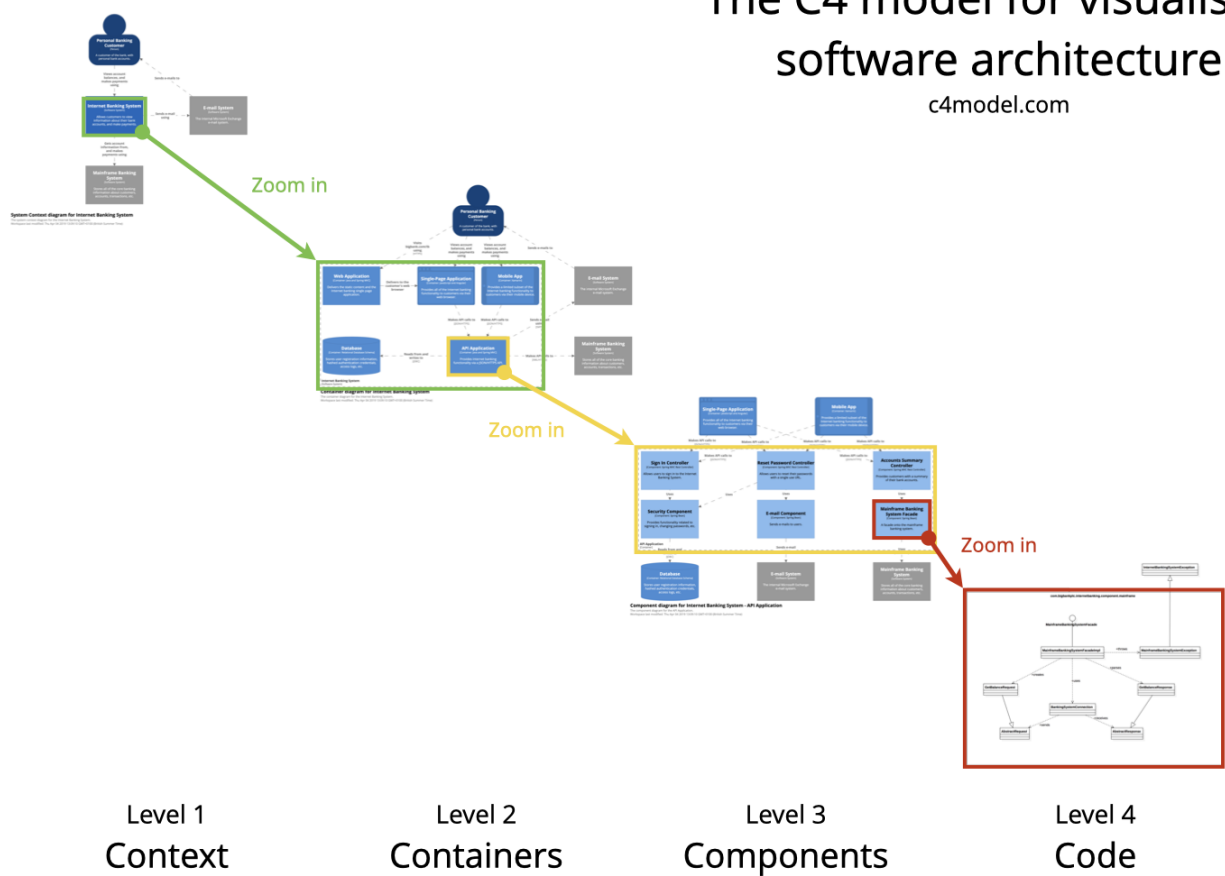
Nous allons survoler C4 dans la suite.

1. <https://c4model.com/>

C4 — vue d'ensemble

The C4 model for visualising software architecture

c4model.com



C4 — diagrammes de base

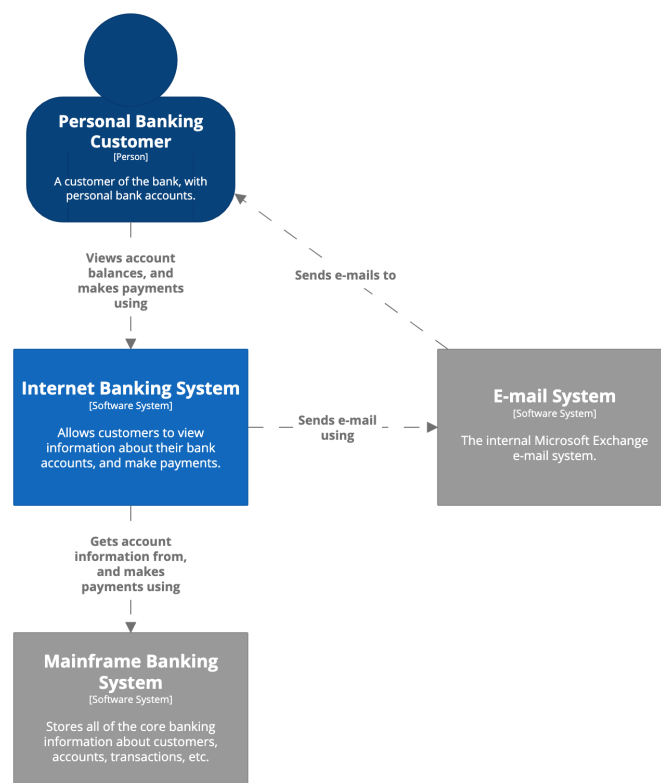
diagramme de contexte où le système se situera, quelle est sa frontière, qui sont ses proches

diagramme de conteneurs (rien à voir avec Docker !) vue de haut niveau sur l'architecture logicielle, choix technologiques principales

diagramme de composants vue d'architecture de plus bas niveau, à l'intérieur de chaque composants

diagramme de code \approx diagramme de classe, en général à ne pas utiliser, et à générer automatiquement à partir du code si vous en avez vraiment besoin

C4 — diagrammes de contexte

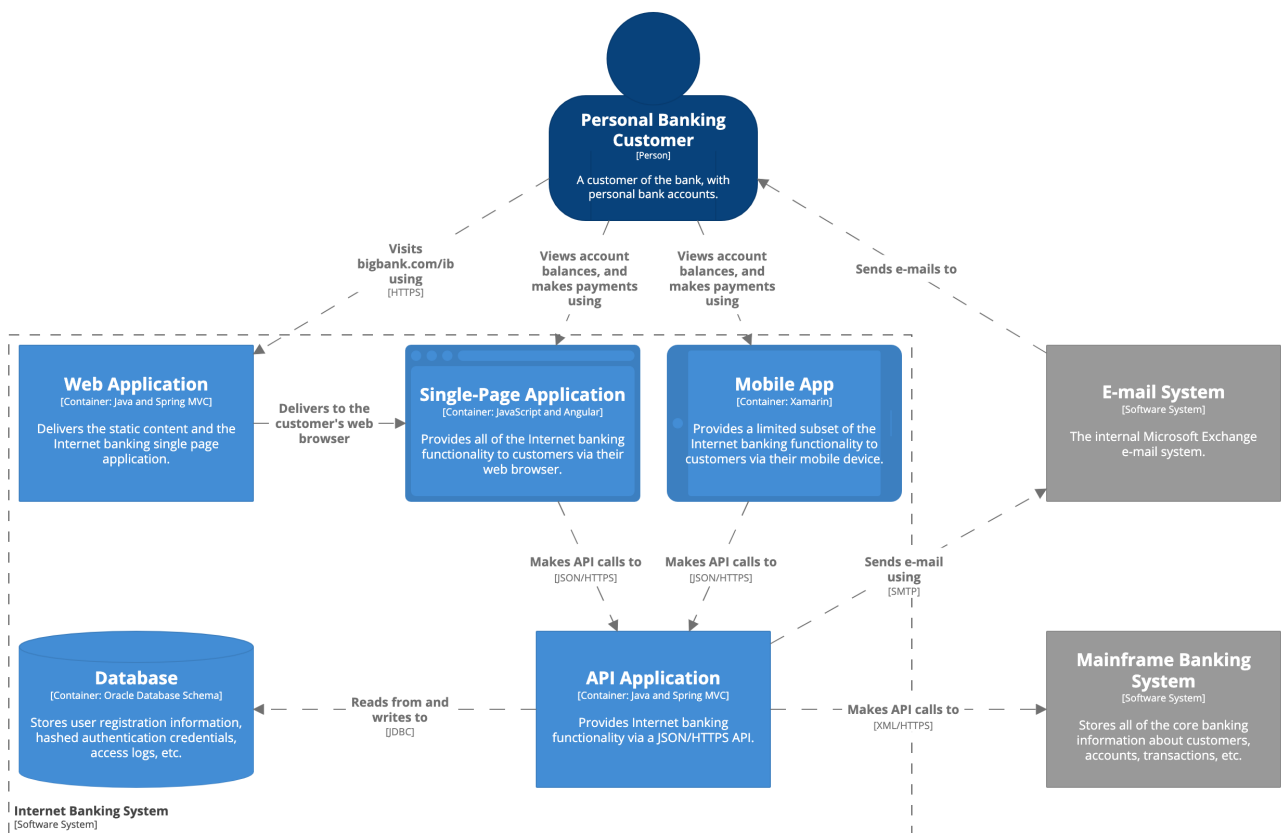


System Context diagram for Internet Banking System

The system context diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

<https://c4model.com>

C4 — diagrammes de conteneurs



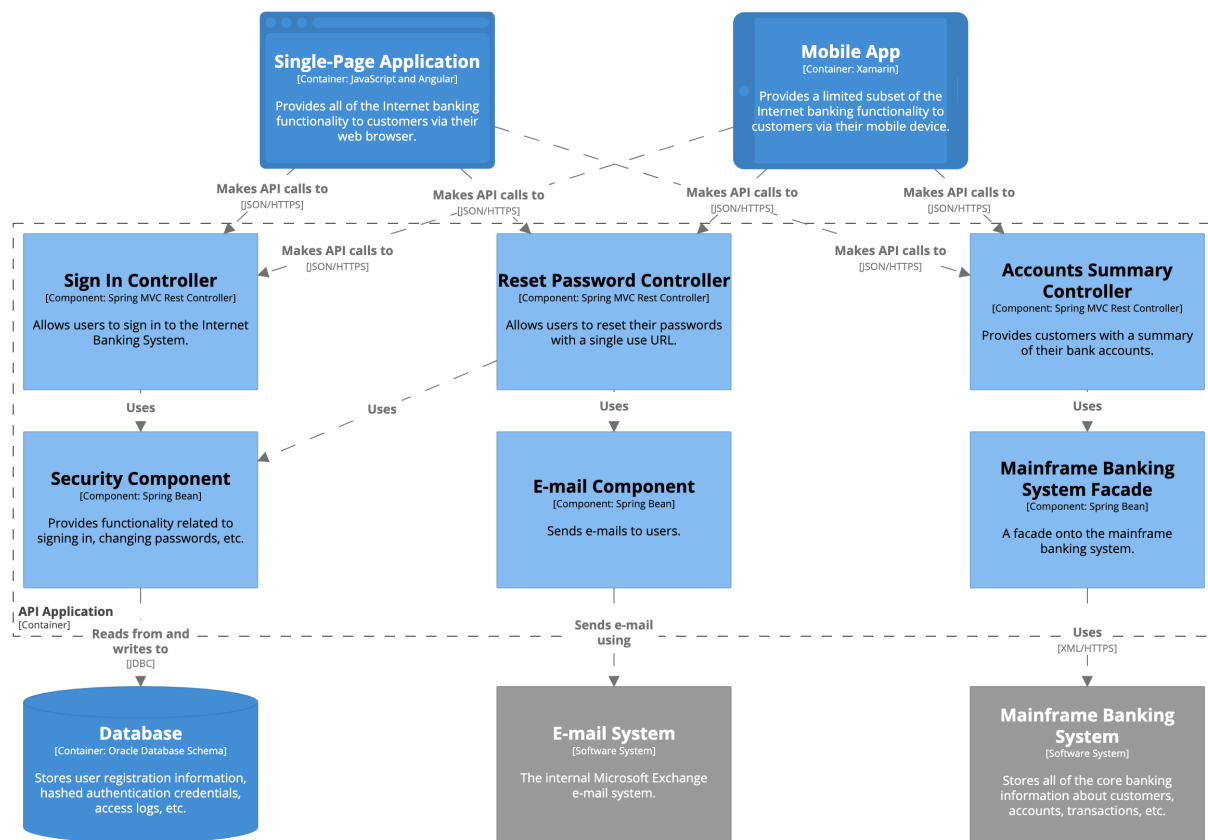
Container diagram for Internet Banking System

The container diagram for the Internet Banking System.

Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

<https://c4model.com>

C4 — diagrammes de composants



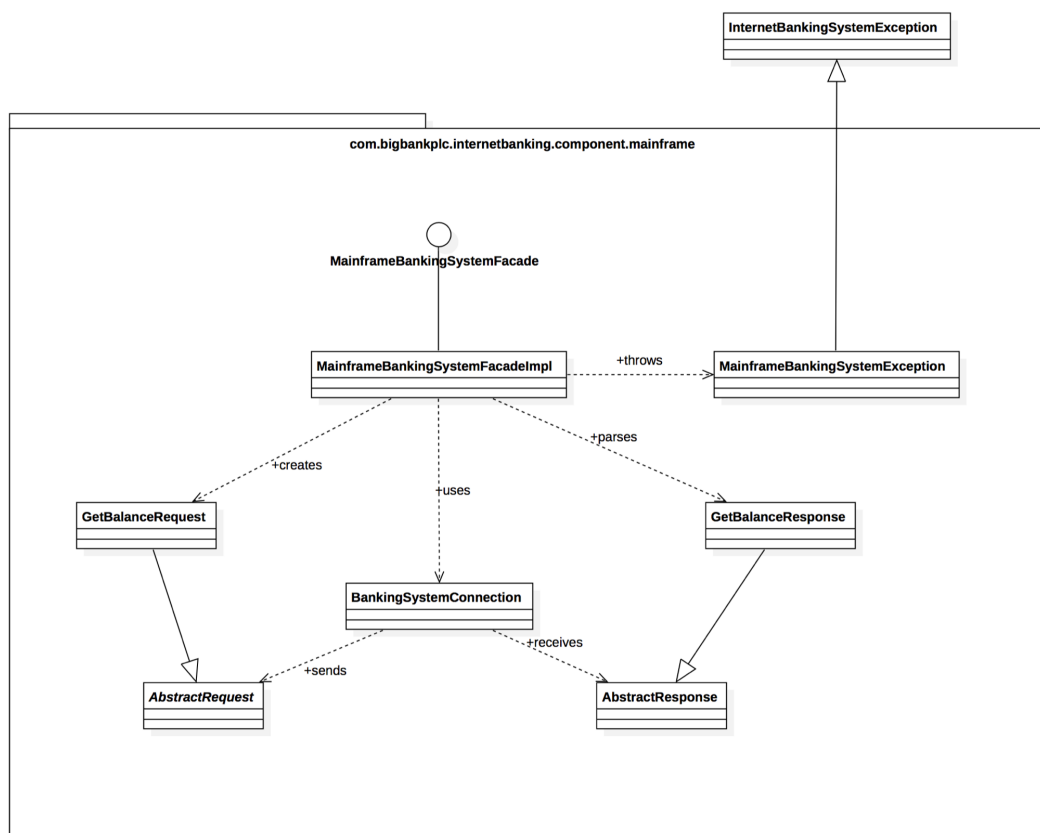
Component diagram for Internet Banking System - API Application

The component diagram for the API Application.

Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

<https://c4model.com>

C4 — diagrammes de code



<https://c4model.com>

C4 — autres diagrammes

C4 dispose aussi d'autres diagrammes, relativement peu utilisés, mais qui peuvent être nécessaires en certaines situations pour représenter d'autres vues sur votre logiciel :

- diagrammes de paysage (*landscape* en anglais)
- diagrammes dynamiques (\approx UML communication diagram)
- diagrammes de déploiement (\approx UML deployment diagram)

Pour plus de détails et exemples : <https://c4model.com>.

Outline

- 1 Le modèle à objets
- 2 Un processus associé au modèle à objets
- 3 Spécification à l'aide d'UML
 - Vues de cas d'utilisation
 - Vues d'architecture
 - Vues dynamiques
 - Vues statiques
- 4 Modélisation C4
- 5 Synthèse

Résumé

- Nous avons brièvement présenté RUP.
 - ▶ Nous en étudierons les principes dans le cours de conception orientée objet des systèmes.
- Nous avons survolé UML.
 - ▶ Ce sera un outil que nous appliquerons et approfondirons par la suite.
- Nous avons survolé C4, comme alternative légère à UML pour les architecture logicielles.