Programmation Objet : Concepts Avancés

Cours 4 : Modélisation Objet à l'aide d'UML

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

24 octobre 2008

Modélisation, Spécification et Développement

- Un logiciel est un système complexe.
- Cette complexité peut s'étendre en profondeur (di□cultés d'ordre techniques) et en largeur (grand nombre de fonctionnalités).
- L'ingénierie appliquée au logiciel informatique, ou génie logiciel, tend à rationnaliser le traitement de cette complexité en fournissant des outils et méthodes pour :
  - modéliser : analyser les besoins ;
  - spécifier : concevoir le comportement logique du système ;
  - développer : implanter (et maintenir) le système.

Notre objectif

- Il existe des processus normalisés (ISO, IEEE) de construction de logiciel.
- Ils servent à déterminer les coûts, les responsabilités, les risques ...
- Ces notions dépassent le cadre de ce cours et concernent la gestion de projet.
- Nous nous intéressons ici à l'implication de la programmation objet dans ce domaine.

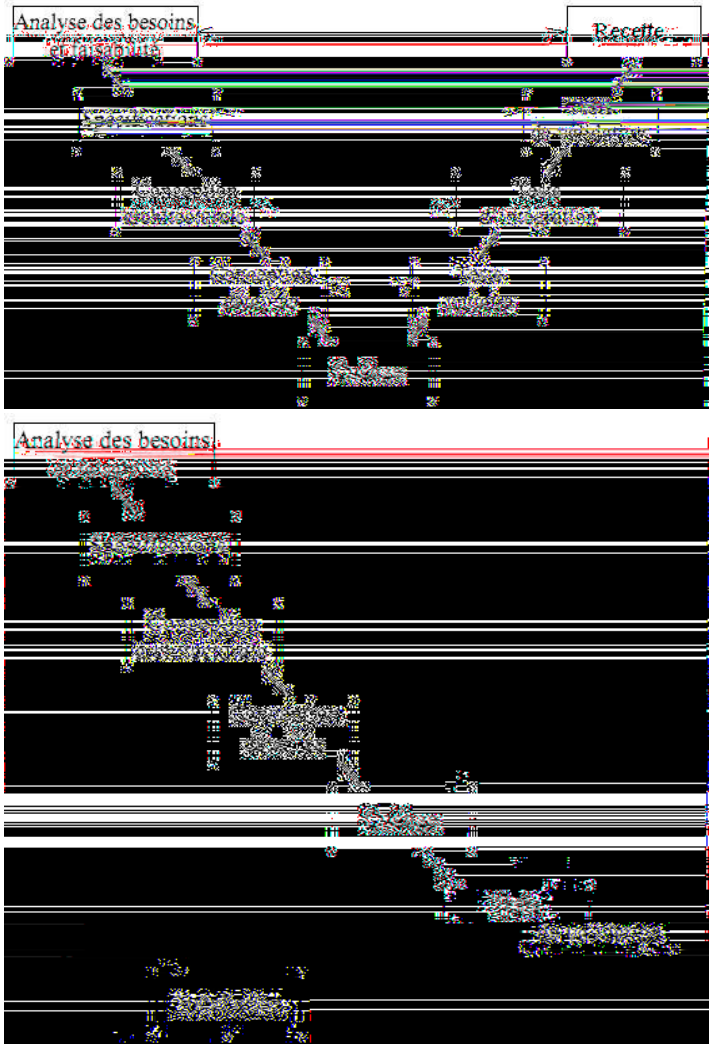
B-ABA du génie logiciel : le cycle de vie du logiciel

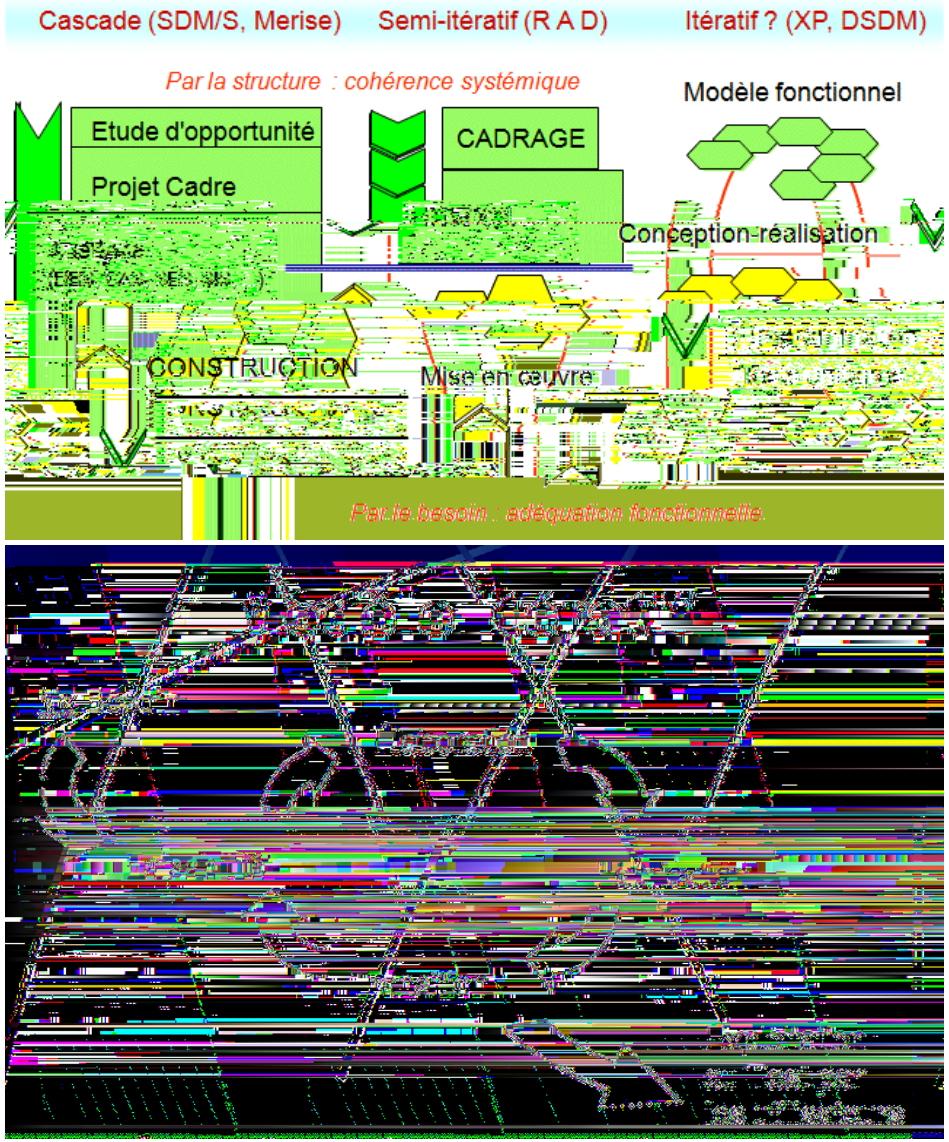
- En génie logiciel : toutes les permutations des activités de modélisation (1), spécification (2) et développement (3) sont possibles !

$\Rightarrow$ Tout dépend du logiciel à construire, des contraintes de développement ...

- Quelques exemples (informels) :
  - script de maintenance UNIX : 1 3 ;
  - script SQL de migration de schéma de base de données : 1 2 3 ;
  - développement à partir d'une idée floue : 1 3 1 3 1 3 2 3 ;
  - développement d'un algorithme connu : 1 2 3 2 3 2 3 ;
  - développement d'un robot envoyé sur Mars : 1 2 3.
- La première chose à faire est donc de déterminer votre situation !

Modèles existants





source : http://fr.wikipedia.org/wiki/Cycle_de_développement

Définition des di□érents cycles

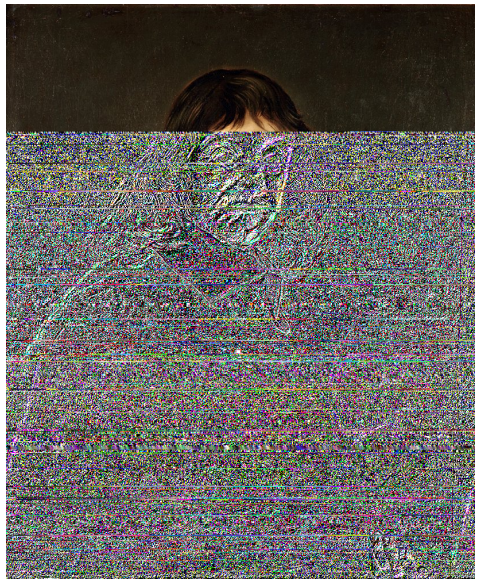
- Modèle en cascade : un unique cycle : des spécifications à l'installation.
  - Contraintes :
    - date de délivrable rigide,
    - conséquences humaines ou/et financières intolérables d'une erreur en amont.
  - Méthodes : raffinement descendant.
  - Exemples : Z, B, ...
  - Utilisations : Robot sur Mars, transport automatique, ...
- Cycle en V, en spirale : plusieurs itérations
  - Contraintes :
    - meilleure réactivité pour la correction des erreurs ;
    - évaluation des risques à chaque itération.
  - Méthodes : globalement descendantes avec retour anticipé sur la réussite de chaque étape.
  - Exemples : MERISE, ...
  - Utilisations : logiciel à cycle de développement long (progiciels, système d'exploitation, ...).

Définitions des di□érents cycles

- Di□culté : la tension entre réactivité et passage à l'échelle.
- Cycle (semi-)itératif : modulariser le développement en sous-produits.
  - Contraintes : réactivité essentielle.
  - Méthodes : cycle de développement très court, ajustement des efforts, prise de risques importante.
  - Exemples : *extreme programming*, RAD, RUP, ...
  - Utilisations : secteur à innovations (carte à puce, WEB, ...)

$\Rightarrow$ Un langage/modèle de programmation favorisant la réutilisabilité et la sûreté du logiciel est un avantage dans ce cadre.

Remarque sur la méthode



- Une méthode, ce n'est pas à moyen mécanique de limiter la réflexion.
- Au contraire, on se donne une méthode pour conduire sa réflexion de manière rigoureuse.
- Chaque projet, chaque individu, est un cas particulier.

Conception orientée objet

Les objets pour une conception itérative et incrémentale

- Les programmations par modules et par objets favorisent :
  - l'extensibilité
  - l'encapsulation (la localité des raisonnements)
  - la réutilisabilité
  - l'abstraction
- Ces caractéristiques augmentent l'indépendance des composants à construire et donc des tâches à e□ectuer.
- Des processus en tirent parti pour augmenter la réactivité du développement.

Étude de cas : un jeu d'échec

- Nous allons modéliser un jeu d'échec à l'aide du *Unified Process*.
- Ses caractéristiques générales sont :
  - UP est à base de composants (objets, modulaires)
  - UP utilise UML
  - UP est piloté par les cas d'utilisation
  - UP est centré sur l'architecture
  - UP est itératif et incrémental
- Cette méthode utilise la notation semi-formelle UML (*Unified Modelling Language*).

Décomposition du processus en phases

- La conception est partitionnée en ces di□érentes phases :
  1. Création : à quoi sert notre système globalement ?
  2. Élaboration : quel est son architecture ?
  3. Construction : par quels composants est peuplés cette architecture ?
  4. Transition : validation et nouveau cycle.
- Les phases se chevauchent et intéragissent.
- Chaque phase donne lieu à des documents de travail et de référence.

UML : un langage graphique

- Notation semi-formelle standardisée de modélisation développée par l'OMG.
- Utilisation généralisée pour rédiger les documents de travail.
- Accent mis sur la description, pas sur la justification.
- On segmente la représentation du système en vues.
- Di□érents types de vue :
  - cas d'utilisation
  - logique
  - implémentation
  - processus
  - déploiement

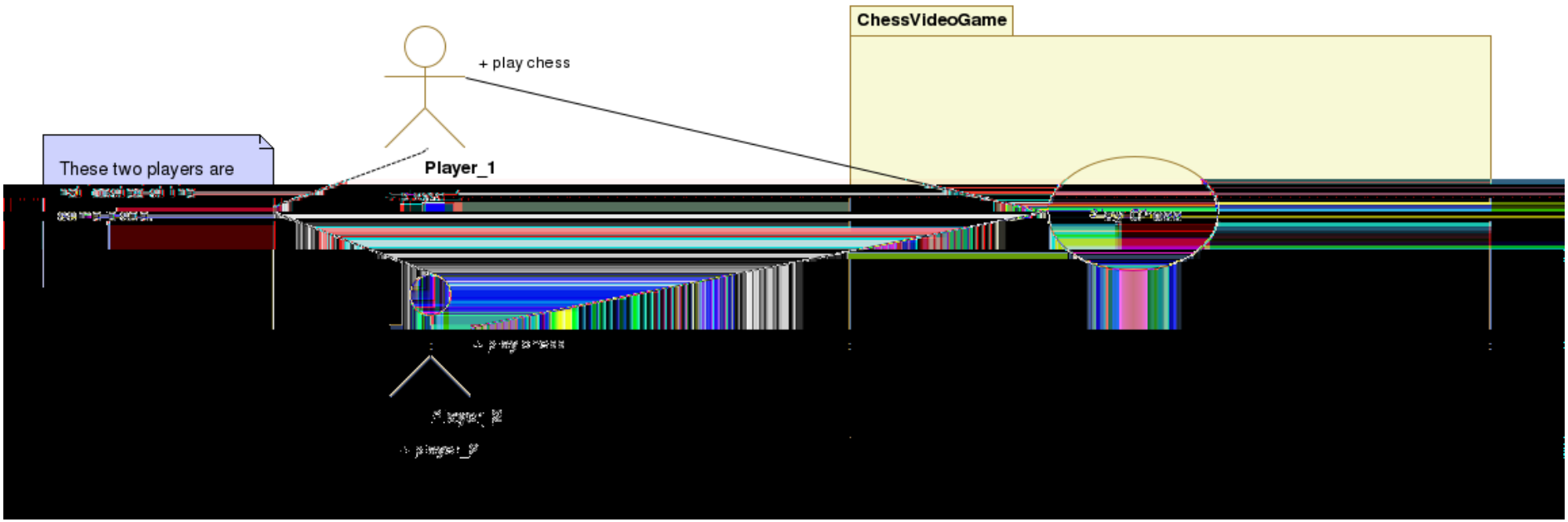
Phase de création

Créer oui, mais QUOI *?*

Création : une idée, une commande . . .

« Nous aimerions introduire le marché des jeux vidéos. Pour commencer, nous voulons vendre un jeu d'échec, jouable sur le réseau Internet. »

Création : préciser l'idée initiale



- `player_1`, `player_2` sont les acteurs du cas d'utilisation
- `play_chess_game` est le nom du cas d'utilisation.
- Des commentaires peuvent être rajoutés.

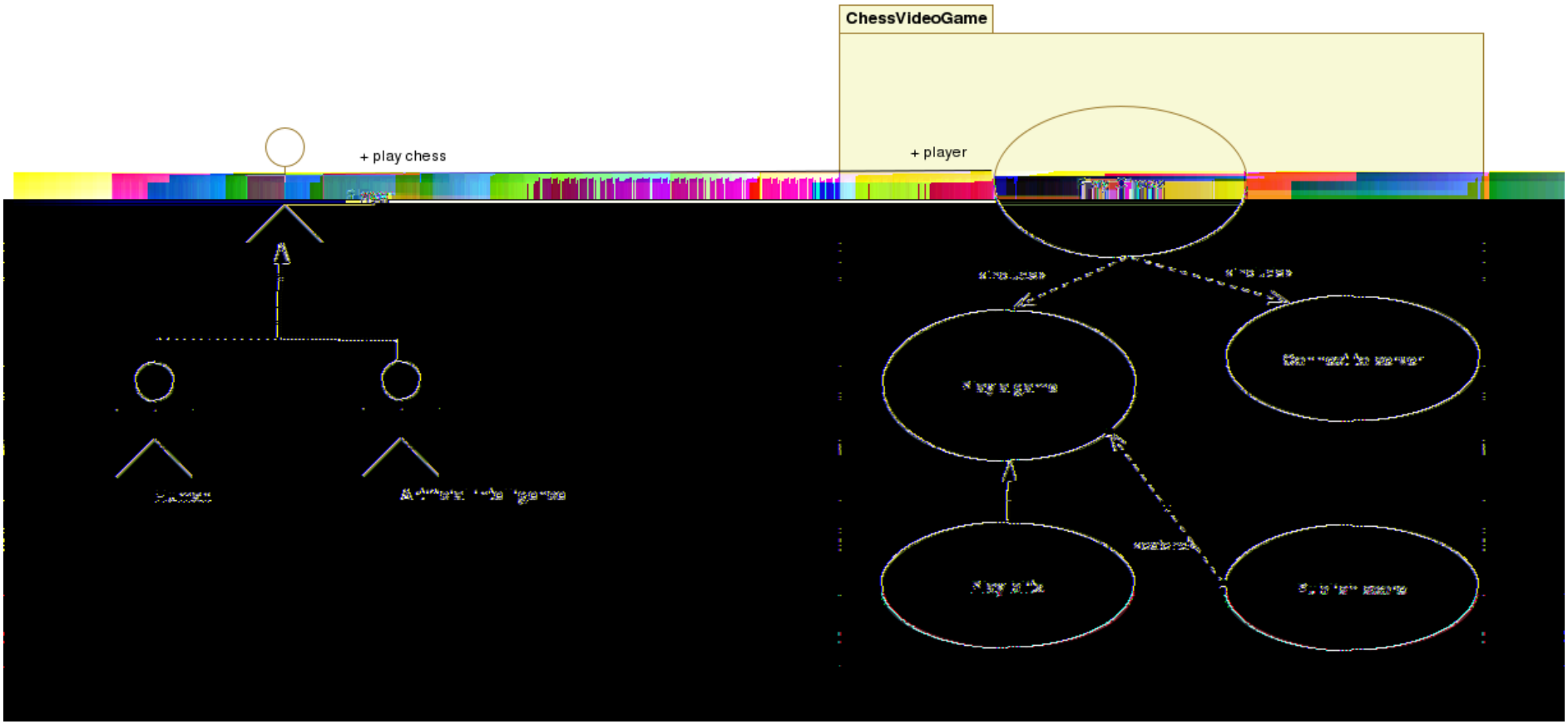
Création : le système doit inclure les cas d'utilisations

- L'utilisation de diagrammes de cas d'utilisation donne un support non technique de communication.
- À l'aide de ces diagrammes, on peut exhiber le contour du système.
- Les acteurs sont des entités externes (humaines ou informatiques) qui utilisent le système.
- Les cas d'utilisation décrivent les interactions possibles entre les acteurs et le système.

Élaboration : didactique de l'extraction des besoins

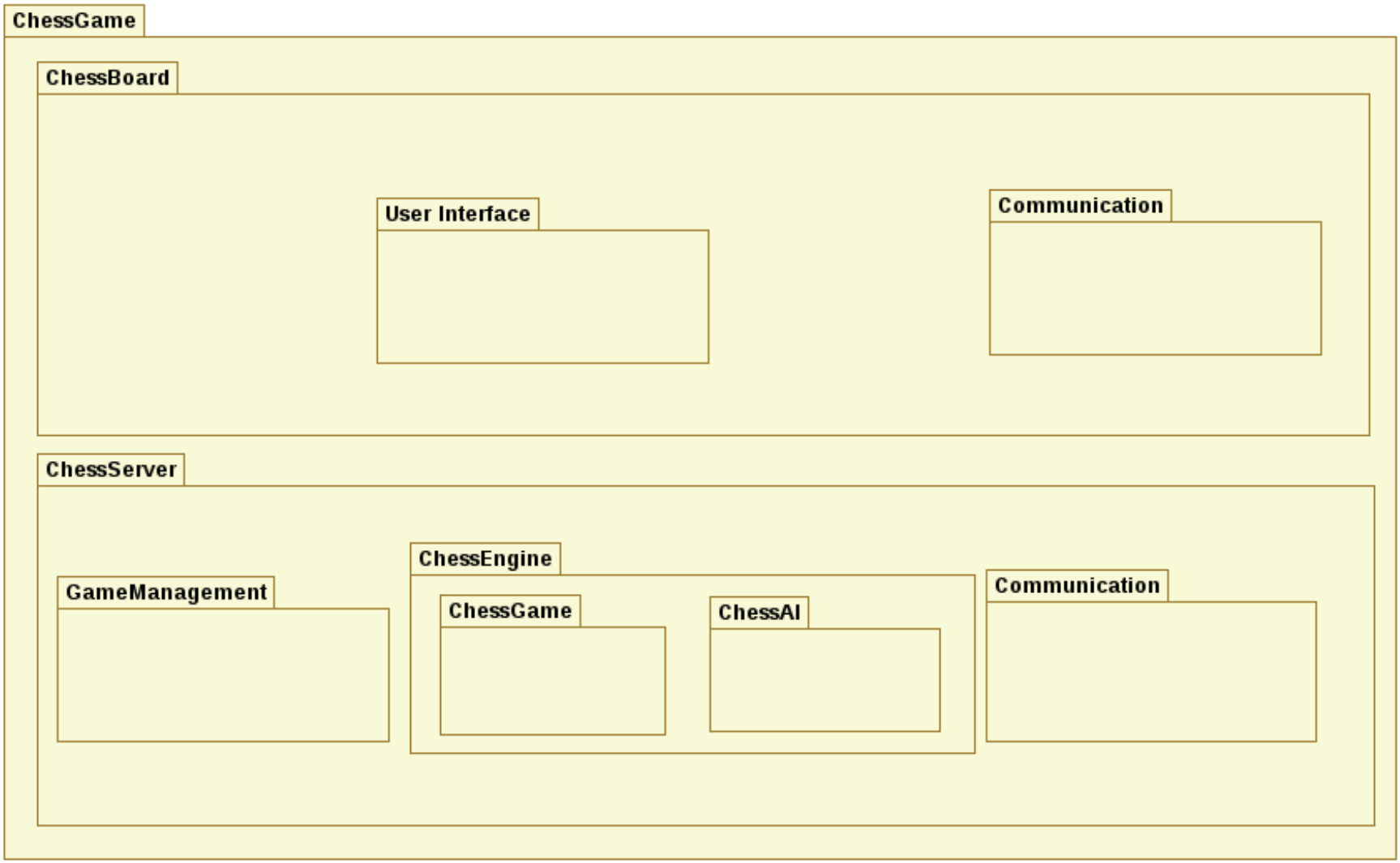
- La phase d'élaboration produit un modèle conceptuel (terminologie UML).
- Ne soyons pas dupes ! Sa simplicité tente de limiter les ambiguïtés.
- Seule une spécification formelle ne contient pas d'ambiguïté.
- Déterminer les besoins implique donc une tension puisqu'il s'agit de trouver le jeu de questions permettant de construire une spécification formelle d'un logiciel à partir des réponses informelles d'un non-expert.
- À tout instant de la conception, on se réfère à l'ensemble des cas d'utilisation pour vérifier la pertinence d'une décision.

Élaboration : structurer l'ensemble des cas d'utilisation



- Relation de généralisation (ligne pleine, flèche triangulaire).
- Relation d'association.
- Relation << `include` >> : sous-cas d'utilisation non indépendant.
- Relation << `extend` >> : cas étendu par une fonctionnalité optionnelle.

Élaboration : fixer une architecture

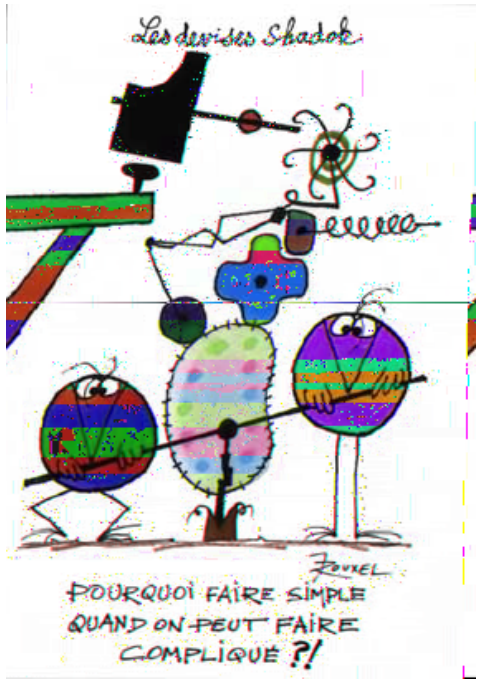


Où placer chaque cas d'utilisation dans cette architecture ?

Définition des domaines

- À ce stade de modélisation, on est capable de nommer les « expertises » nécessaires au développement.
- Ici, il faudra faire appel à des compétences :
  - de conception d'interface utilisateur ;
  - de communication réseau ;
  - d'algorithme d'intelligence artificielle pour les jeux à deux joueurs ;
  - de la connaissance du jeu d'échec.
- La règle d'or est encore une fois la réutilisabilité !
  - Est-ce qu'il existe des bibliothèques existantes pour ces domaines ?
  - Quels sont les savoirs-faire et concepts généraux qui s'appliquent ?

Phase de construction



Je commence à comprendre ce que je dois faire, mais COMMENT *le faire ?*

Construction

- La phase de construction précise les cas d'utilisation en vue d'a□ner l'architecture du système.
- Pour cela, on construit un modèle logique par ra□nement des cas d'utilisation en vues dynamiques du fonctionnement du système et en exhibant son organisation par des vues statiques.

Phase de construction

- Les vues dynamiques sont décris par des diagrammes :
  - de séquences
  - de collaboration
  - d'états-transitions
  - d'activités
- En les précisant, on identifie les objets et les mécanismes généraux du système.
- On organise ces objets en vue statique sous forme de diagrammes de classes ou d'objets.
- On peut alors procéder par itérations : analyse - design - code - test.

Vues dynamiques

Diagramme de séquences

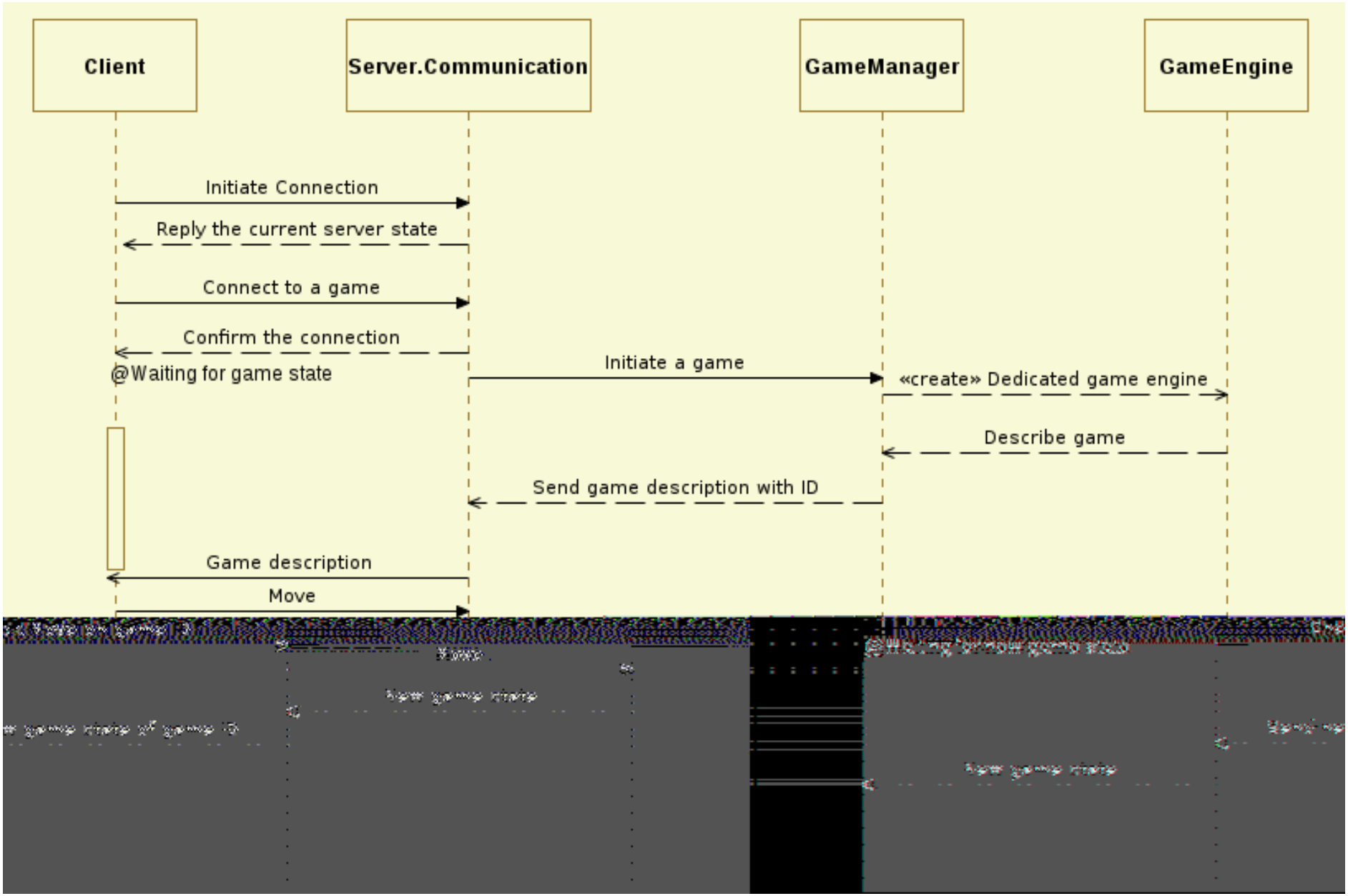


Diagramme de séquences

- Un diagramme de séquence est une vue dynamique.
- Chaque colonne dénote la ligne de temps d'un objet mis en jeu.
- Sur les flèches sont indiqués les messages envoyés.
- En pointillés sont indiqués les réponses aux messages.
- Il existe des notations pour décrire précisément :
  - la nature des messages (asynchrones ou synchrones) ;
  - l'existence de plusieurs *threads* ;
  - l'état de l'objet avant et après la réception ou l'envoi d'un message.

Diagramme de collaboration

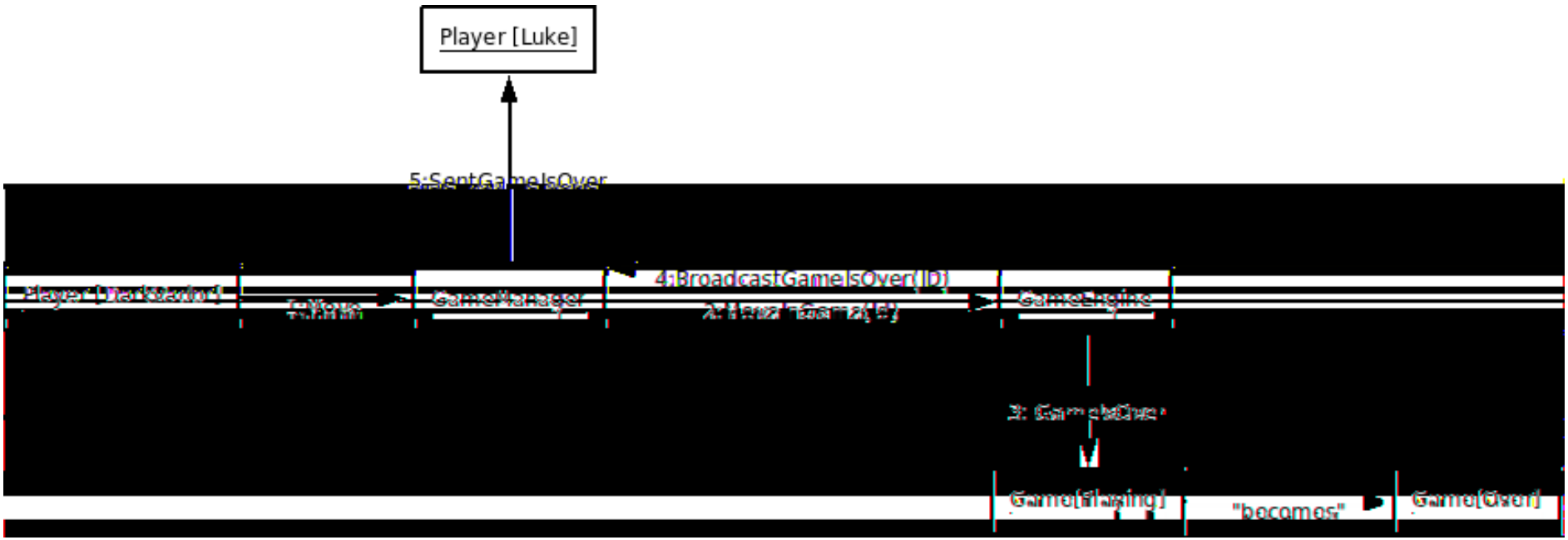


Diagramme de collaboration

- Un diagramme de collaboration est une vue dynamique.
- Un diagramme de collaboration met l'accent sur les relations entre objets.
- Un diagramme de séquence se focalise sur l'enchaînement des messages.

Diagramme d'états-transitions

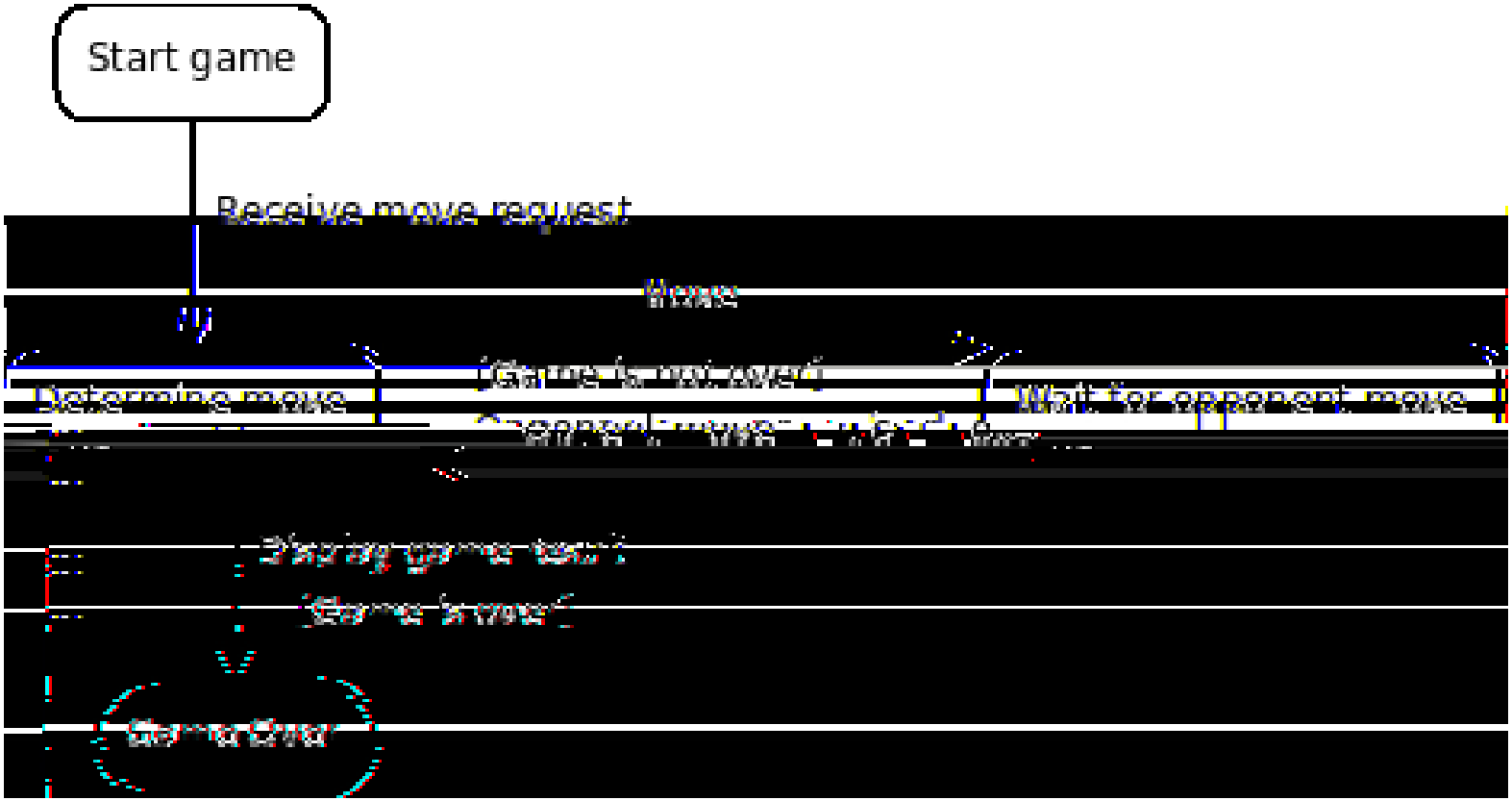


Diagramme d'états-transitions

- Un diagramme d'états-transitions est une vue dynamique.
- Il dénote les di□érents états du système au niveau :
  - global ;
  - d'un cas d'utilisation ;
  - d'une opération ;
  - d'un objet particulier.
- Une transition est formée par :
  - un déclencheur (un signal, un envoi synchrone de message, une condition logique, une condition temporelle) ;
  - une garde (une condition logique de validité de la transition) ;
  - un action (effectuée lors de l'activation de la transition).

Diagramme d’activités

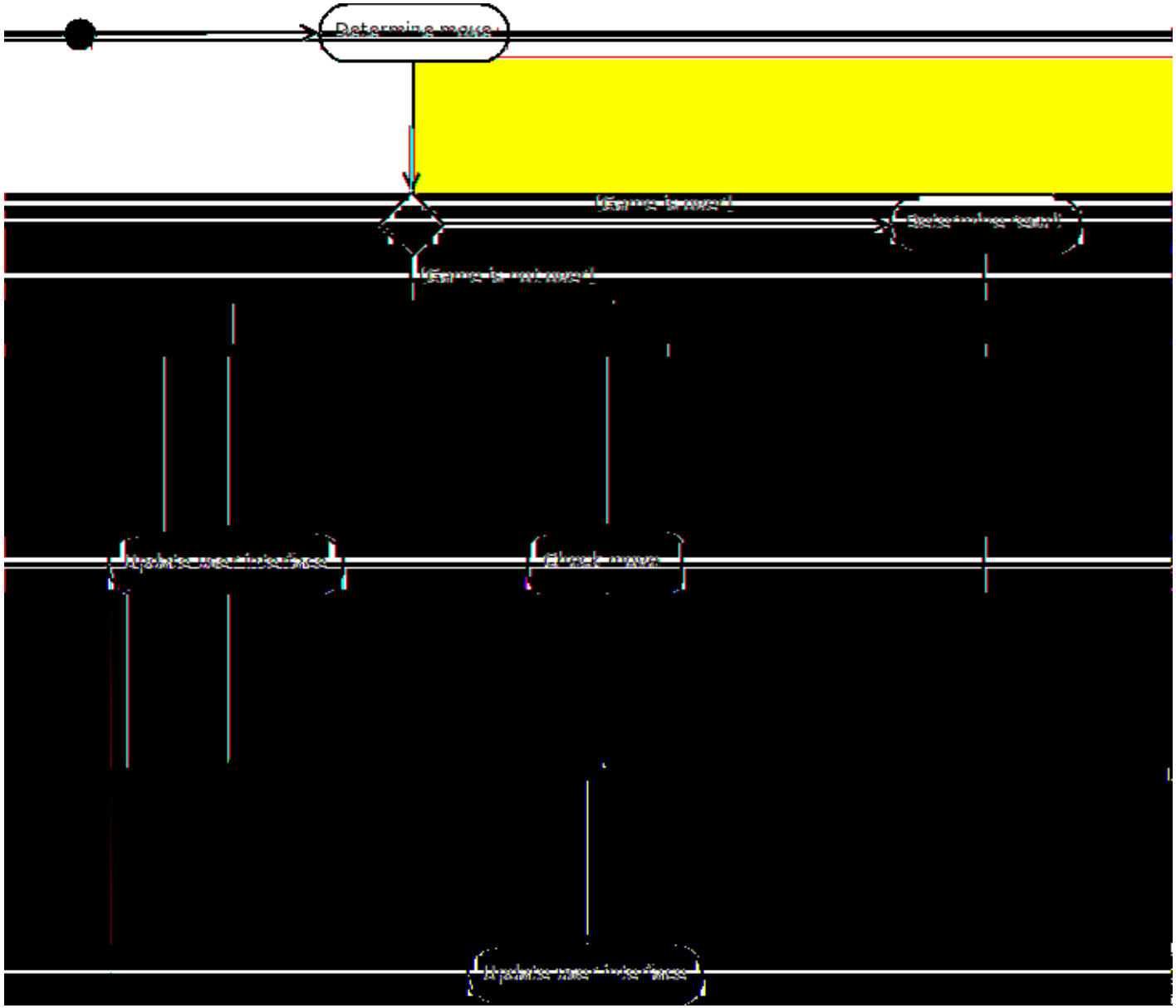


Diagramme d'activités

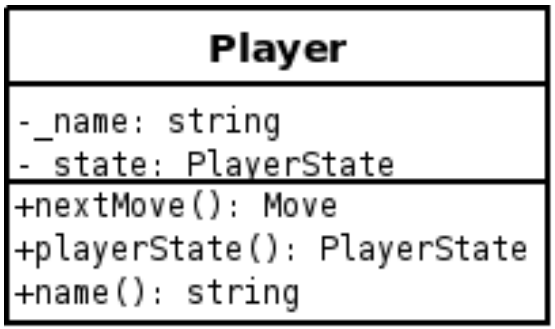
- Un diagramme d'activité est une vue dynamique.
- Un tel diagramme décrit l'enchaînement des opérations concurrentes e□ectuées par le système.
- On explicite les introductions et les points de rendez-vous des opérations concurrentes.

Vues statiques

Diagramme de classes

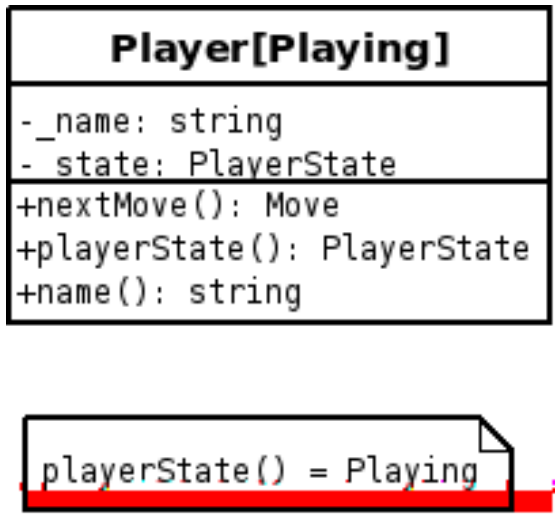
- Un diagramme de classes est une vue statique décrivant l'organisation des composants.
- Un composant peut être :
  - une classe ;
  - une classe dans un état donné ;
  - un objet ;
  - un nœud ou une ressource logicielle ;
  - un rôle ;
  - une interface ;
  - un acteur ;
  - un cas d'utilisation ;
  - un sous-système.
- Deux composants peuvent être en relation :
  - de généralisation ;
  - d'association ;
  - de dépendances ;
  - de réalisation.
- Une notation existe pour chacun de ces types de composant et de relation.

Composant : classe



- Premier bloc : nom de la classe (en italique si abstrait), éventuellement préfixé par un stéréotype (« interface », « role », ...).
- Second bloc : liste des attributs, chaque ligne contient :
  - $+$ signifie public, - signifie privé, $\#$ signifie protégé ;
  - le nom de l'attribut ;
  - le type de l'attribut.
- Troisième bloc : liste des opérations, chaque ligne contient :
  - $+$ signifie public, - signifie privé, $\#$ signifie protégé ;
  - le nom de l'opération ;
  - ses arguments ;
  - le type de la réponse.

Composant : classe dans un état donné



Composant : objet

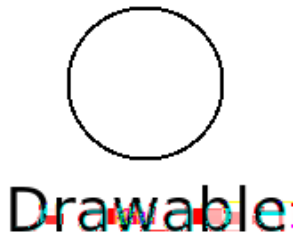
- Représente une instance particulière d'une classe.
- Il est possible de spécifier la valeur des attributs de l'instance.

Composant : nœud ou ressource logicielle

Composant : rôle

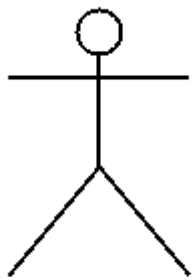
- Un rôle décrit l'interface d'un objet prenant part à une interaction.
- Nous reviendrons sur cette notion un peu plus tard.

Composant : interface



- Une interface représente un ensemble d'opérations caractérisant un comportement.

Composant : acteur



- Entité externe au système et sollicitant son utilisation.

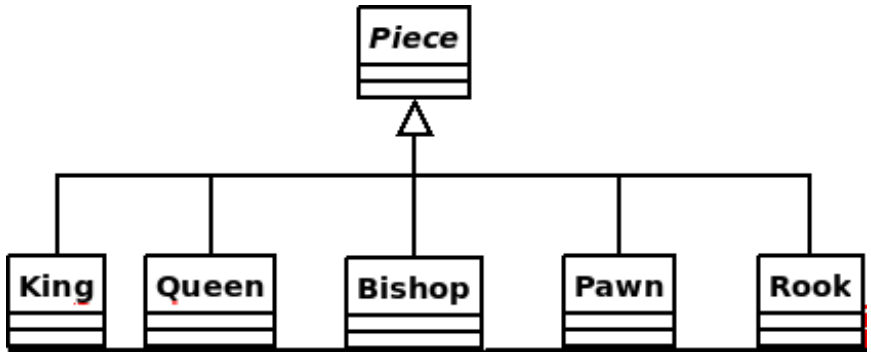
Composant : cas d'utilisation

- Une spécification du comportement du système lors de l'interaction avec une entité externe.

Composant : sous-système

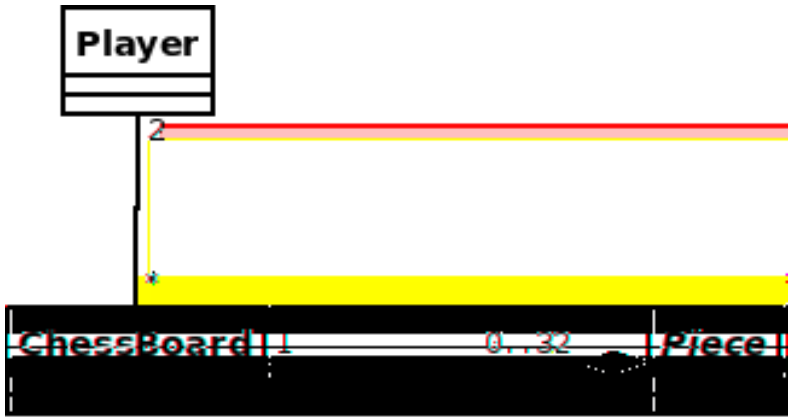
- Une unité logicielle caractérisée par une identité, une spécification et une implémentation.

Relation : généralisation



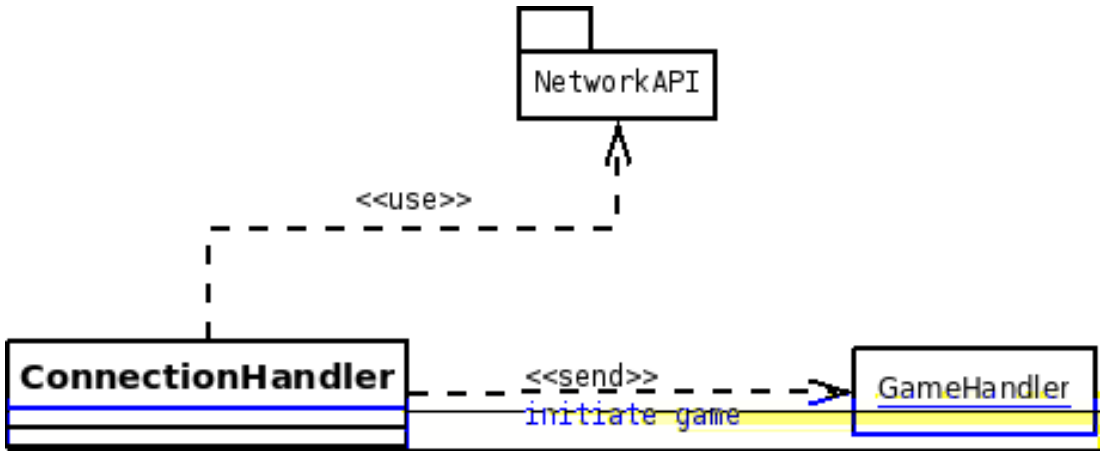
- Une relation de généralisation est notée par une ligne pleine et terminée par une flèche.
- Elle dénote qu'une classe est plus générale qu'une autre.
- Elle peut signifier qu'une classe hérite d'une autre, mais pas nécessairement.
- Nous allons voir cette distinction un peu plus loin.

Relation : association



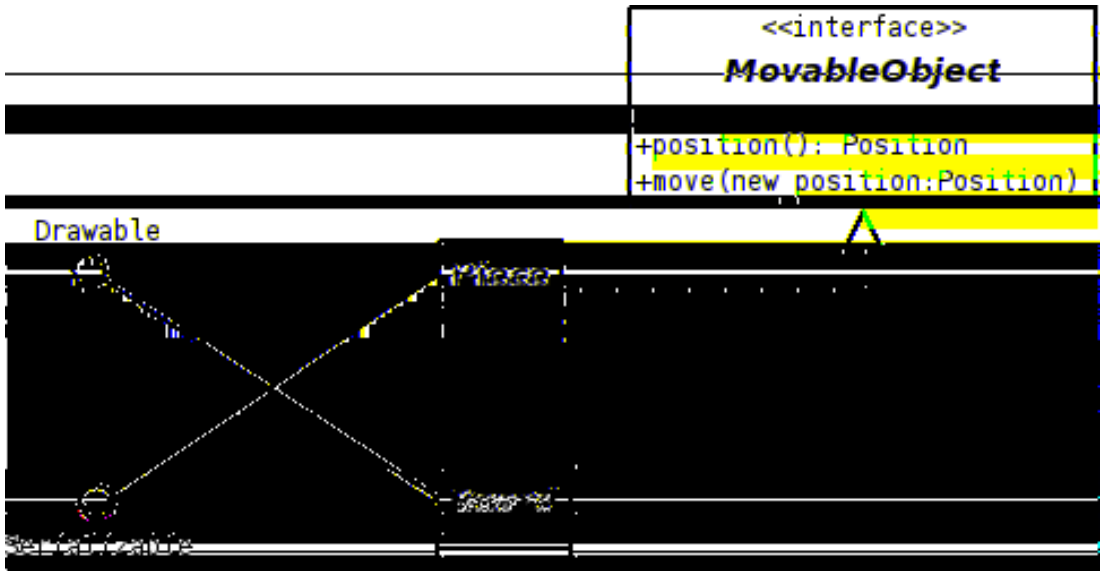
- Une relation d'association est notée par une ligne pleine dont les extrémités sont annotées (optionnellement) par :
  - une multiplicité :
    - * signifie « un nombre indeterminé »
    - $n \in \mathbb{N}$, un nombre n fixé.
    - $m \quad n$ $(m\ n) \in \mathbb{N}^2$, un nombre entre m et n.
  - un losange plein signifie que l'association est une composition.
  - un losange vide signifie que l'association est une aggrégation.
- La composition donne le droit de vie ou de mort à un objet sur un autre.

Relation : dépendance



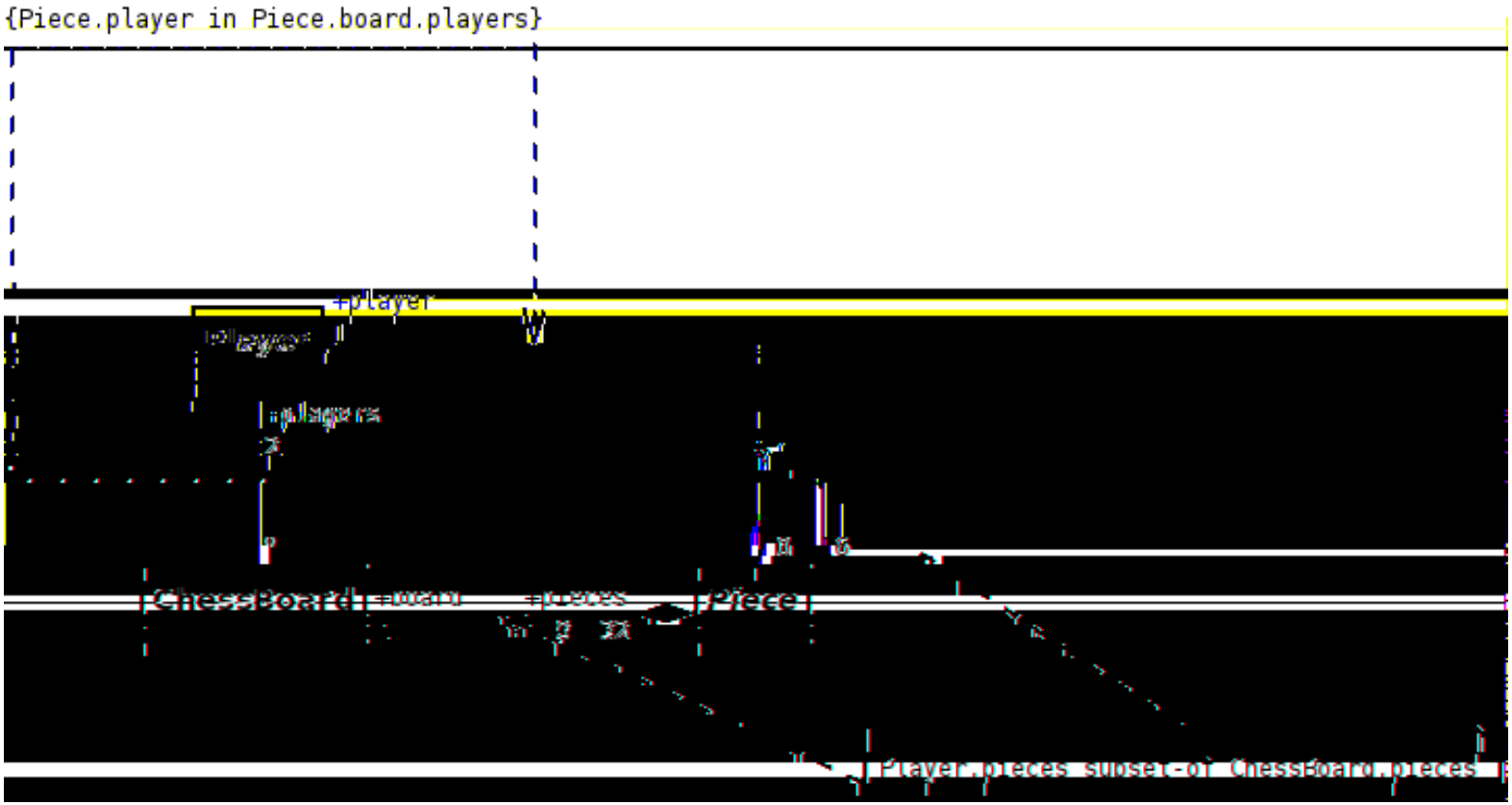
- Une relation de dépendance est notée par un ligne pointillé terminée par une flèche.
- Elle dénote l'existence nécessaire de certains composants pour le bon fonctionnement d'un composant particulier.
- Il y a di□érents types de dépendances : `call`, `bind`, `access`, `derive`, `friend`, `import`, `instantiate`, `parameter`, `realize`, `refine`, `send`, `trace`, `use`.

Relation : réalisation



- Il y a deux façon de noter une relation de réalisation :
  - une ligne pleine connectant une classe et une interface nommée ;
  - une ligne pointillée terminée par une flèche d'une classe à une classe-interface.
- Attention, une réalisation logique ne s'implémente pas forcément par la réalisation d'interface du langage de programmation (**implements** ou héritage multiple . . .). Nous allons revenir sur ce point.

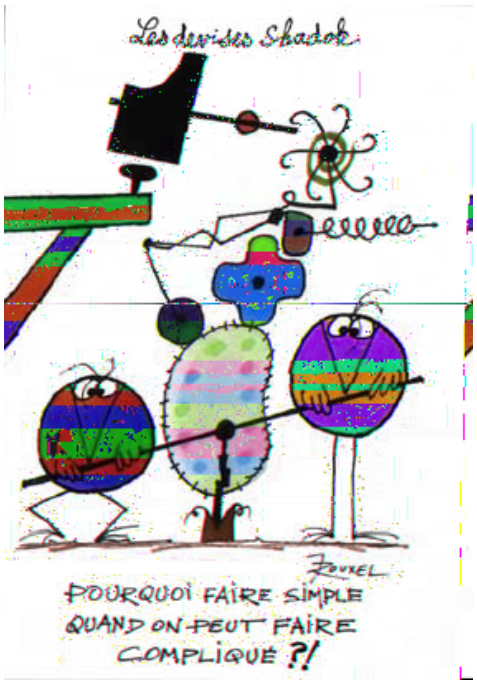
Contraintes sur les relations



Former des mots n'est pas écrire ...

- Les vues proposées par UML permettent de former des modèles du système.
- Un modèle est une simplification d'un système complexe.
- Il sert à l'analyse, c'est-à-dire l'étude du problème en termes logiques.
- Il permet la conception, c'est-à-dire la mise au point de la solution.
- Un modèle *trop concret* ne permet pas de se focaliser sur les mécanismes généraux mis en jeu (et d'obtenir une solution simple).
- Un modèle *trop abstrait* peut trop s'éloigner des contraintes techniques réelles (et peut rendre la solution irréalisable).
- Savoir dessiner des modèles UML ne su□t donc pas : une connaissance transversale et en profondeur des concepts et des technologies de l'informatique est indispensable.

Conseil

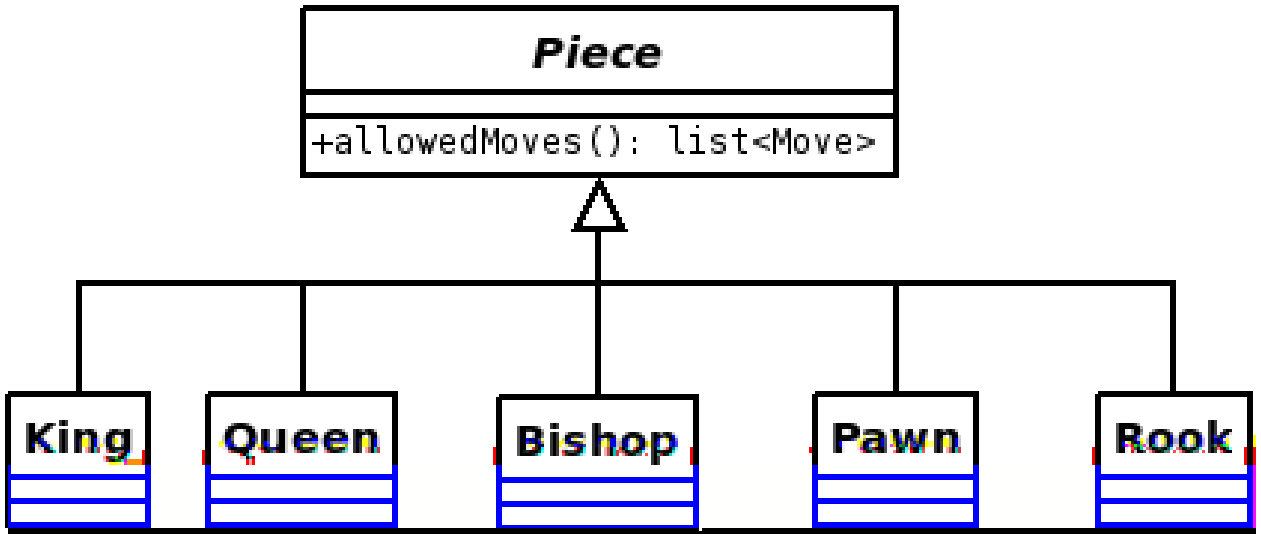


- Rester simple et éviter les réponses automatiques (déterminisme) sont de bons réflexes à avoir en tête.

Modèles logiques et d'implémentation

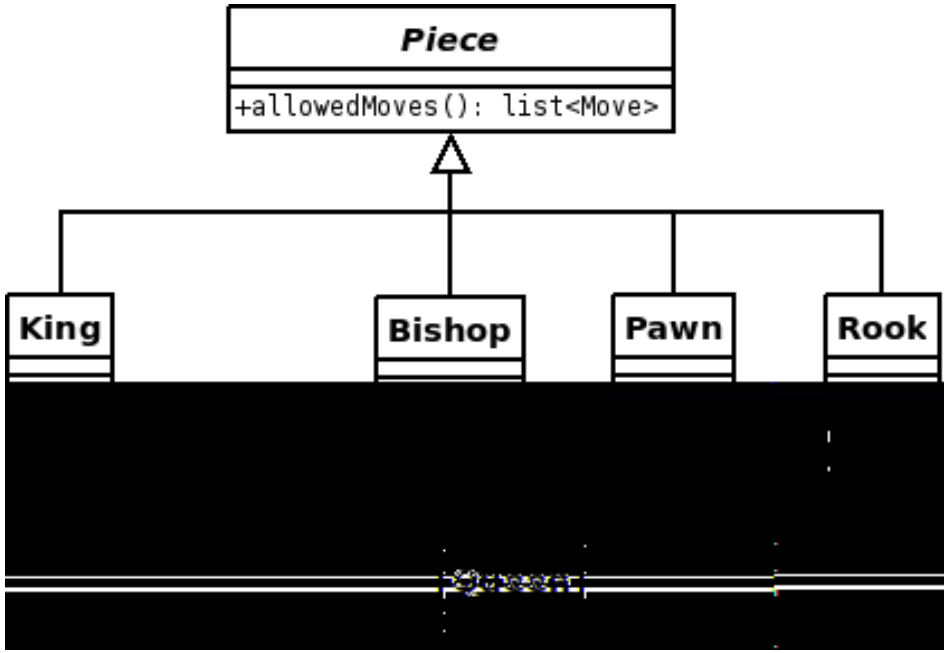
- Il est souvent utile de partitionner l'ensemble des modèles en deux groupes :
  - les modèles logiques, abstraits et simples, qui rendent compte d'une version idéale du système.
  - les modèles d'implémentations qui décrivent la façon dont le code est structuré (pour des raisons techniques).

La généralisation dans un modèle logique



- La relation logique de généralisation entre une classe A et une classe B peut s'exprimer : « B est un A ».
- Exemple : « un roi est une pièce. »
- C'est le principe de subsomption.

La généralisation dans un modèle d'implémentation

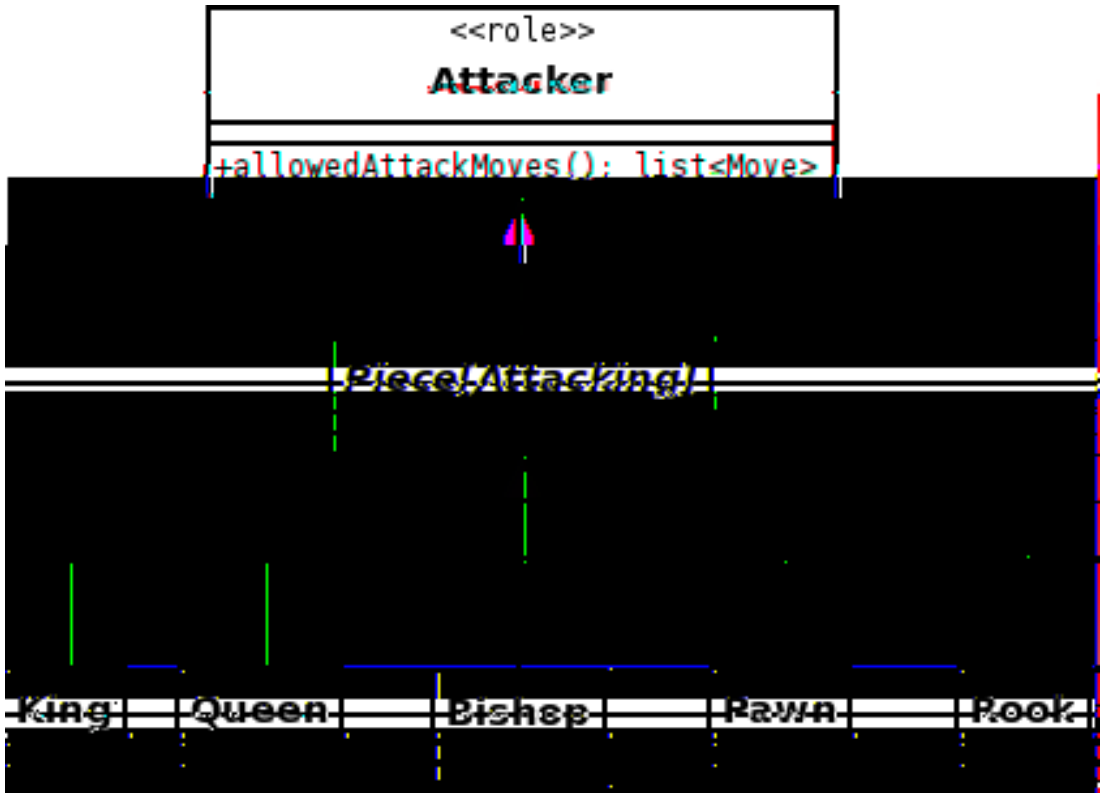


- La relation de généralisation entre une classe A et une classe B peut aussi rendre compte d'un héritage.
- Dans notre exemple, l'héritage est utile pour factoriser le code.
- Cependant, il faut alors utiliser la relation de subsomption définie par le modèle logique, pas par la relation d'héritage !

Di□érences entre généralisation et rôle

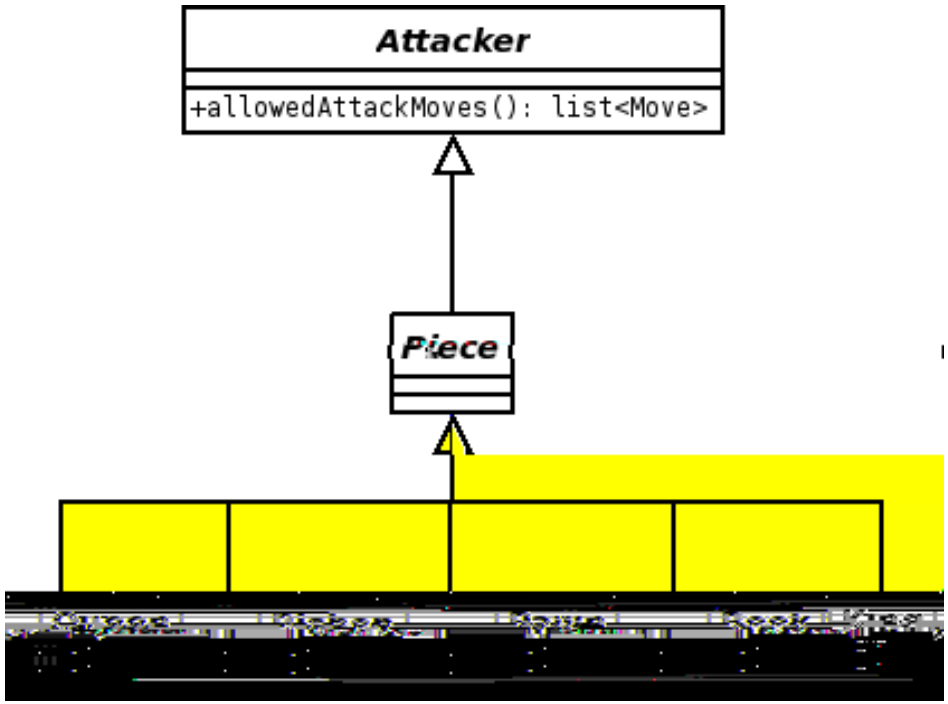
- Un objet peut parfois prendre un masque lorsqu'il est pris dans une interaction donnée.
- Exemple : lorsqu'une pièce tente la prise d'une autre pièce, elle a le rôle d'un attaquant. Le gestionnaire de règle doit vérifier que le coup est valide, en fonction du type de la pièce. Il attend donc d'un attaquant qu'il décrive ses règles spécifiques.
- Pourtant, on ne peut pas dire qu'une pièce est toujours un attaquant.
- Les rôles servent de classifications et déclassifications dynamiques.
- Au contraire, les généralisations constituent des classifications statiques.

Rôle dans une vue logique



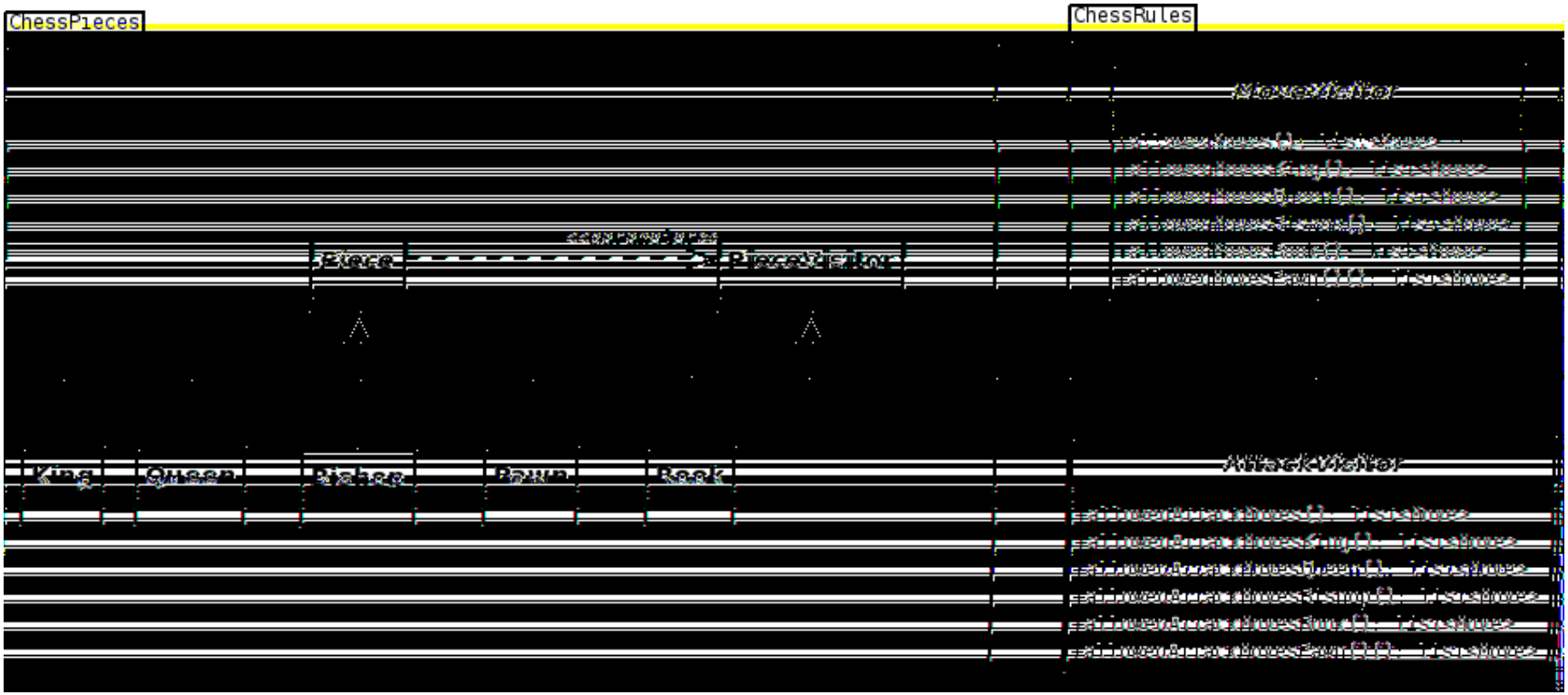
- C'est une solution simple et logique mais comment implémenter cela ?

Rôle dans une vue d'implémentation : première solution



- Toutes les classes filles doivent implémenter la méthode `allowedAttackMoves`.
- On trouvera aussi certainement une méthode `allowedMoves`.
- Peut-être peut-on faire mieux pour avoir la garantie qu'on ne confondera jamais ces deux méthodes ?

Rôle dans une vue d'implémentation : seconde solution



- On a gagné :
  - Une simplification de l'interface des pièces.
  - La définition des règles est locale à un module.
  - La hiérarchie des pièces est extensible fonctionnellement.
- On a réduit l'encapsulation de nos pièces : l'état `Attacking` est fragmenté.

Fixer une frontière entre encapsulation et externalisation

- Choisir entre les deux solutions précédentes est non trivial.
- Il faut se donner des critères d'évaluation des solutions en termes :
  - d'extensibilité ;
  - de simplicité ;
  - de modularité ;
  - d'unité conceptuelle.

Limiter les interférences

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only ; we also know that it should be e cient and we can study its e ciency on another day, so to speak. In another mood we may ask ourselves whether, and if so : why, the program is desirable. But nothing is gained –on the contrary !– by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for e ective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect" : it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.

Edsger W. Dijkstra
"On the role of scientific thought"
1974

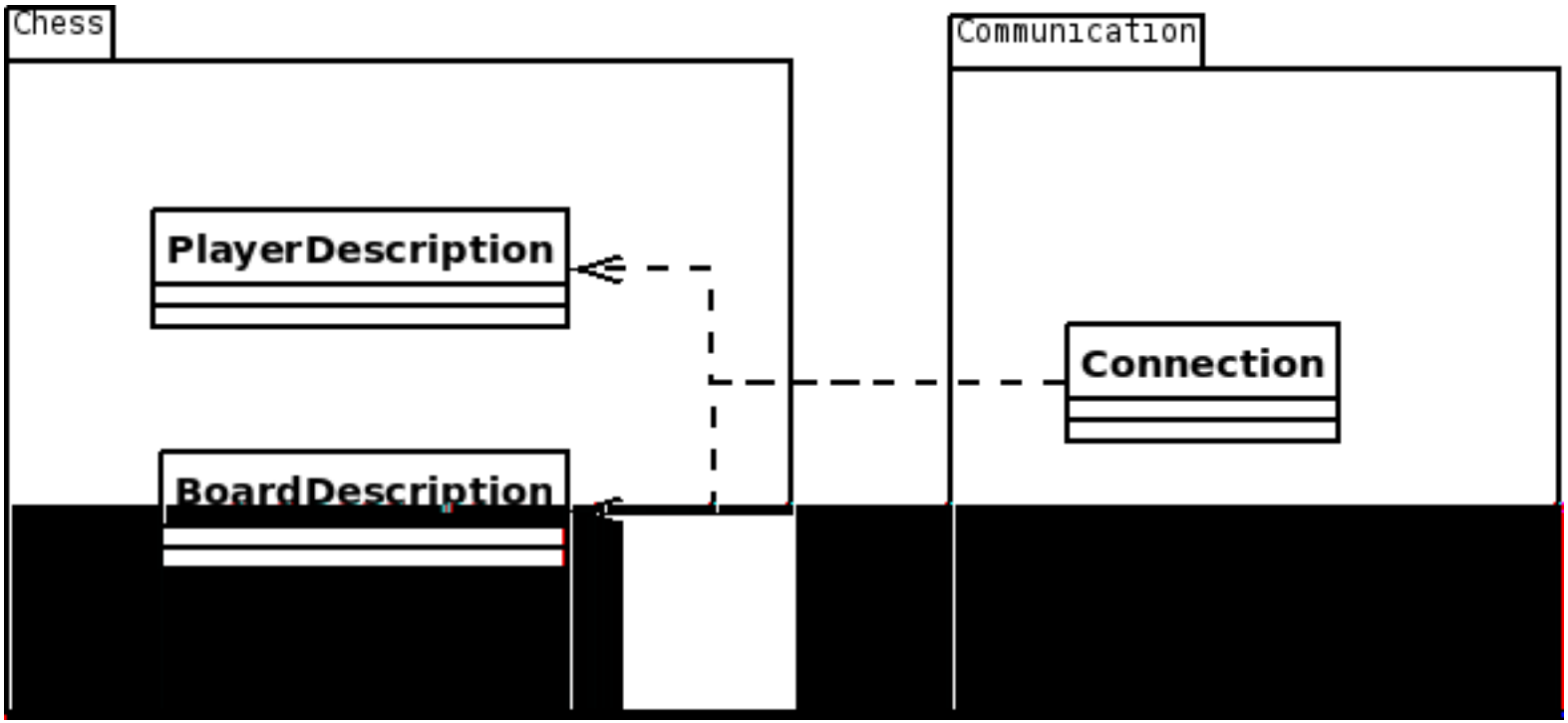
Limiter les interférences

- D'une façon générale, il faut chercher à simplifier la résolution des problèmes en limitant leur interférence. C'est ce qu'on appelle la séparation des domaines (*separation of concerns*).
- Exemple : la gestion d'une base de données doit être indépendante du procédé utilisé pour a□cher ses données.

Encapsuler pour maintenir les invariants

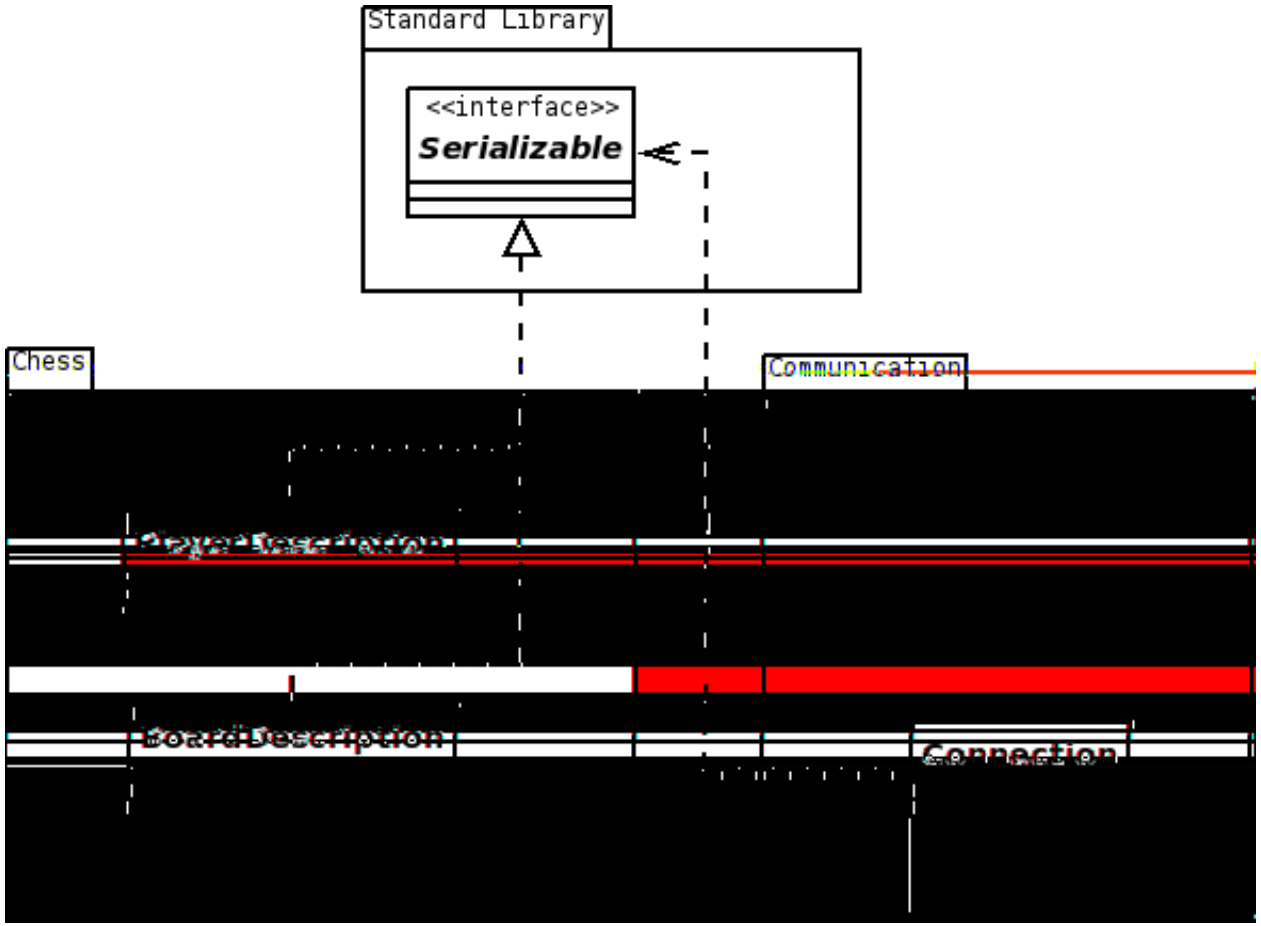
- Si une séparation nécessite une exposition trop importante de l'état d'un objet (qui pourrait mettre en jeu la maintenance des invariants de cet objet), alors elle est certainement douteuse.
- Exemple : un objet implémentant un ensemble à l'aide d'un arbre rouge-noir ne doit pas exposer la structure de cet arbre.

À quoi servent les interfaces ?



- La classe `connection` a besoin de connaître les classes de description pour pouvoir les utiliser lors des entrées/sorties.
- Ces deux modules sont donc dépendants.
- La classe `connection` *fait référence directement* à ces deux classes.

À quoi servent les interfaces ?



- La classe `connection` s'appuie sur une définition générale.
- Les deux modules sont maintenant dépendants d'une convention commune.
- La classe `connection` *fait référence à l'existence* de classes sérialisables.

Comment implémente-t-on une interface ?

- L'implémentation d'une interface peut prendre plusieurs formes :
  - en fonction du langage de programmation utilisé ;
  - en fonction du niveau d'intrusion voulue (implémentation externe ou interne).

Implémenter une interface dans di□érents langages

- En JAVA :

```
interface Serializable {
   private void writeObject (...) ;
   // ...
}
public abstract class Piece implements Serializable {
}
```

- En C++ :

```
class Serializable {
   protected : void writeObject (...) = 0 ;
   // ...
} ;
class Piece : public Serializable {
}
```

- En O'CAML :

```
class type serializable = object
   method writeObject : stream → unit
end
class virtual piece = object
   method virtual writeObject : stream → unit
end
```

En O'CAML, le sous-typage structurel permet d'implémenter une interface par le seul fait de posséder les méthodes attendues.

Implémenter une interface sans le savoir *a priori*

- Imaginons donner une hiérarchie de pièces du jeu d'échec n'implémentant pas les méthodes nécessaires à l'interface `Serializable` mais une fonction de conversion entre les pièces et les chaînes de caractères.
- On peut utiliser le patron de conception d'adaptateur pour construire une classe implémentant l'interface `Serializable` à partir des méthodes de conversion. Nous verrons cela lors du prochain cours !
- Cet adaptateur serait alors dépendant de la classe `Piece`.
- La programmation générique permet d'écrire un adaptateur traduisant toute classe sachant se convertir en chaîne de caractère en une classe implémentant l'interface `Serializable`. Nous verrons cela lors du cours sur la programmation générique !

Transition

« Je suis très content de ce jeu d'échec ! Il devrait être facile pour vous de me développer un jeu de dame, maintenant ! Non ? »