

# Conception & Programmation

Objet *Avancées*

Mireille Blay-Fornarino  
septembre 2018

<https://mbf-iut.i3s.unice.fr/>

---

# Objectifs

- ➔ O1 : Produire une **conception détaillée** en appliquant des  **patrons de conception**, la **mettre en oeuvre** en utilisant **des bonnes pratiques de programmation** orientée objet.

**A poor team will always create a poor system.**

**Martin Fowler, 2014, Microservices**

**<https://martinfowler.com/articles/microservices.html>**

# Evaluation du module

- 1 note d'examen (coefficient 1)
- Rendus des TDs et assiduité (coefficient 0,75)
- Rendu de fin de module : Carnet de bord (coefficient 0,75)

Toutes les informations sont sur le site web  
qui est la référence !

# Evaluation du module

## Bonus

- Des QCMs
- Des cours préparés par les étudiants
  - Vous choisissez les thèmes et on en discute, si le thème est accepté,
  - Vous préparez le cours ....
    - précisez sa durée (forcément inférieure à 1h)
  - Vous faites la présentation.

# Outils indispensables

- Un gestionnaire de version : GIT
  - Si vous voulez aussi : SourceTree : outil au dessus de GIT (<https://www.sourcetreeapp.com/>)
- Un environnement de développement :
  - Eclipse : <http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/marsr>
  - ou IntelliJ : <https://www.jetbrains.com/idea/>
  - NON : ~~JCREATOR~~ ou ~~notePad~~ ou autre
- ☞ Un outil de modélisation : <https://www.modelio.org/>

## Tableau de synthèse des évaluations

Thème d'évaluation	Très satisfait	Assez satisfait	Peu satisfait	Très peu satisfait	Sans objet	NbR	(*)TxS
Compréhension des objectifs du module	21	23	10	4	0	58	68%
Qualité des supports pédagogiques	24	21	10	3	0	58	71%
Cohérence entre apports théoriques et pratiques (Cours, TD, éventuellement : TP, Labo, Projet...)	27	16	12	3	0	58	72%
Adéquation entre les contrôles et le contenu des enseignements	26	23	5	4	0	58	74%
Concernant votre implication personnelle dans cet enseignement, vous êtes	21	23	10	4	0	58	68%
<b>Taux moyen de satisfaction</b>	<b>41%</b>	<b>37%</b>	<b>16%</b>	<b>6%</b>	<b>0</b>	<b>290</b>	<b>71%</b>

(\*) NbR = Nombre de réponses.

TxS = Taux de satisfaction = ( Très satisfait \* 100% + Assez satisfait \* 66.67% + Peu satisfait \* 33.33% + Très peu satisfait \* 0% ) / Nombre de réponses.

Les résultats sont arrondis, à 1% prêt

Les commentaires de l'an dernier pour expliquer, prévoir, prévenir.

Les diapos sont très utiles lors des CM, mais lorsque nous les relisons sans les explications de l'enseignant il est plus difficile de comprendre certaines notions (beaucoup de pages)

Prenez des notes !

Le niveau en Java demandé pour comprendre les notions abordées est trop élevé. Beaucoup trop d'étudiant sont perdus en TD, ce n'est pas normal, il faut faire quelque chose ! De plus je ne comprends pas pourquoi nous passons 75% du temps dans ce module à coder en java et si peu de temps à travailler en

Voir objectifs du module

Peu d'implication car je ne comprend rien à ce cour. Malgré cela l'interro n'est pas très compliquée.

Tout d'abord je dois avouer que j'étais plus que tenté de faire une fausse évaluation pour figurer sur votre diapo de premier CM de l'année prochaine :-). Plus sérieusement, le module qui est passionnant et on apprend énormément d'éléments ! Qui plus est, le module est encadré par une titulaire qui ne compte pas ses heures et dont les supports & TDs proposés sont toujours de très bonne qualité. On ajoutera également que les TDs sont toujours en adéquation avec les notions abordés en cours. Enfin, Mme Lecat (mon encadrante de TD) qui est toujours aussi géniale et disponible : si l'on se montre intéressé, on peut quasiment tout demander, y compris par mail et à (presque) toute heure ! :-). (mention spéciale aussi pour la titulaire pour qui c'est pareil) En bref : un module génial avec des profs géniaux - si tous les modules pouvaient se passer comme ça, je signe tout de

Ok.. Parlez et je vais tenter d'améliorer

Les TD sont parfois pas claires. Et quelques patterns designs sont assez difficiles à assimiler et à mettre en œuvre.

Pas le meilleur cours de la formation, il y a des bons éléments mais les tds ne sont pas de la meilleure qualité. Le carnet de bord final a demandé des heures de travail supplémentaires en dehors des cours ce qui est inacceptable.

Non. Etudiant? et vous le savez dès aujourd'hui !.

Un module avec des objectifs assez incompréhensible, ou les tests sont omniprésents et souvent plus important que la conception.

Oui, aujourd'hui les deux vont de paires, et cela doit vous aider.... mais OK

J'ai pas du tout aimé le principe d'utiliser des codes inconnus pour réaliser un TD. On attend de nous de comprendre des classes complexes écrites par quelqu'un maîtrisant mieux le langage que nous, et réaliser le TD, en 1 à 2 séances. C'est très facile de se perdre, et j'ai personnellement peu apprécié la fin du module. Il faudrait soit limiter la quantité de codes étrangers fournis, ou alors les expliciter clairement en début de séance (ça peut prendre du temps), sinon on risque de perdre beaucoup d'élèves.

Non... L'objectif est que VOUS appreniez à apprendre !

Les Design Pattern sont compliqué à intégrer

Une enseignante proche des étudiants, qui comprend et écoute leurs points de vue. Des CM utiles et ludiques, et des

classes complexes écrites par quelqu'un maîtrisant mieux le langage que nous, et réaliser le TD, en 1 à 2 séances. C'est très facile de se perdre, et j'ai personnellement peu apprécié la fin du module. Il faudrait soit limiter la quantité de codes étrangers fournis, ou alors les expliciter clairement en début de séance (ça peut prendre du temps), sinon on risque de perdre beaucoup d'élèves.

Les Design Pattern sont compliqué à intégrer

Une enseignante proche des étudiants, qui comprend et écoute leurs points de vue. Des CM utiles et ludiques, et des TDs qui demandent une certaine implication personnelle et qui permettent de comprendre des principes passionnants de la programmation orientée objet, le tout encadré par une très bonne enseignante.

Module très dynamique, j'ai apprécié les différentes approches sur le développement. J'aurai aimé cependant avoir plus d'exercices en fin de module qui nous entraînent à identifier les patterns après les avoir étudiés dans la globalité.

Rien à redire sur le contenu de cette UE. Tous les TDs sont très prenant. Bravo ! On aimerait quand même avoir quelques lignes de codes dans le cours qui sont typiques et réutilisables. Afin de pouvoir comprendre comment les utiliser.

Pour moi, elles y sont... si vous confirmez, dites moi .

Je n'ai réussi à aimer cette matière pour autant notre professeur est très agréable et prend du temps pour tout expliquer que se soit en cours ou en TD.

Implication du professeur (Mme Blay) irréprochable. Mais les TDs sont un peu difficiles à suivre. En effet, le choix des sujets repose sur des problèmes assez complexes, qui demandent beaucoup d'analyse, et le temps est trop limité.

Oui. DUT... Du soutien ?





Globalement assez satisfait du module, en revanche il faudrait peut être un peu plus aider les étudiants au début de l'année puisque après 2 mois de vacances sans programmer c'est assez difficile de reprendre le java :s

Les TD sont assez longs, la matière devrait donc avoir plus d'heures par semaine afin de pouvoir cerner la totalité des sujets lors des heures de TD


Un module qui nous apprend la programmation "professionnelle" dans le sens "coder proprement". Intervention de prof pour concrétiser les éléments théoriques vu en cours. Enseignante super, investie, qui cherche avant tout à faire progresser les gens en donnant de sa personne. Rien à ajouter, j'ai tout dit en 2 lignes, quand les choses sont claires, justes et surtout bien faites pas besoin de faire un pavé pour le justifier.

Intéressant mais manque de concret, et un peu trop répétitif

Très bon cours, la passion de l'enseignant donne envie de s'investir. De plus l'intervenant en td est à l'écoute et vérifie qu'en els élèves aient compris les notions abordées.

Module claire qui nous fait évoluer sur la programmation orienté objet avec les patterns etc. Le module est plutôt vivant les td sont sympa.

Le module est globalement bien expliqué, cependant il y a un écart entre les TD et l'amphi qui nous fait penser à deux cours différents.



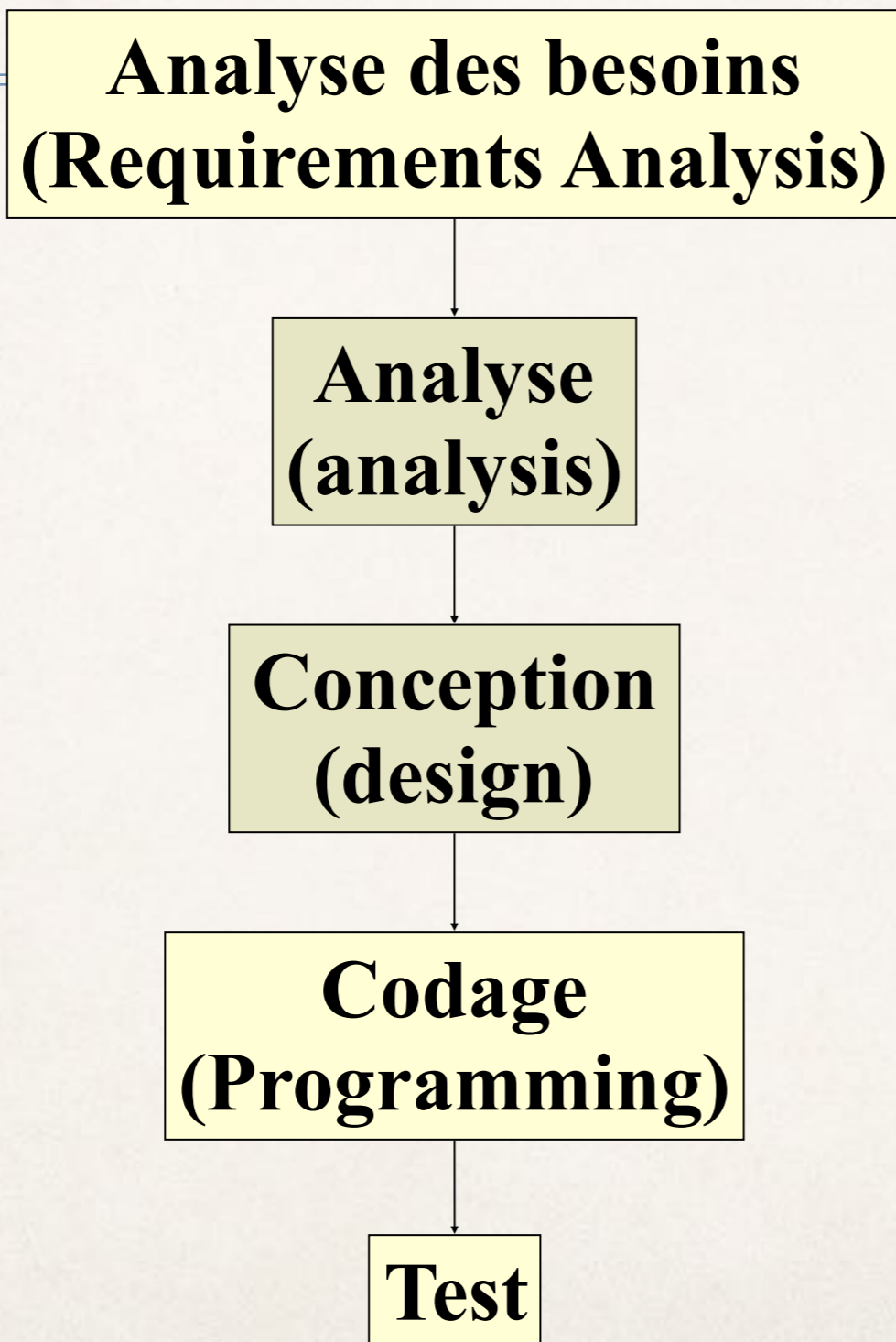
Très bonne prof, beaucoup de pédagogie, le cours est très intéressant et il en est de même pour les TDs, cependant manque de 30mins pour que la durée du DS soit parfaite, on a pas assez le temps de bien faire jusqu'au bout

On verra...

# De l'analyse à la conception orientée objet

- 1- Rappels UML etc. basés sur le cours de Pascal Molli, Université de Nancy, Loria
- 2- Positionnement des cours suivants

# Introduction et vue d'ensemble

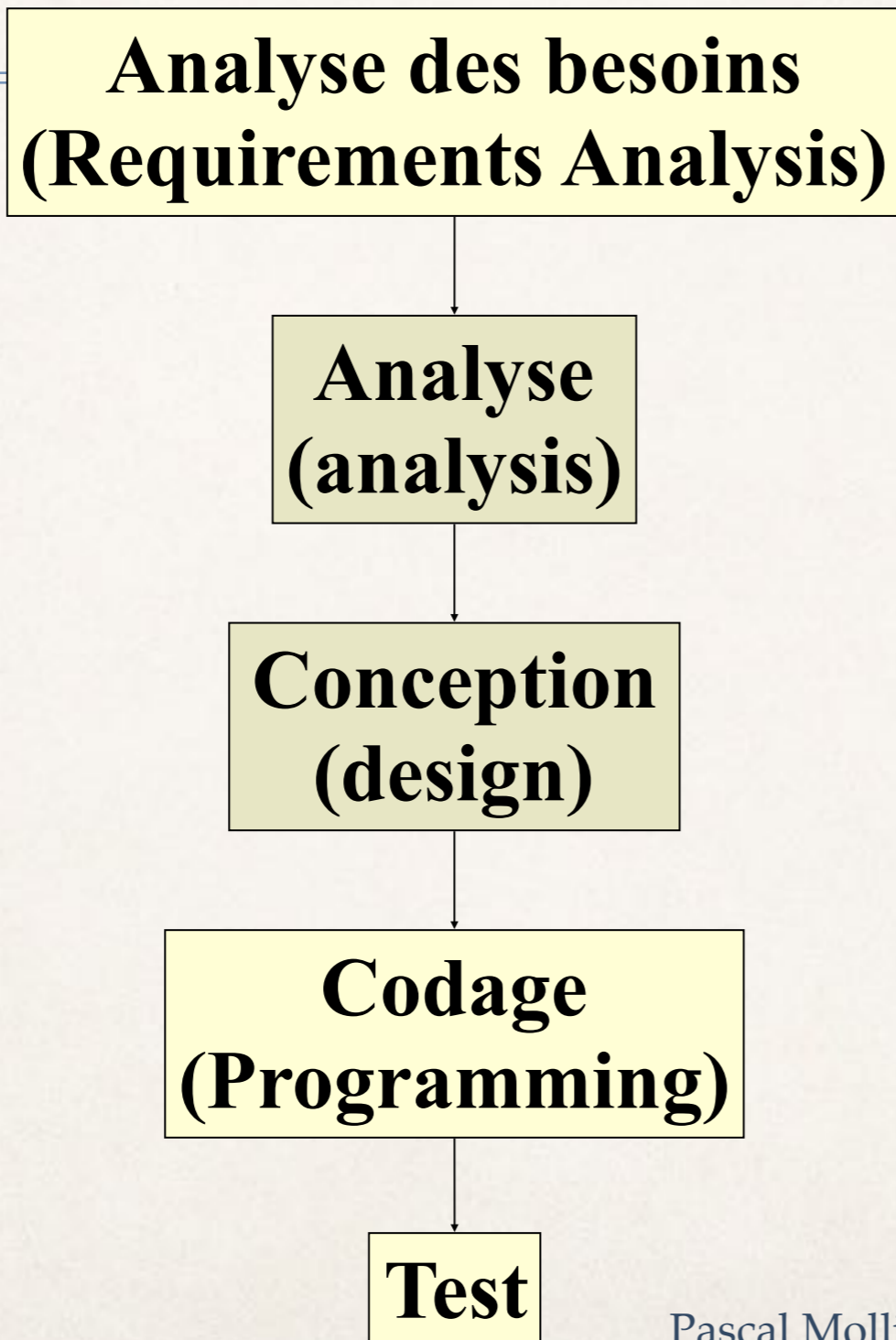


# Analyse

---

- ✓ Indépendante de l'implémentation !
  - ✓ Vise à circonscrire l'application
    - Quelles sont ses *limites*? les *exigences* auxquelles elle doit répondre?
      - Identification des classes et des relations
      - Description des collaborations entre les objets des différentes classes
- ➡ Diagrammes de cas d'utilisation, de classes, de séquences, d'activités.

# Introduction et vue d'ensemble



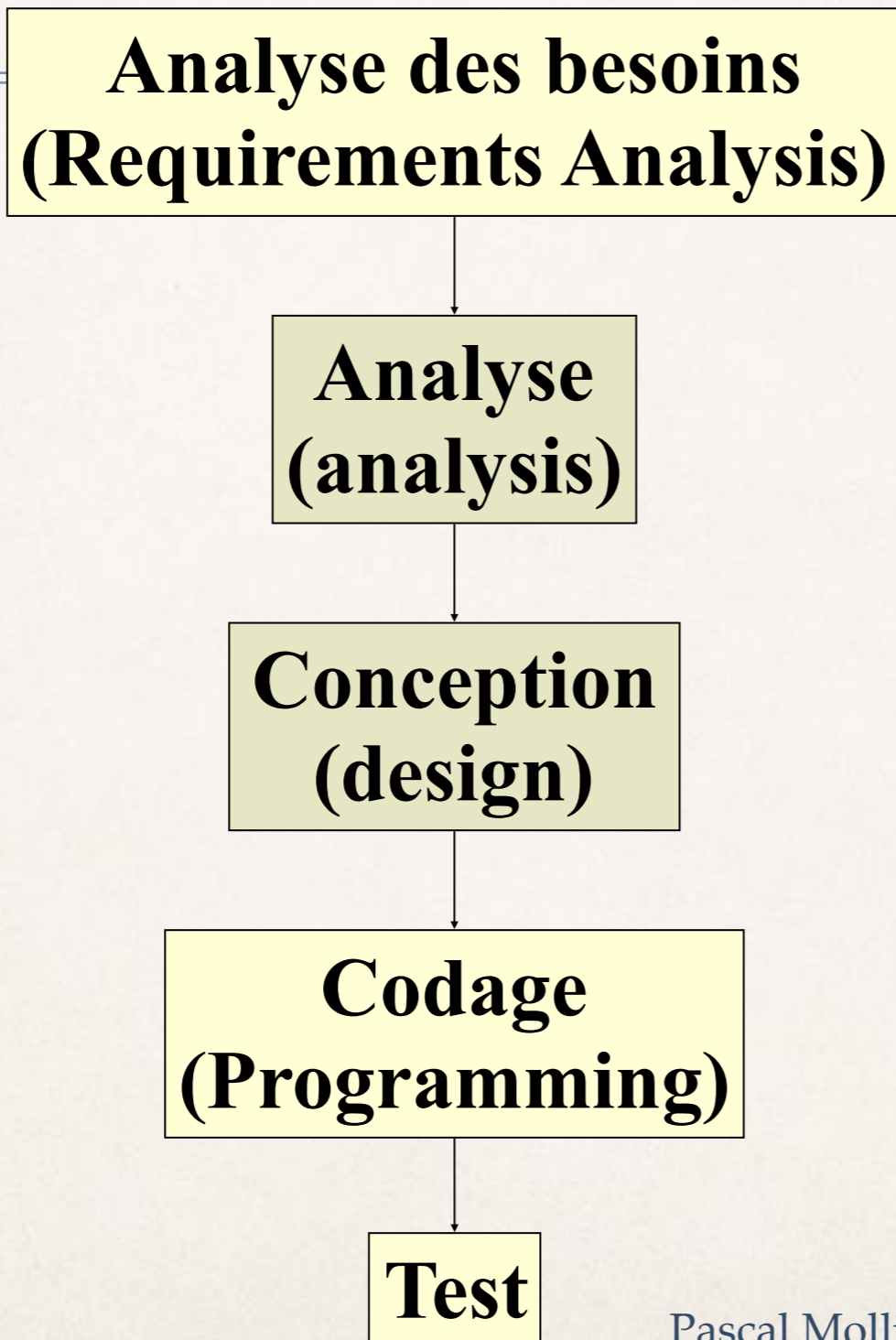
# Conception (design)

---

- ✓ Prise en compte de l'architecture informatique
- ✓ Classes techniques pour gérer l'interface graphique, la distribution, la persistance, la concurrence, ...
- Diagramme de classes, de séquences, *de composant, de déploiement, d'états*

Importance des patterns de conception pour construire des codes «pragmatiques»

# Introduction et vue d'ensemble



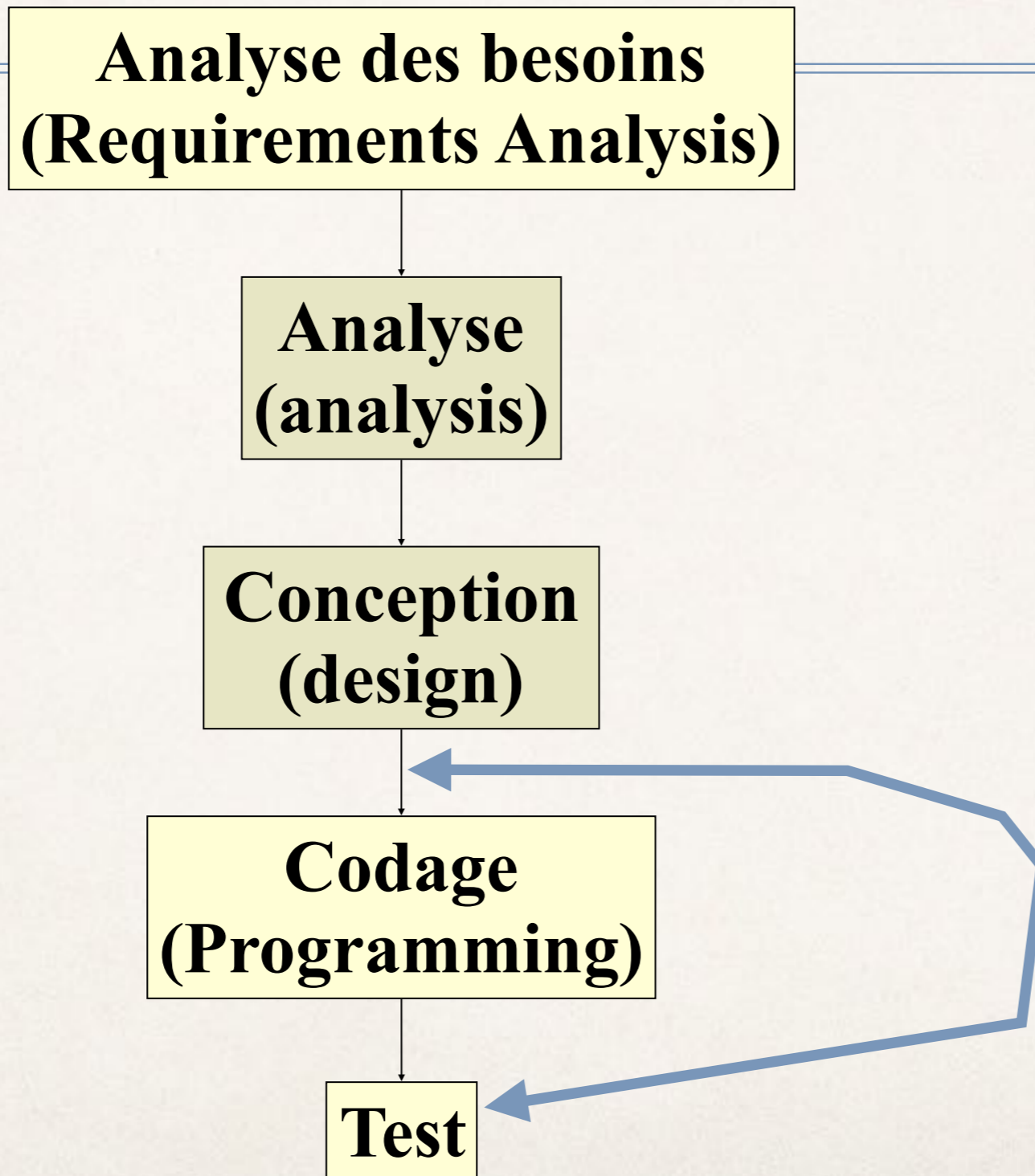
# Programmation

---

- ✓ Conversion des classes de conception vers les langages cibles: java, sql, c++, IDL
- ✓ Conversion des classes persistantes vers les modèles de persistance (SGBD, BDOO, Langages persistants)
- ✓ etc ...



# Introduction et vue d'ensemble

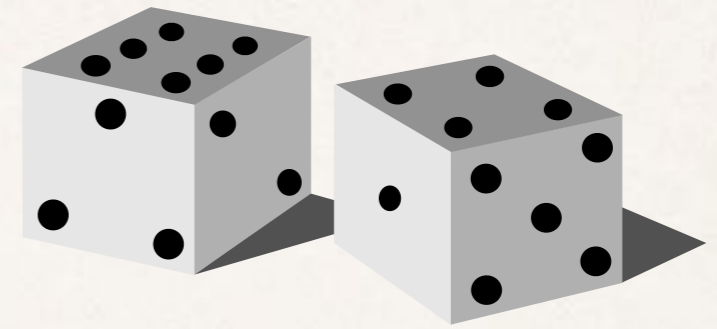


# Tests

---

- ✓ Tests unitaires: diagrammes de classes
- ✓ Tests d'intégration: diagrammes de séquences, d'activités
- ✓ Test du système: diagramme de cas d'utilisation, de séquences système.

Préparation et Evaluation des tests tout au long du cycle de vie du logiciel



# Mise en oeuvre sur un exemple

---

- ❖ Un jeu de dés
- ❖ Le joueur lance 10 x 2 dés
- ❖ Si le total d'un lancé fait 7, le joueur marque 10 points ajoutés à son score
- ❖ En fin de partie, son score est inscrit dans le tableau des scores.

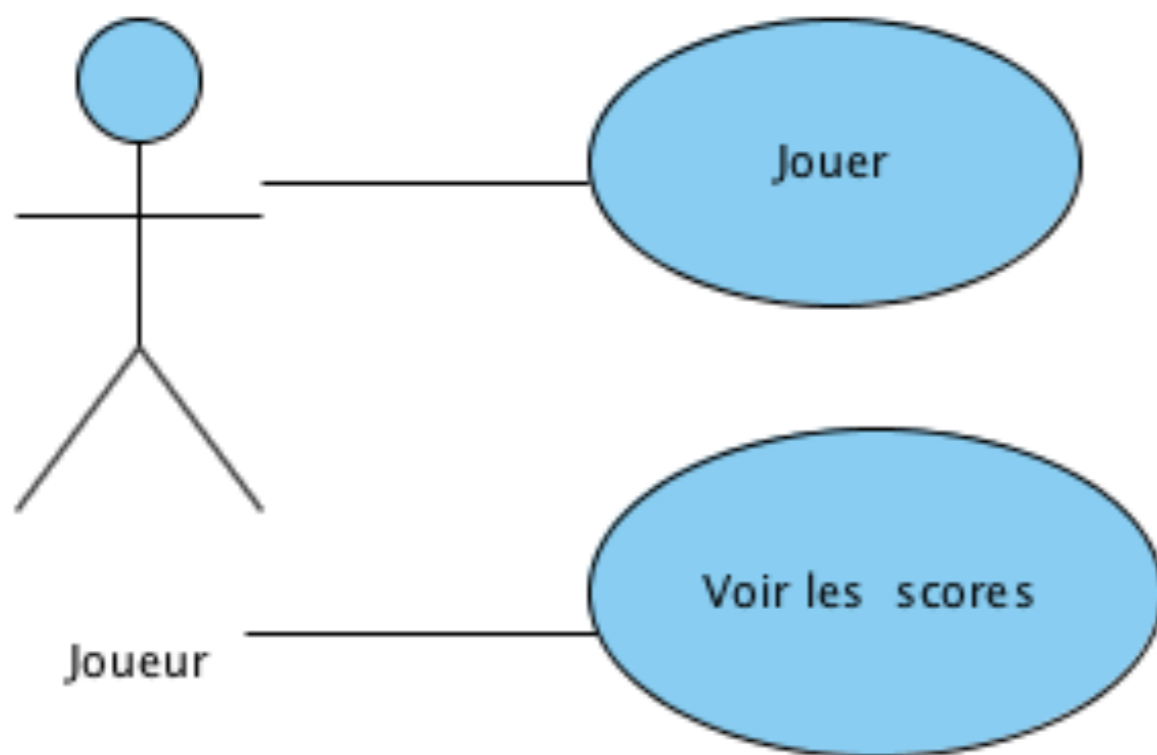
# Analyse des besoins

---

- ❖ Identifier les acteurs.
- ❖ Identifier les cas d'utilisations possibles du système
  - Ses fonctionnalités externes !

# Premiers Cas d'utilisation

---



## ❖ Jouer:

- Acteur: Joueur
- Descr: Le joueur prend 10x les dés, à chaque fois que le total fait 7, +10pts

## ❖ Voir les scores

- Acteur: Joueur
- Descr: Le joueur consulte en read only les scores précédents obtenus par les joueurs

# Use Case

---

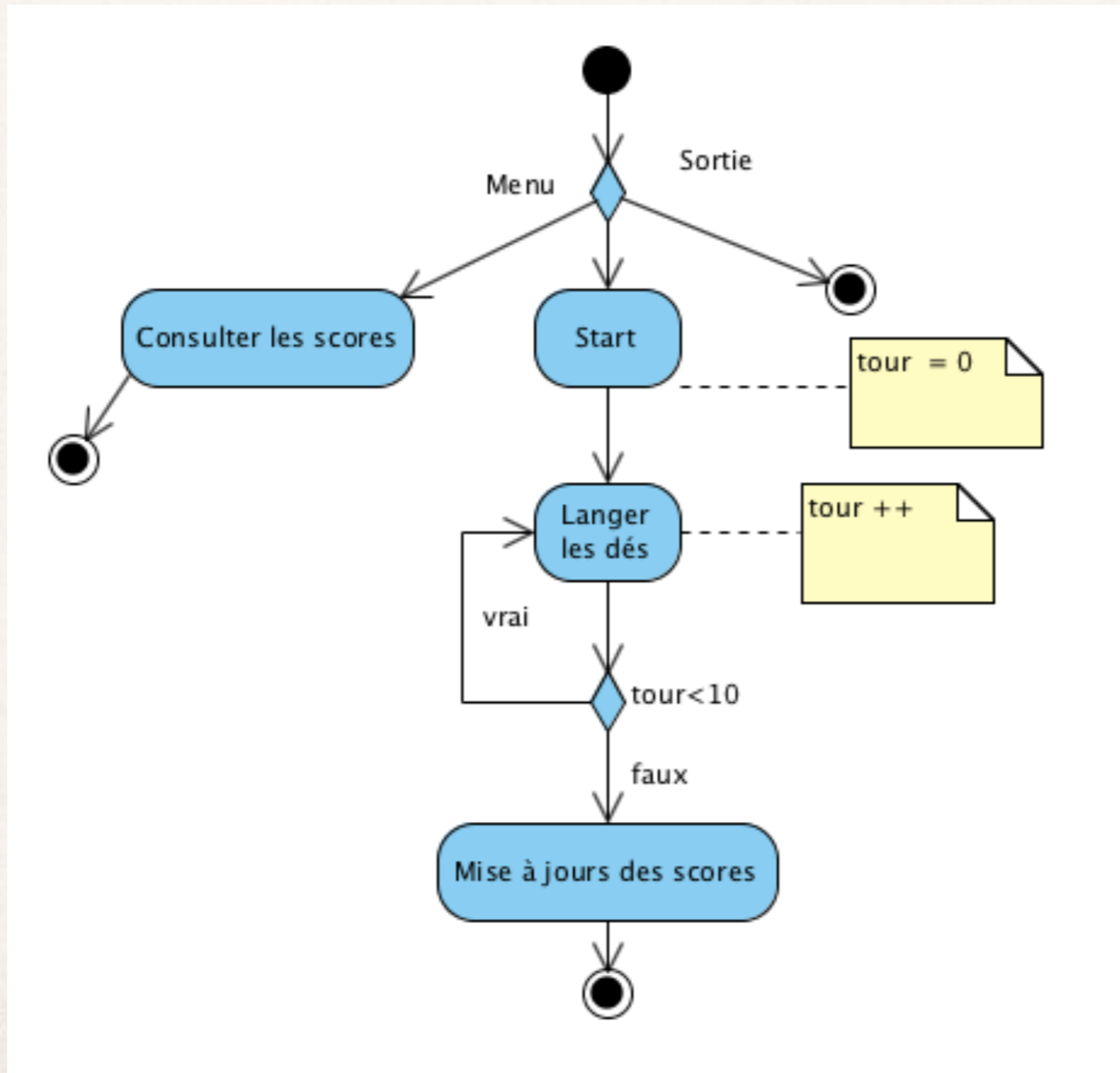
- ❖ Diagramme extrêmement important !
- ❖ IMHO, Il doit figurer dans un cahier des charges
- ❖ IL DOIT ÊTRE COMMENTÉ de manière rigoureuse !
- ❖ Il sert de référence pour toute la suite des opérations.

# Diagramme d'activité

---

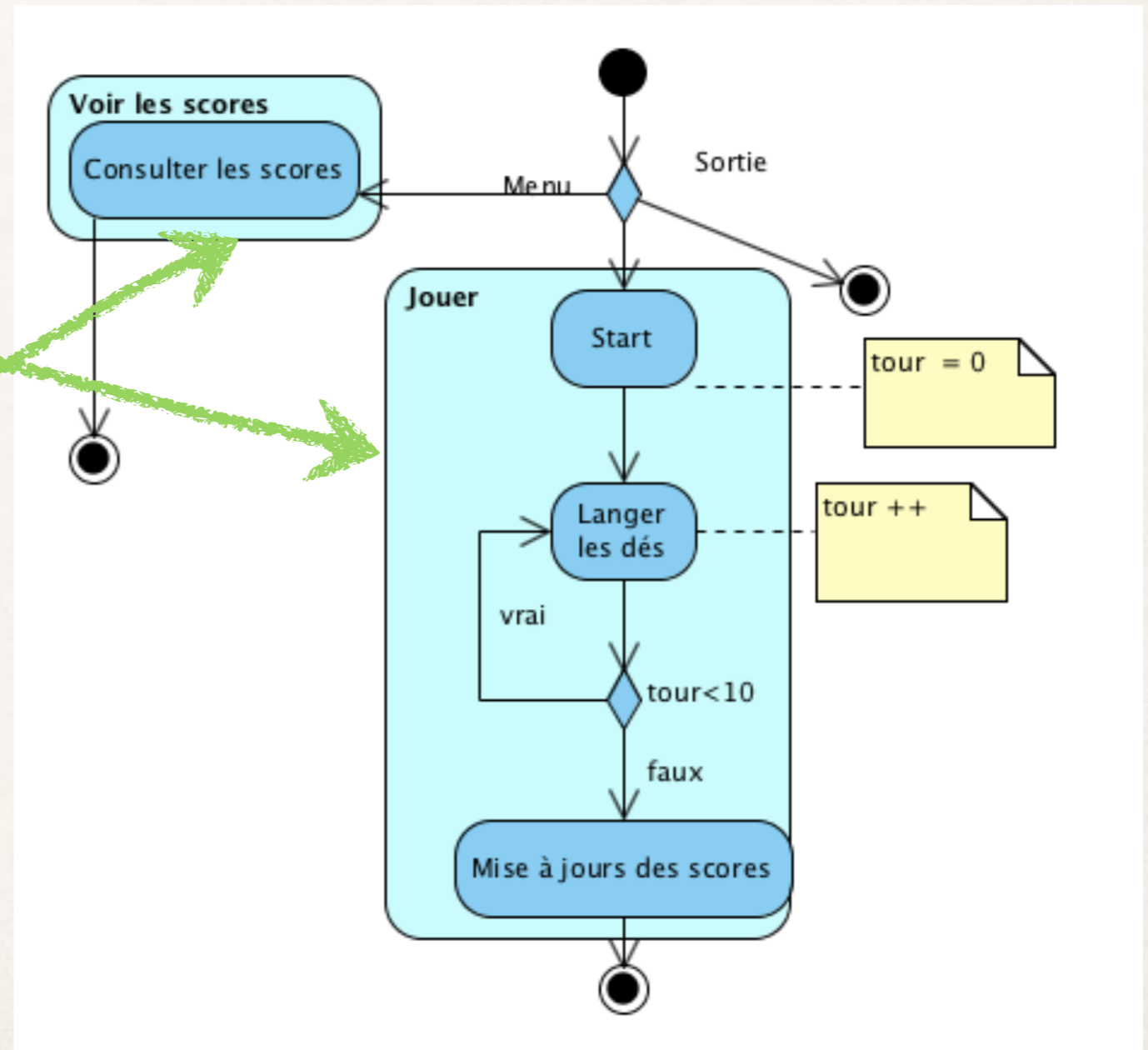
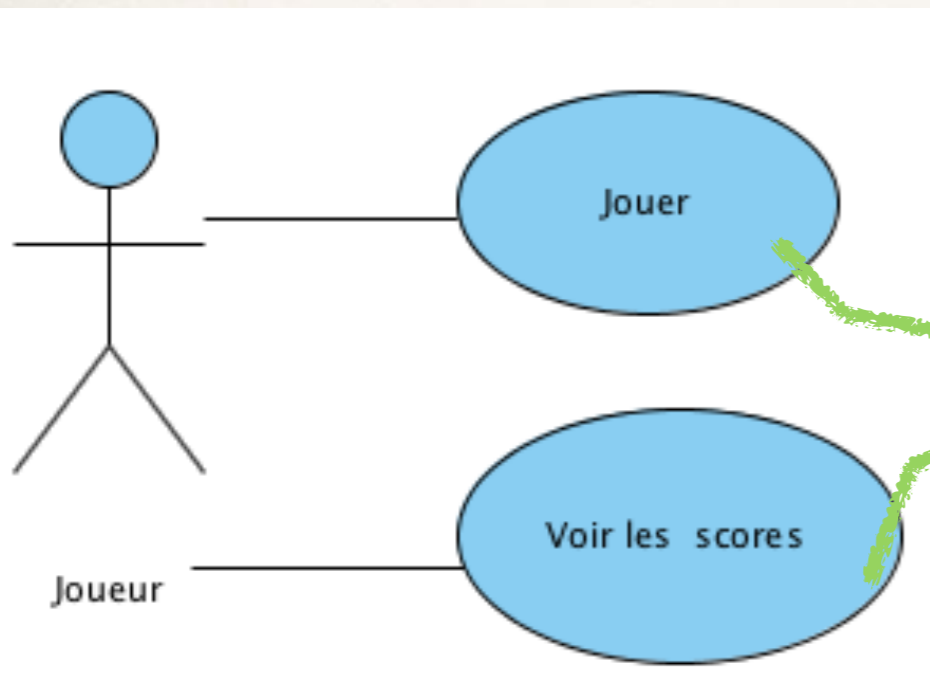
- ❖ Identifier les activités (en s'appuyant sur les cas d'utilisation)
- ❖ Identifier les transitions entre activités et donc entre les cas d'utilisation

# Diagramme d'activité

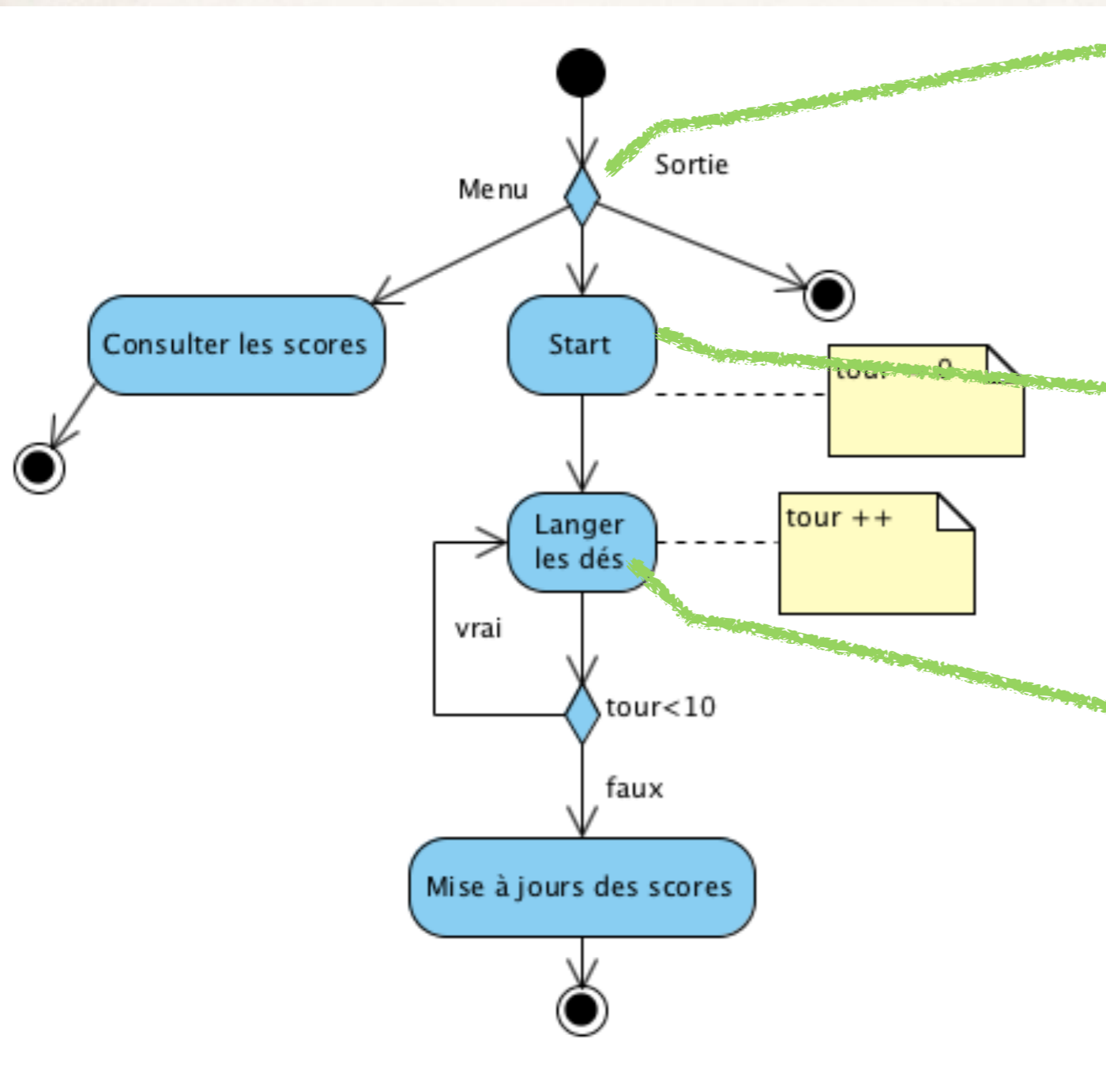




# Diagramme d'activité



# Diagramme d'activité



# Analyse

---

- ❖ Indépendante de l'implémentation
- Déterminer les classes d'objets du domaine étudié et concernées par l'application
  - premier diagramme de classe
- Modéliser la dynamique du système
  - diagrammes de séquence.

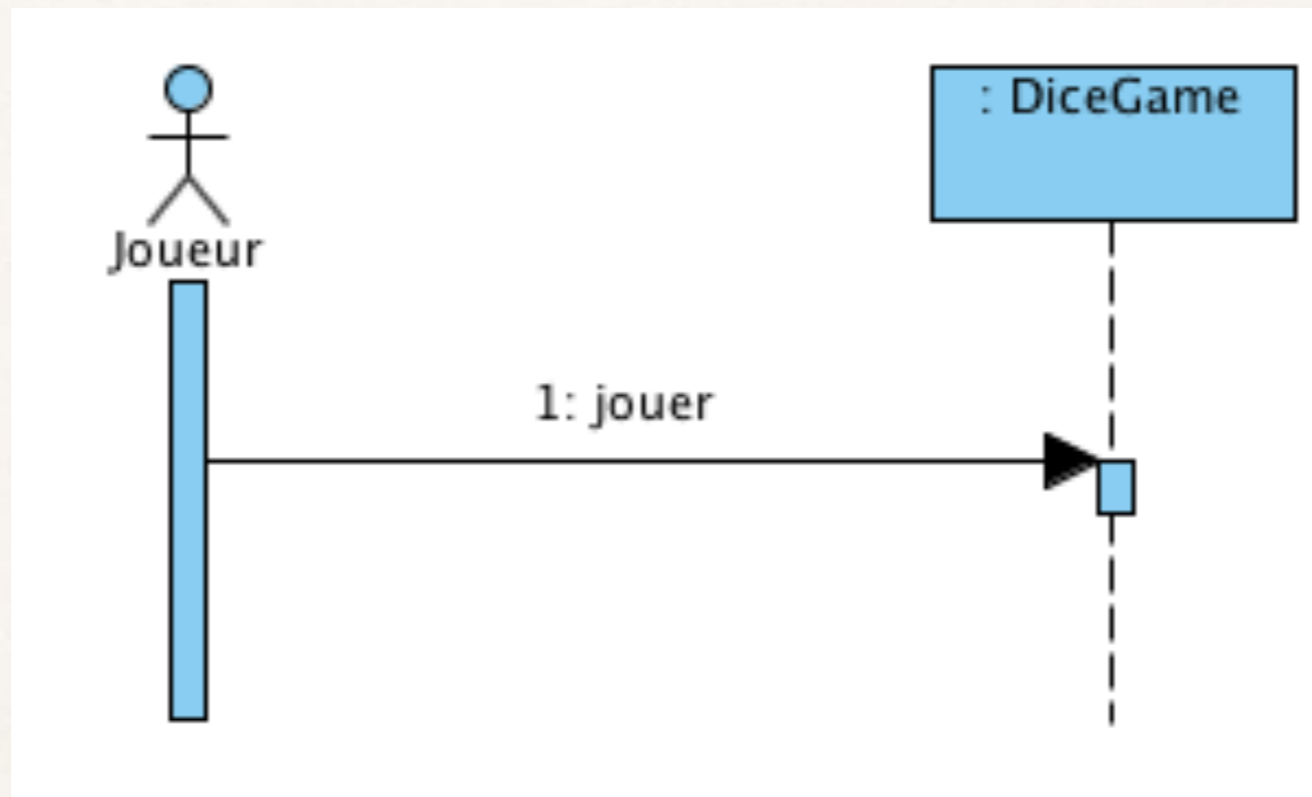
# Diagramme de séquences

---

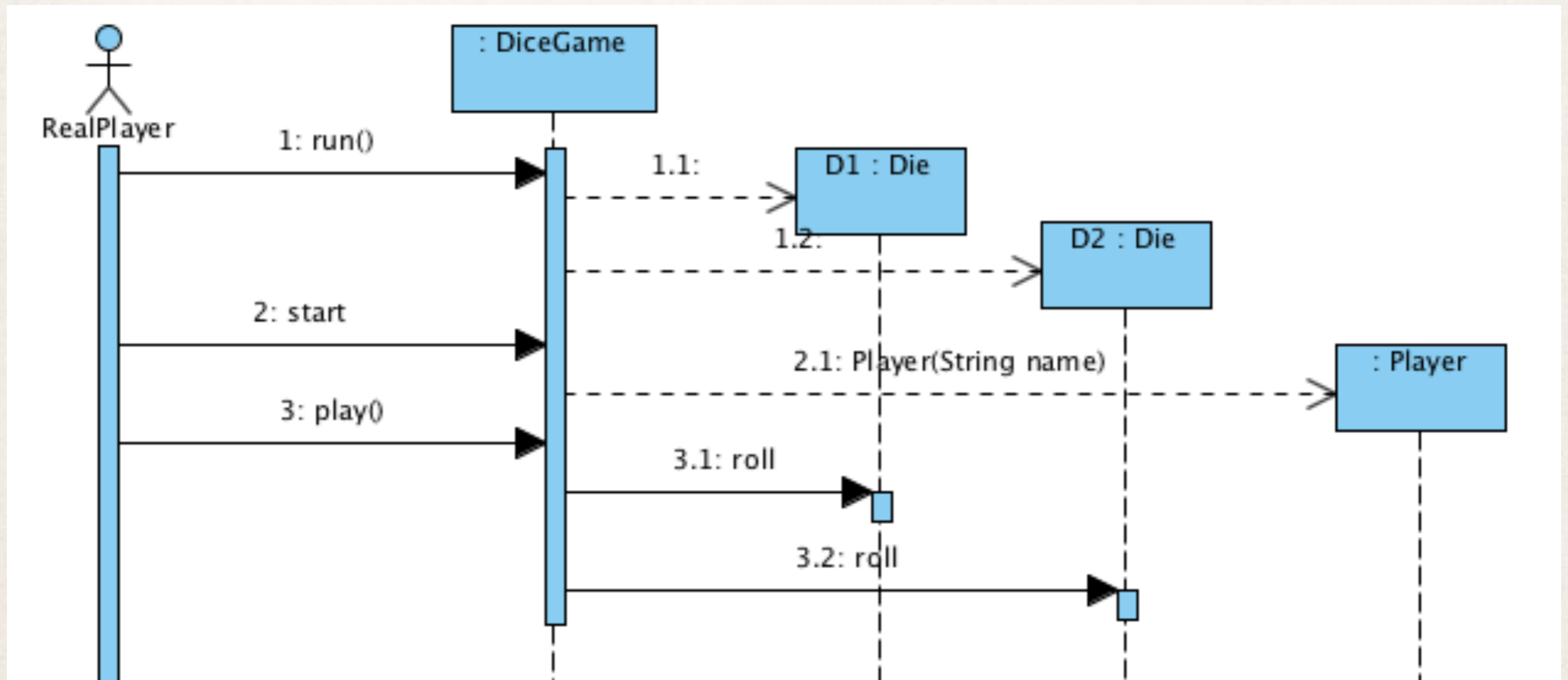
- ❖ Modélise la dynamique
- ❖ Focalise sur l'enchaînement des messages
- ❖ Permet d'identifier les objets, les messages et leur ordonnancement.

# Diagramme de séquences

---



# Diagramme de séquences

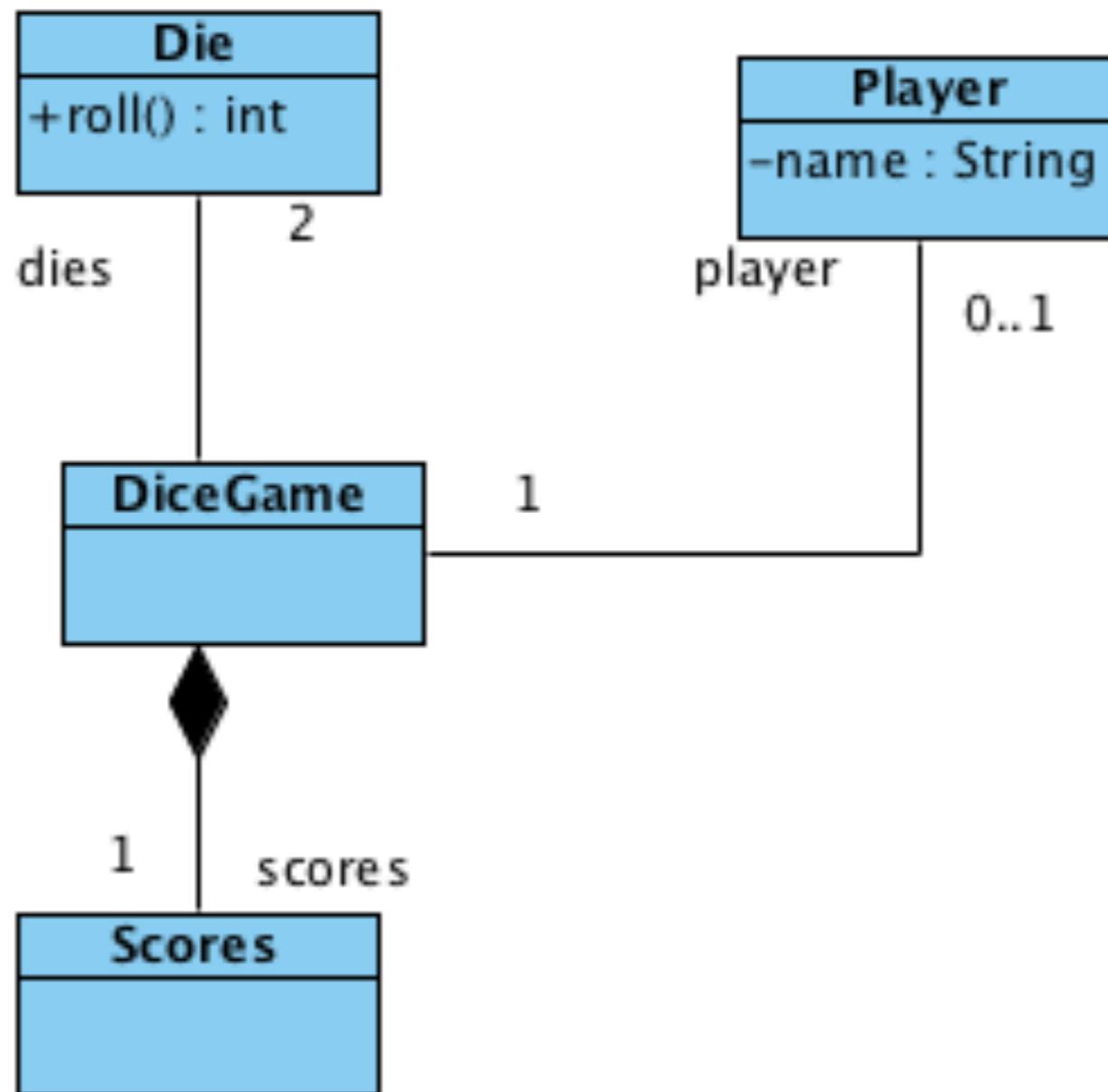


# Diagramme de classes

---

- ❖ Identifier les classes
- ❖ Identifier les relations statiques et dynamiques entre les classes
- ❖ Déterminer les cardinalités des relations
- ❖ Déterminer les attributs des classes
- ❖ Déterminer les méthodes et leurs paramètres

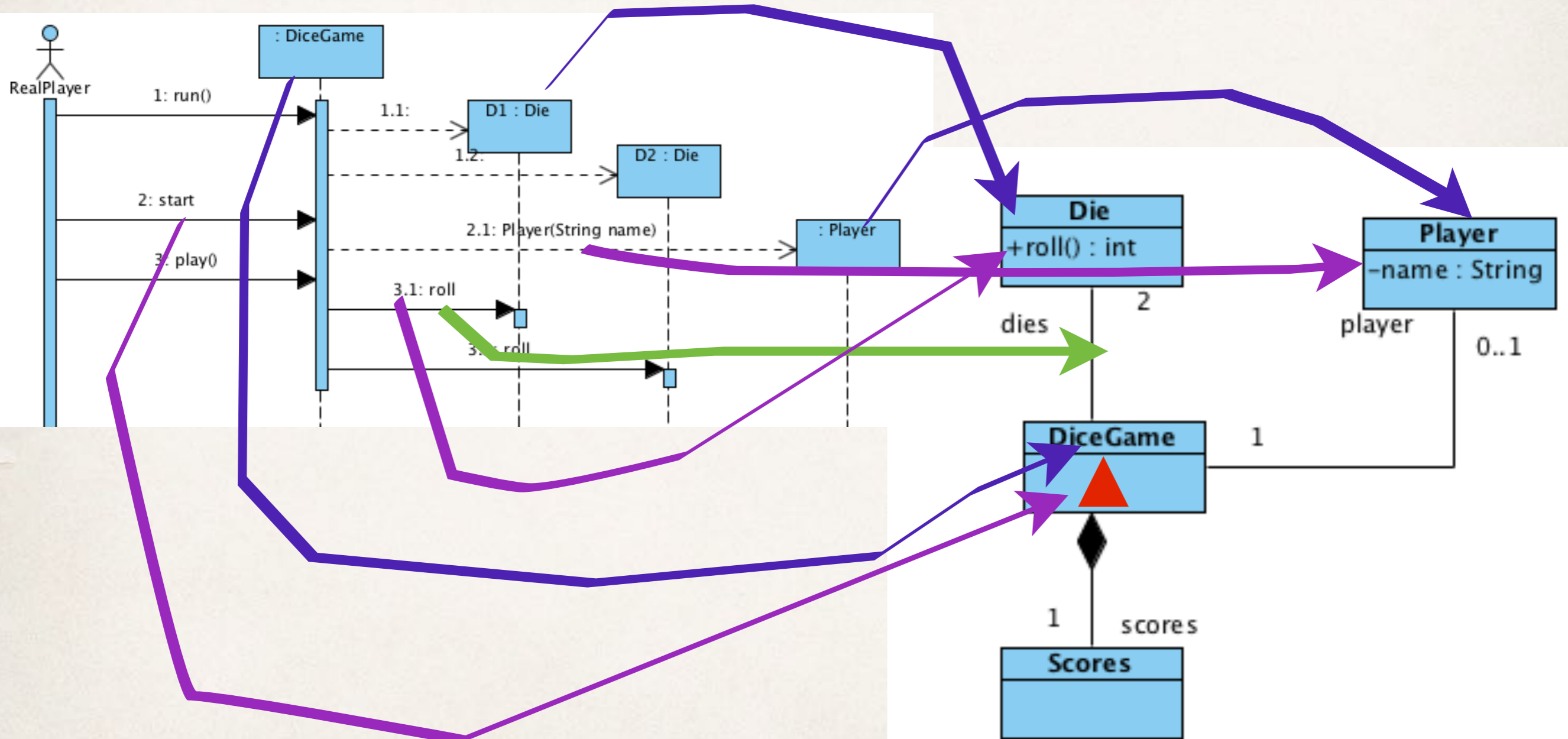
# Diagramme de classes



Attention  
diagramme non  
complet!



# Diagramme de classes contre le diagramme de séquence

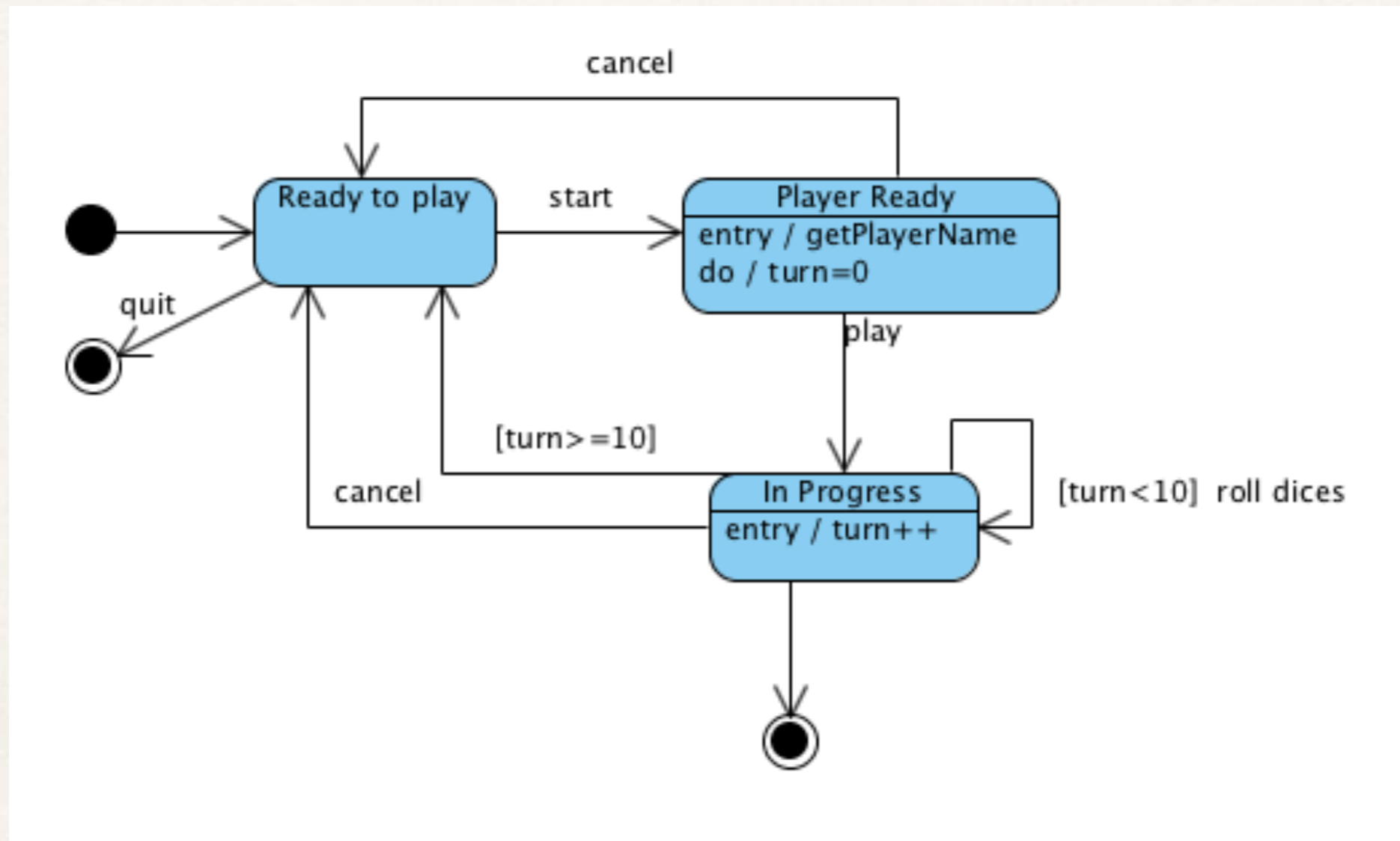


# Diagramme d'états

---

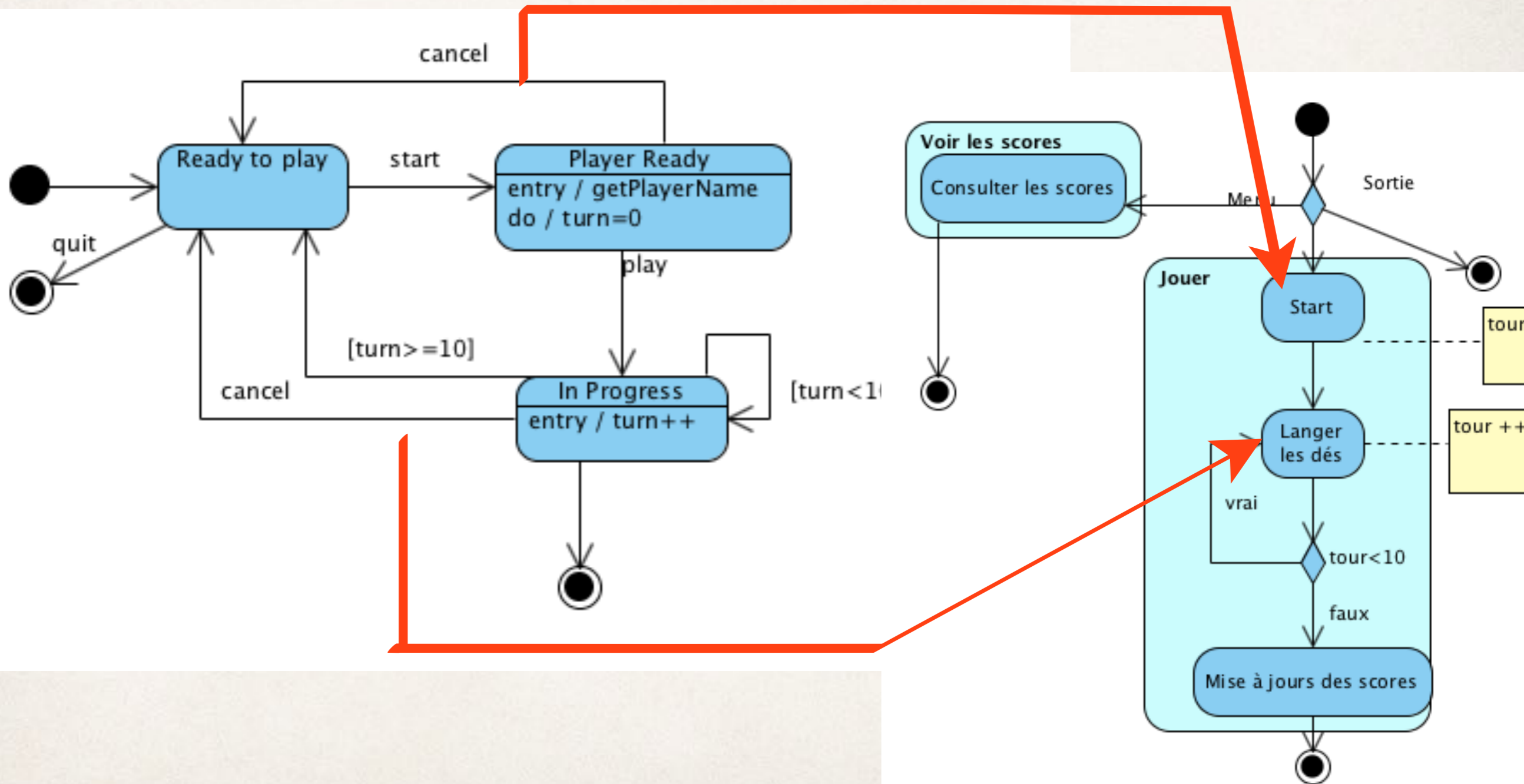
- ❖ Identifier les états d'un **objet**
- ❖ Identifier les transitions entre les états

# Diagramme d'état d'un objet «partie»

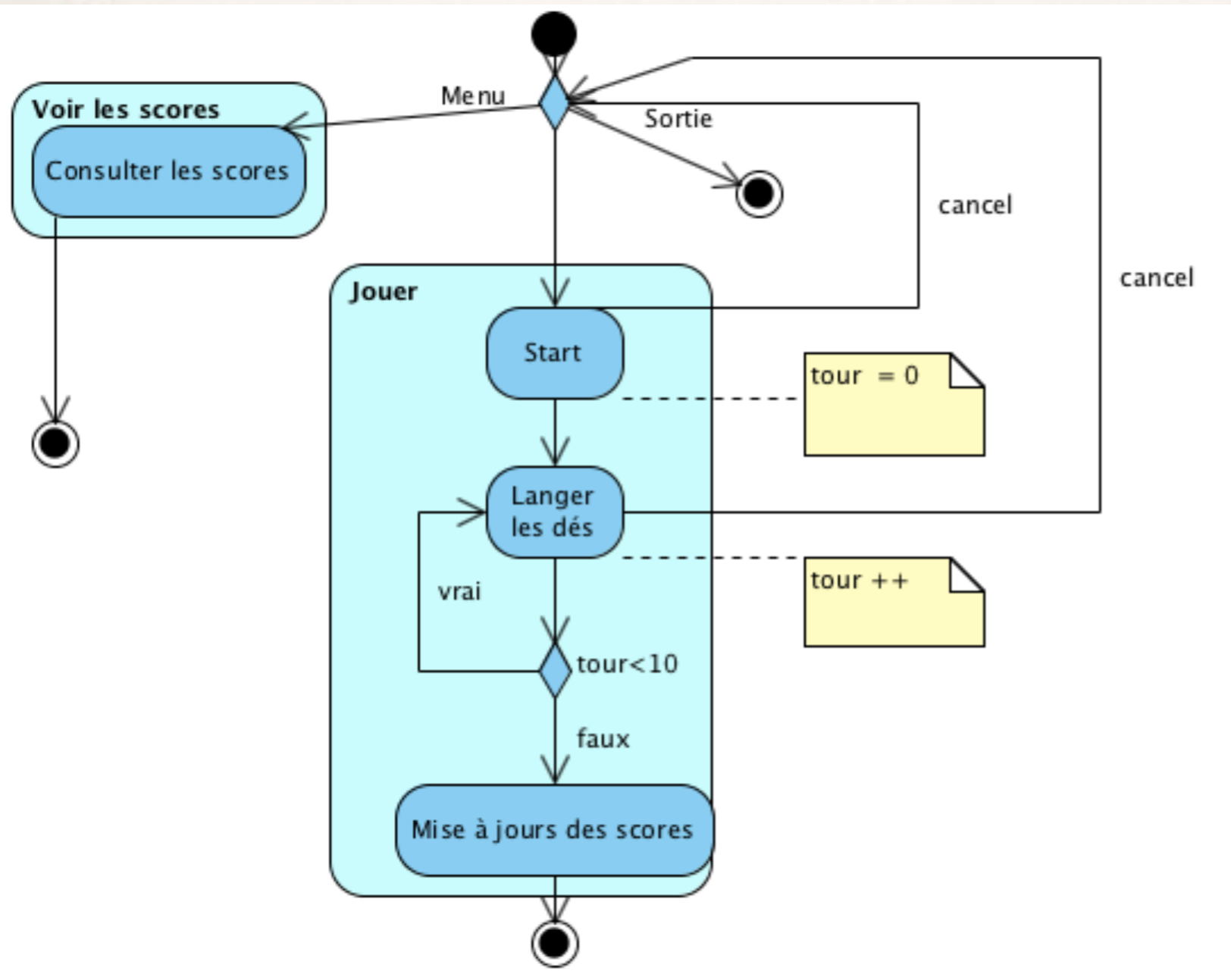


# Diagramme d'état d'un objet «partie» & diagramme d'activité

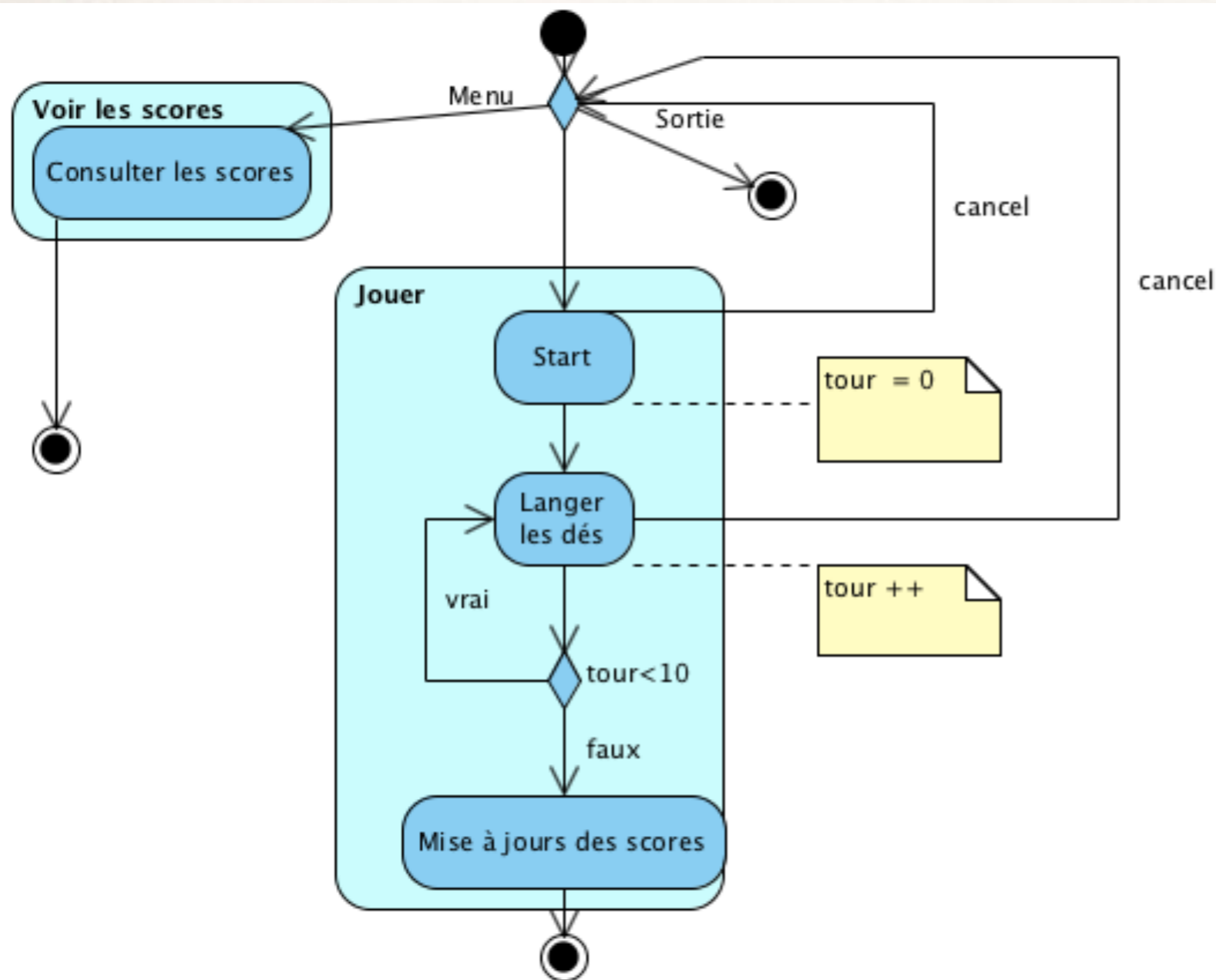
Cancel ?



# Mettre à jour les schémas



# Vérifier la cohérence !



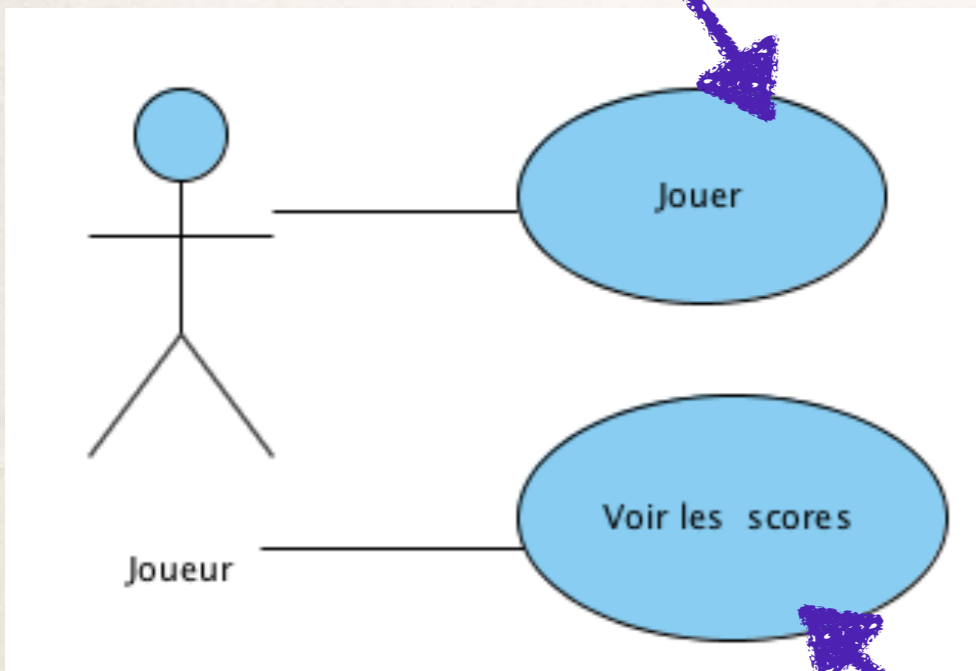
# Analyse terminée ?

---

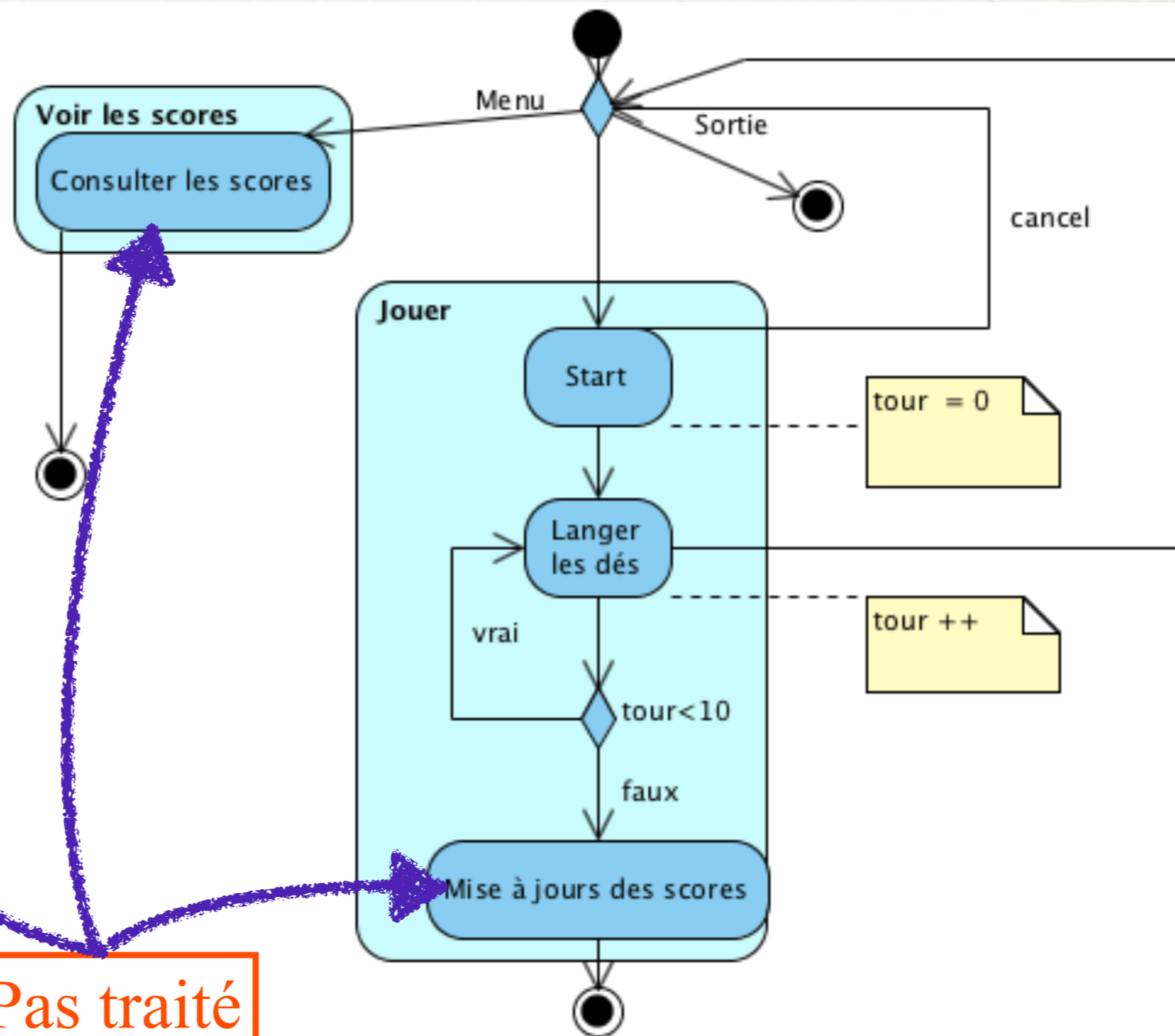
- ❖ Vérifier la couverture des diagrammes «use-case» et d'activités...
- ❖ Use case « Voir les scores » ?
- ❖ Use case « Jouer » partiellement traité.

# Couverture des diagrammes

Partiellement traité

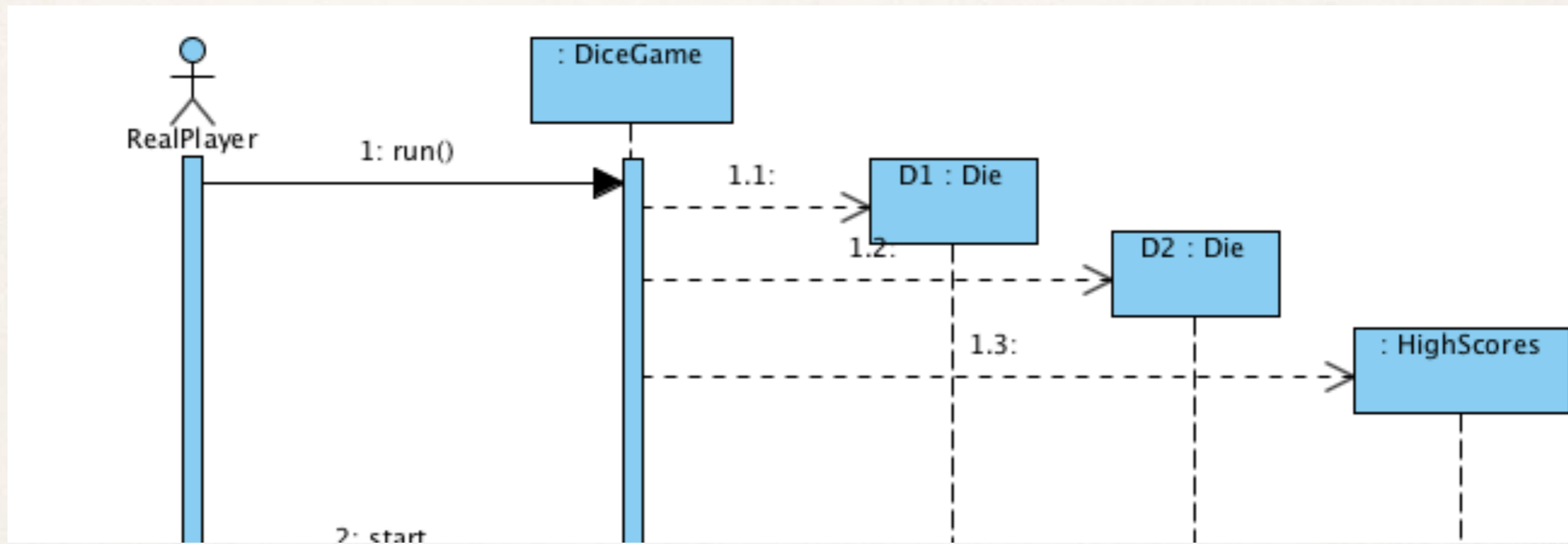


Pas traité



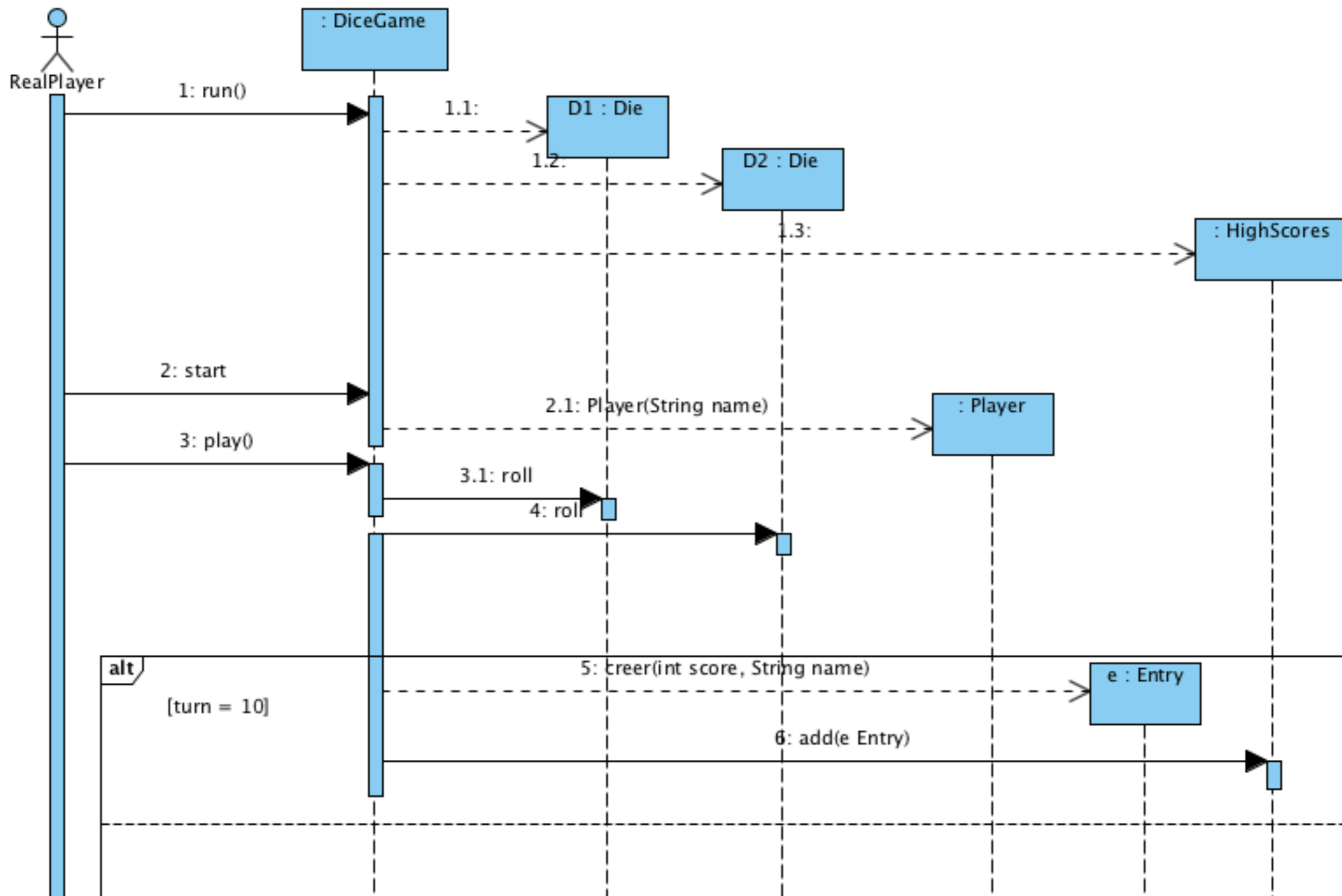


# Diagramme de séquences complété

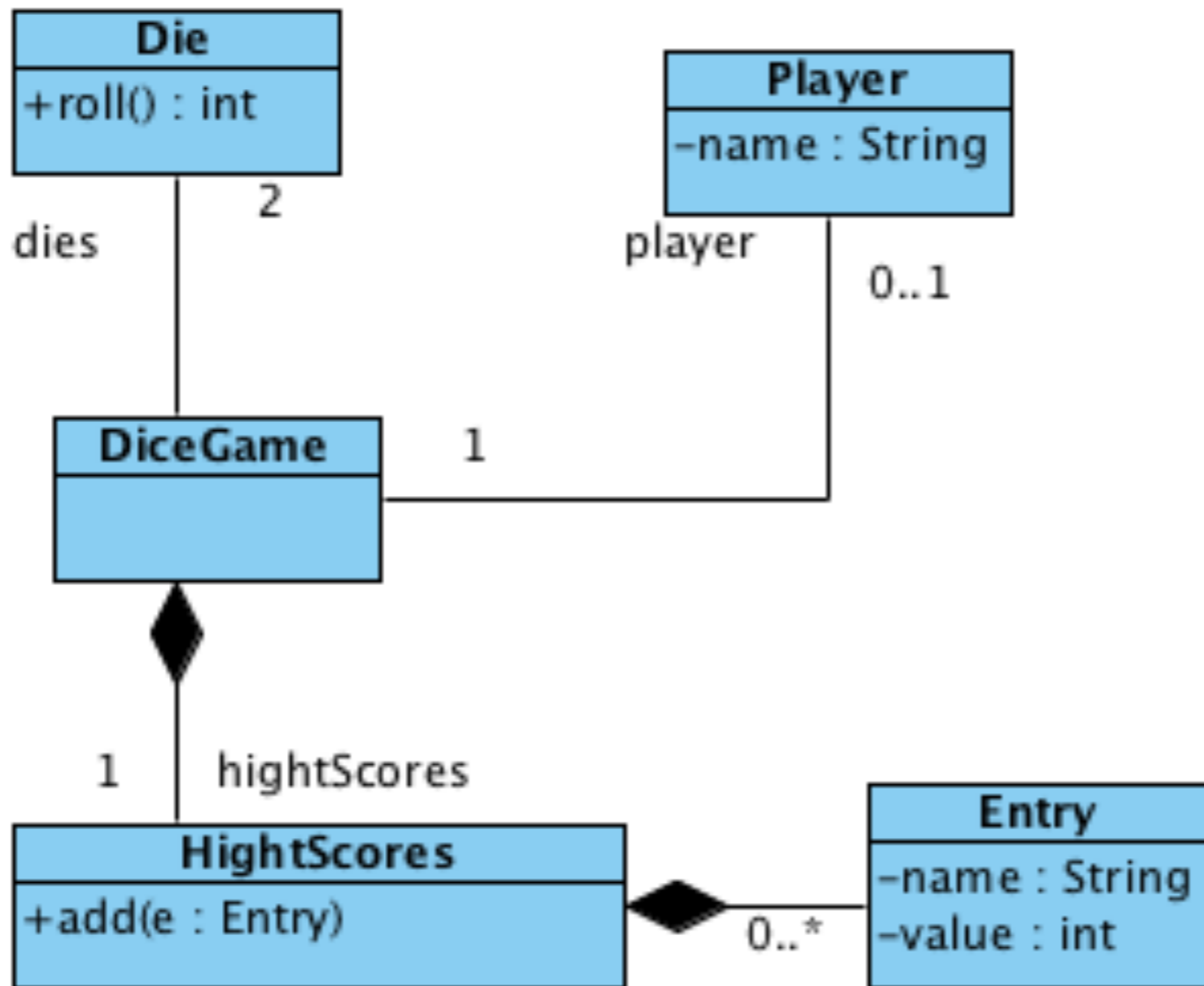


d'après Pascal Molli, molli@loria.fr

# Diagramme de séquence modifié



# Diagramme de classes



# Fin de l'analyse ?

---

- ❖ Couverture « à peu près » bonne
- ❖ Cohérence entre les schémas correctes
  - La dynamique manque de détail (dynamique du cancel ?)
  - Les schémas sont trop peu expliqués...
  - Les diagrammes de séquence du jeu ne sont pas assez détaillés : manque quelques méthodes...

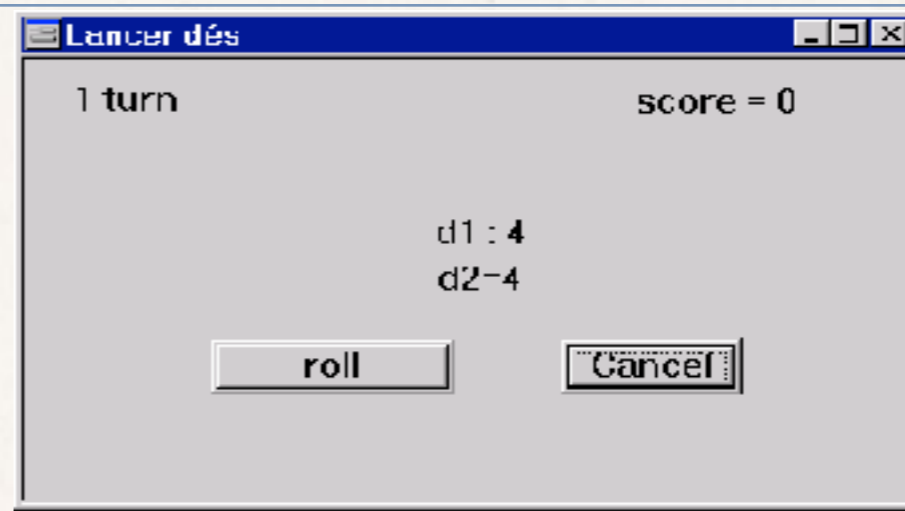
# Conception

---

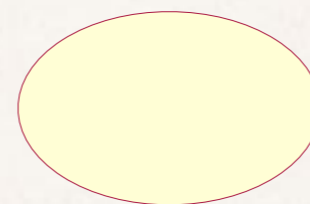
- ❖ Définir l'architecture
- ❖ Rajouter les classes techniques permettant d'implémenter cette architecture !
- ❖ Prendre en compte l'implémentation
  - ➔ Gérer la partie interface graphique
  - ➔ Gérer la persistance

# Conception de l'architecture

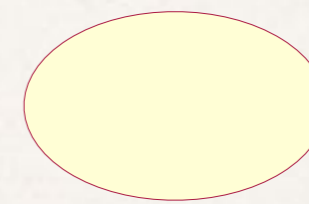
Présentation



Applicatif



Jouer



Voir les scores

Persistance



Fichier ou BDD

Couches  
(Layers)



# Architecture en couches ...

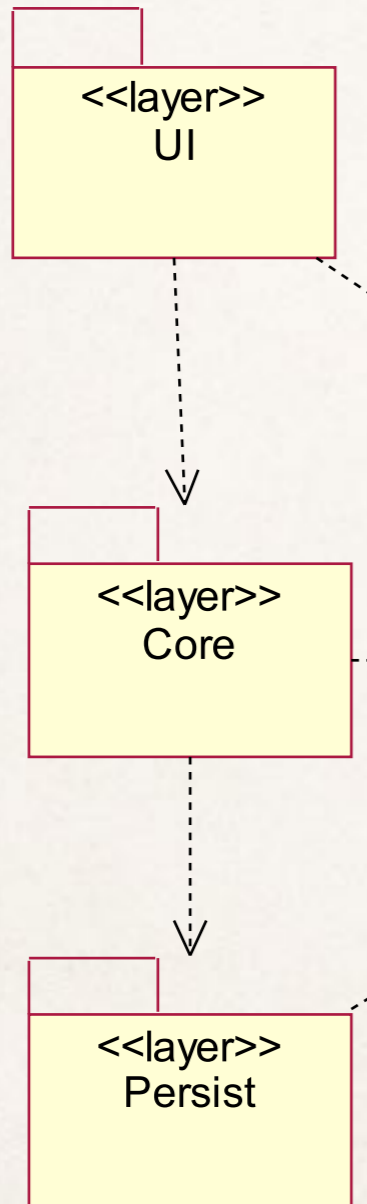
---

- ❖ Une architecture possible...
- ❖ Les couches doivent être le plus indépendantes possible
  - ➔ « Découpler » les couches en s'appuyant sur des interfaces et des classes abstraites
- ❖ Il ne doit exister qu'une seule orientation des dépendances entre couches.

Des solutions à des problèmes récurrents : les «patterns»

# Découpage en « packages » logiques

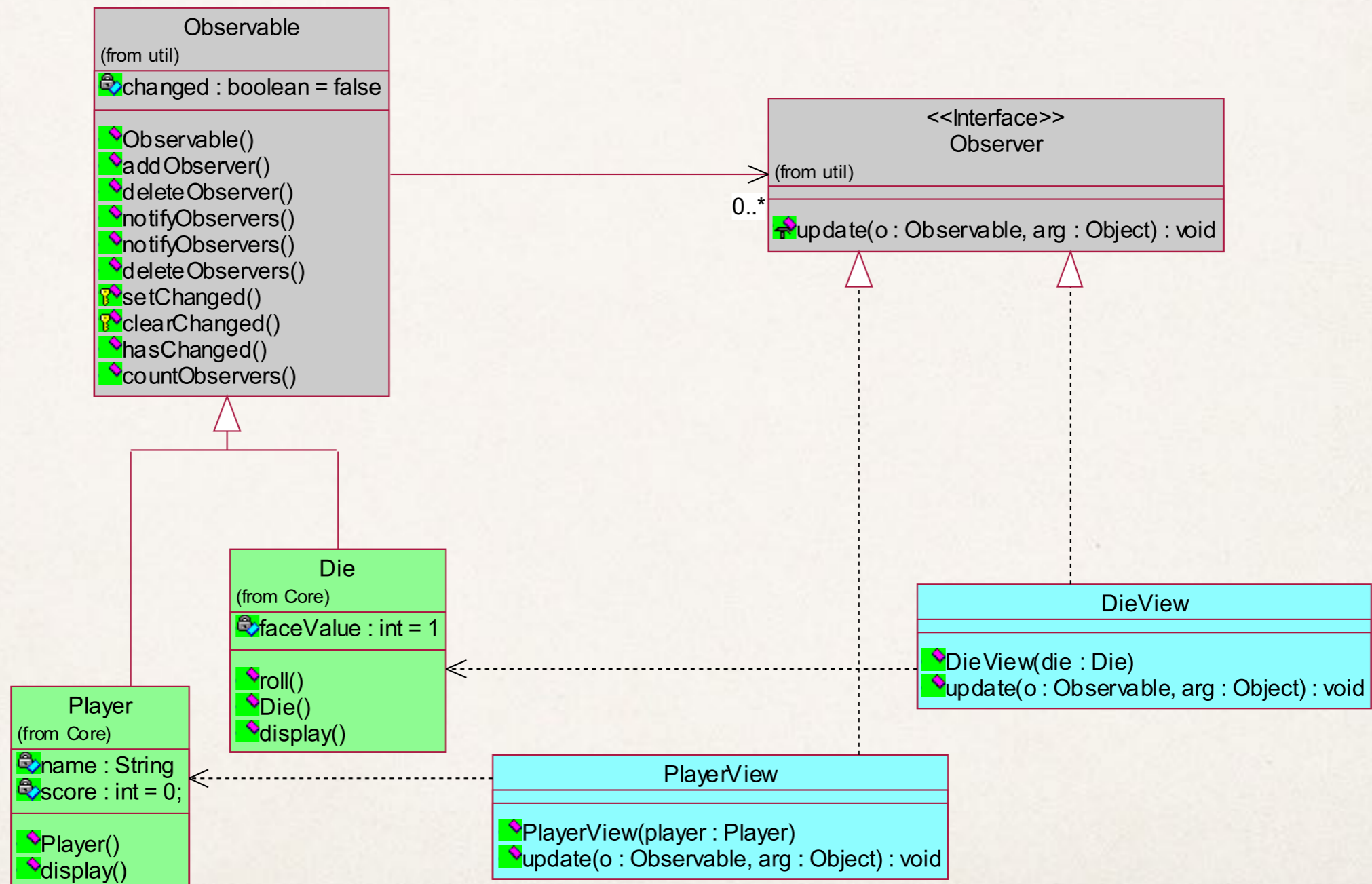
---



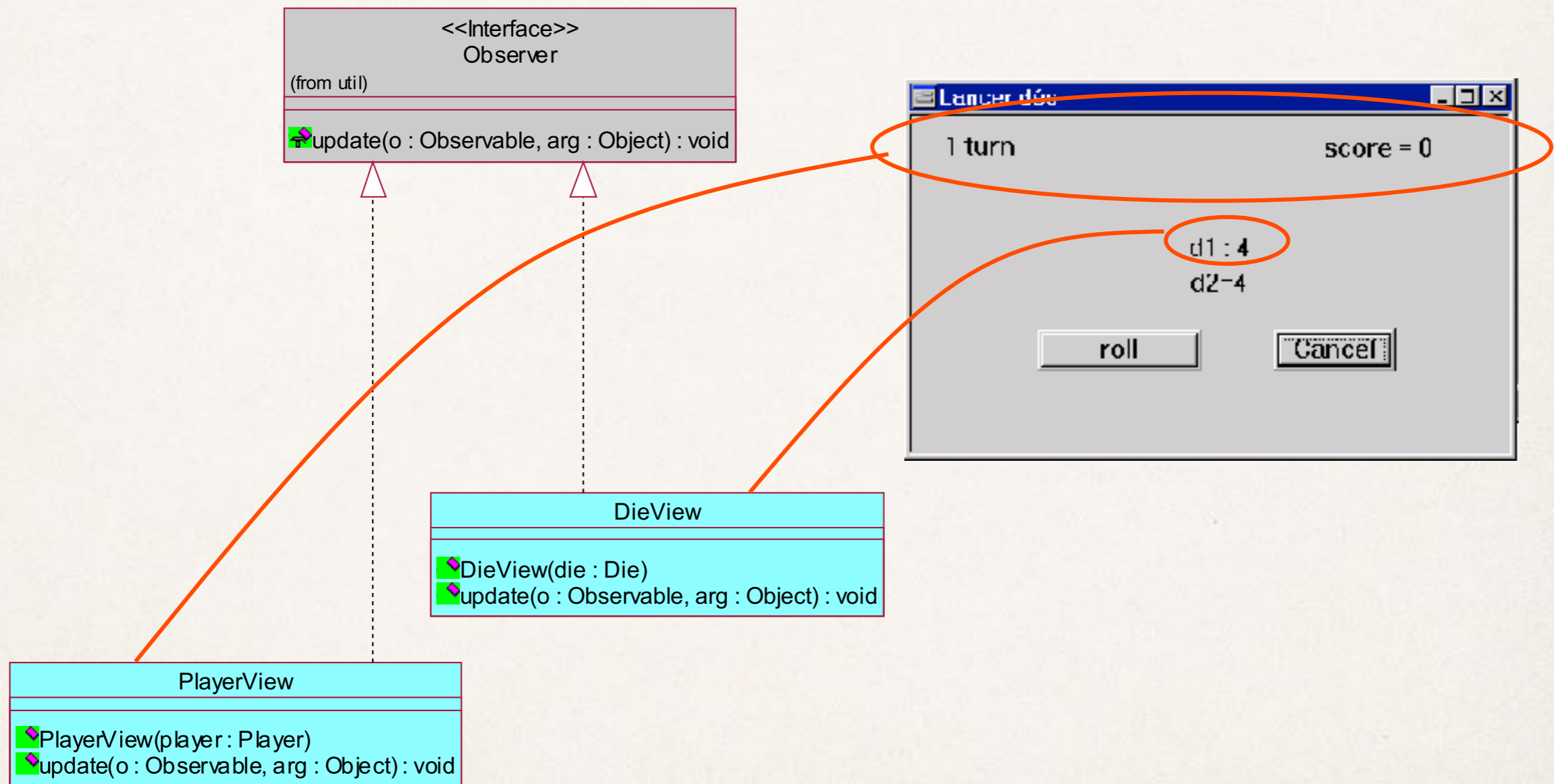
- ❖ Mapper l'architecture sur des packages « layer »
- ❖ Exprimer les dépendances



# Découplage interface graphique

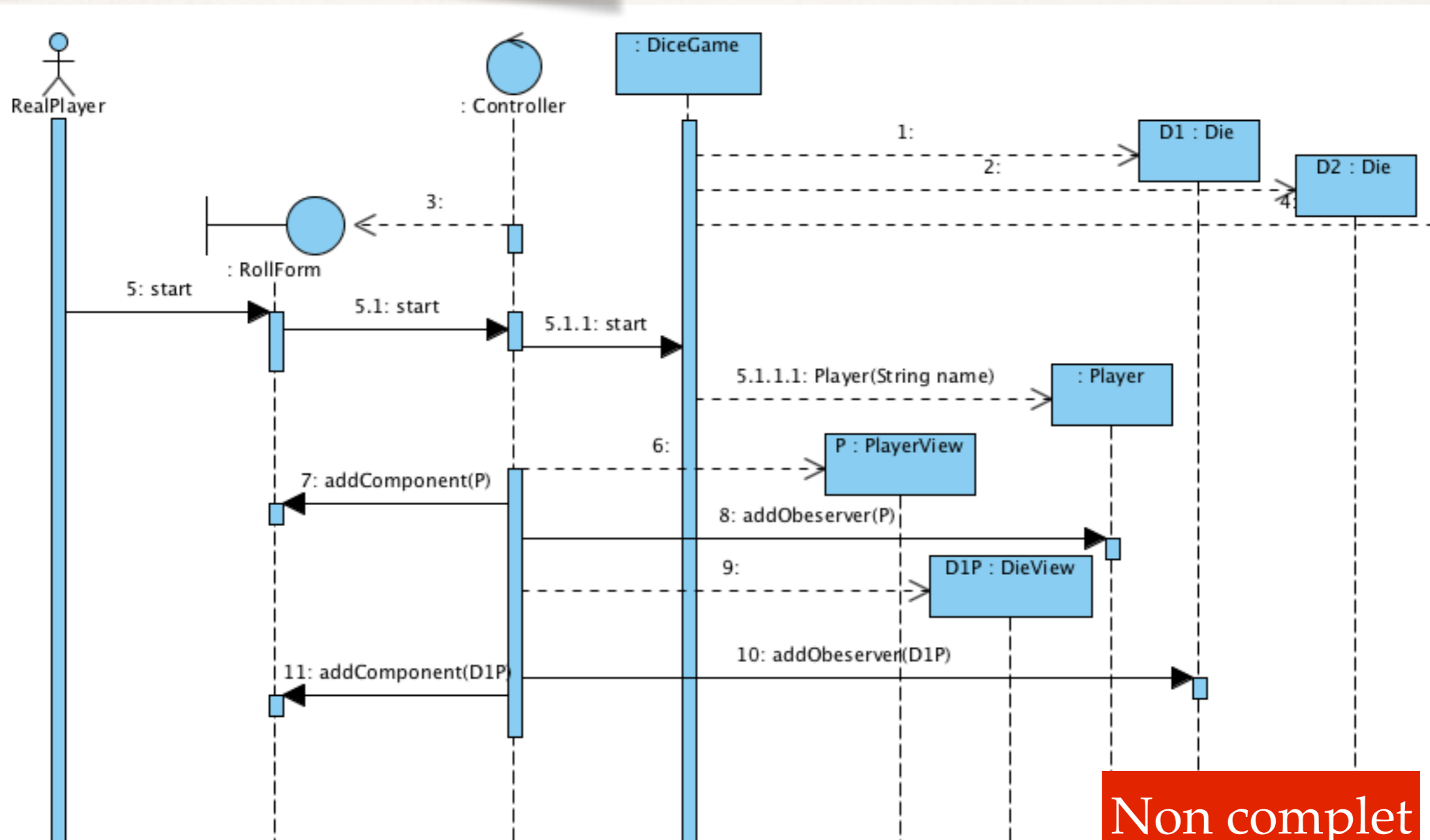


# Vues ?



# MVC en action : 1 mise en place

Approfondissement de ce type d'architecture dans un prochain cours dédié.



# Layer « core »/Couche applicative

---

- ❖ Les classes représentant la logique de notre application.
- ❖ En fait, les classes d'analyses «revisitées» en vue de la réalisation

Démarche abordée l'an dernier-  
Nous la conforterons dans la suite des cours et TDs  
dans une approche «Agile» et «Pragmatique».

# Layer « Persist » : Couche de persistance

---

- ❖ Classes techniques de persistance
- ❖ Assurer l'indépendance Core / Persist
  - ➔ pouvoir changer de « persistent engine »
- ❖ Par exemple:
  - ➔ Persistance par « Serialisation »
  - ➔ Persistance via une base de données relationnelle (JDBC).

# Design terminé ?

---

- \* Couverture des fonctionnalités : comparer Use-case et activity diagram...
- \* Cohérence entre les diagrammes ?
  - Quelques incohérences... UI vs Core ; gestion des changements d'états?
  - Indépendance Core / Persist partiellement atteinte...

# Générer le code : code mapping

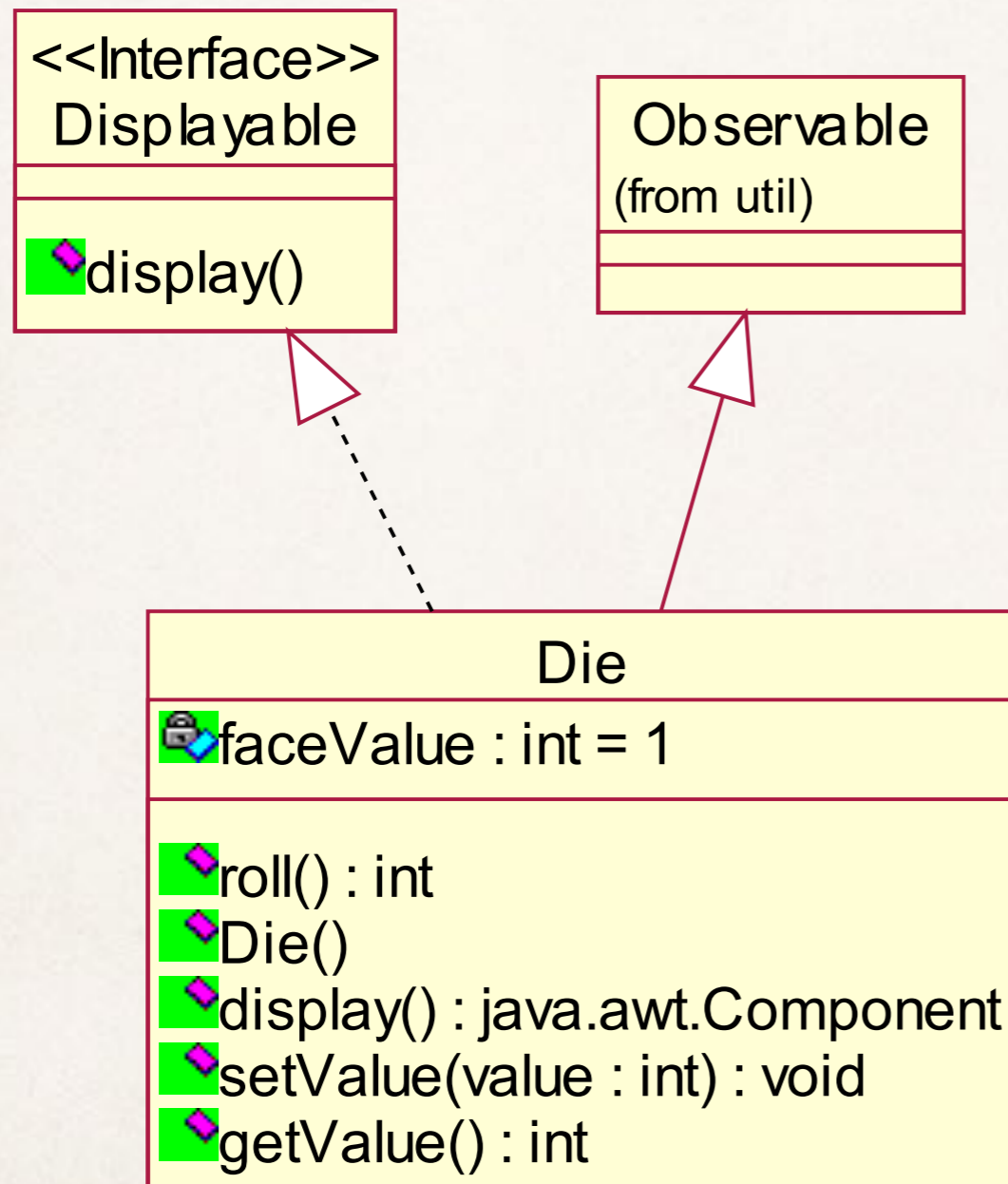
---

- ❖ Mapper vers n'importe quel langage !\*
  - Les langages objets : java, C++, C#, smalltalk
  - Mais aussi les autres: VB, C, Fortran, Php
  - Voir aussi: SQL...

Déjà vu l'an dernier

\* automatiquement, encore faut-il que l'environnement le supporte !

# Mapping java... (Rappels)



```
package Core;

import Util.Randomizer;
import UI.DieView;
import java.util.*;
import java.awt.Component;

public class Die extends
Observable implements Displayable
{
    private int faceValue = 1;

    public int roll() {

        setValue(Randomizer.getInstance().
            getValue());
        return getValue();
    }

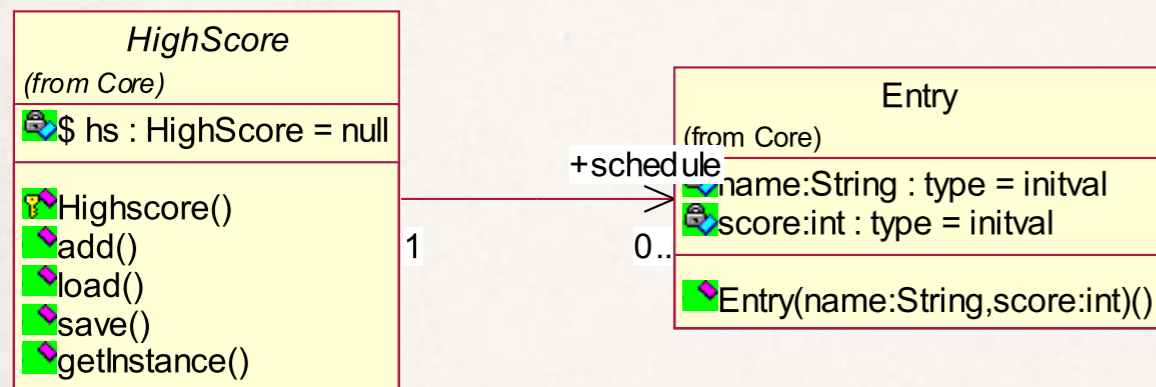
    public java.awt.Component
    display() {
        Component c=new DieView(this);
        this.addObserver((Observer)c);
        return c;
    }

    public void setValue(int value) {
        faceValue=value;
        this.setChanged();
        this.notifyObservers();
    }

    public int getValue() { return
    faceValue;}
}
```



# Mapping Java : Relations (Rappels)



```
package Core;
import java.util.*;
import java.awt.Component;
import UI.HighScoreView;
public abstract class HighScore
extends Observable implements
java.io.Serializable, Displayable {
    protected static HighScore hs = null;
    public Vector schedule=new Vector();
public void add(Entry entry) {
    entries.addElement(entry);
    this.setChanged();
    this.notifyObservers();
}

    public Enumeration elements() {
        return entries.elements();
    }
public abstract void load();
public abstract void save();
public Component display() {
    Component c=new HighScoreView(this);
    this.addObserver((java.util.Observer)c);
    return c;
}
public static HighScore getInstance() {
    if (hs==null) {
        new Error("No Persist Kit declared");
    }
    return hs;}
}
```

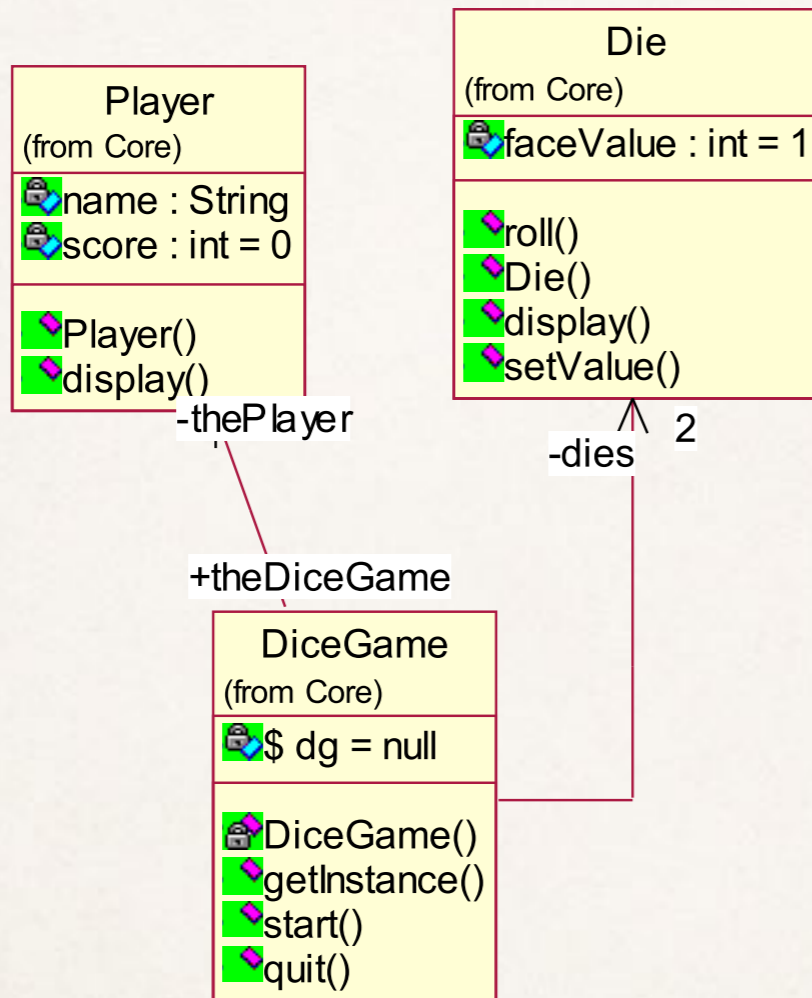
# Codage...

---

- ❖ Utiliser les fonctionnalités de « forward engineering » des outils
- ❖ Puis de « reverse engineering »
- ❖ Mais l'idéal : « round trip engineering »
- ❖ Assurer la cohérence Code / Design / analyse...

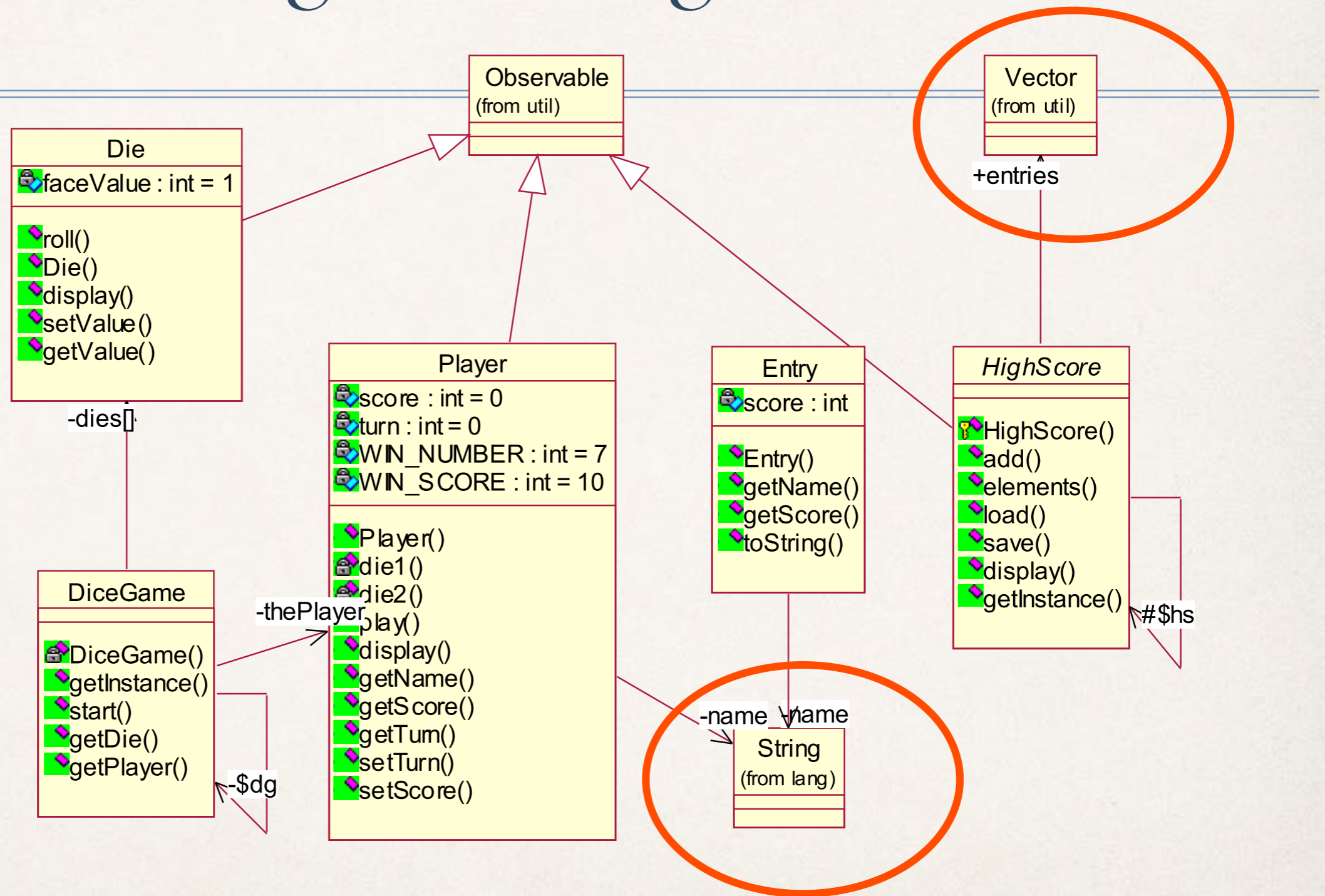
Une séance de TD consacrée au Reverse et à la qualité  
du logiciel

# «Forward engineering»



```
package Core;
public class DiceGame {
    private static int dg = null;
    private Die dies[];
    private Player thePlayer;
    DiceGame() {
    }
    /**
     * @roseuid 37F877B3027B
     */
    private DiceGame() {
    }
    /**
     * @roseuid 3802F61403A0
     */
    public void getInstance() {
    }
    /**
     * @roseuid 37F8781A014D
     */
    public void start() {
    }
    /**
     * @roseuid 38074E7F0158
     */
    public void quit() {
    }
}
```

# «Reverse Engineering...»



# Est-ce que ça marche ? Tester

---

- ❖ Tests unitaires : tester classe par classe, méthode par méthode
  - Diagramme de classes
- ❖ Tests d'intégration :
  - Nous travaillerons sur des diagrammes de classes et de packages
- ❖ Tests du système :
  - Diagramme Use Case + Activités

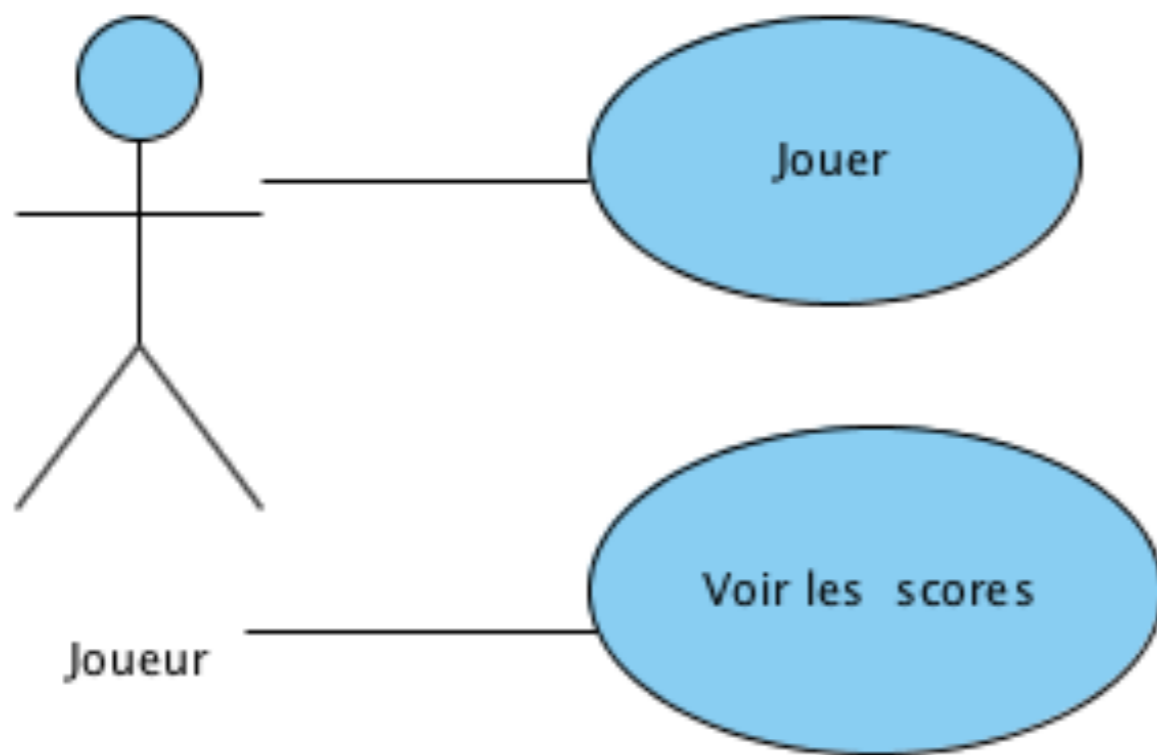
Plusieurs séances consacrées aux tests

d'après Pascal Molli, molli@loria.fr



# Test du système

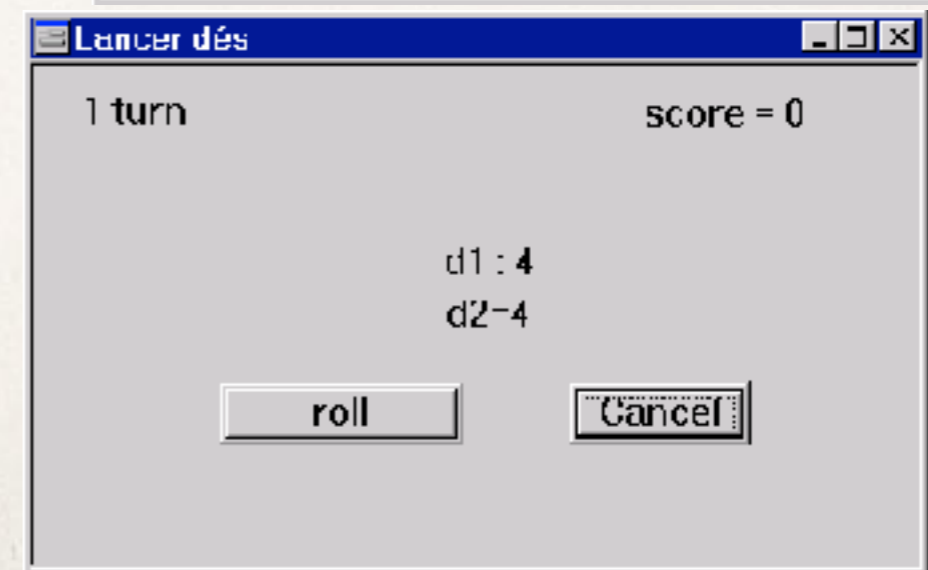
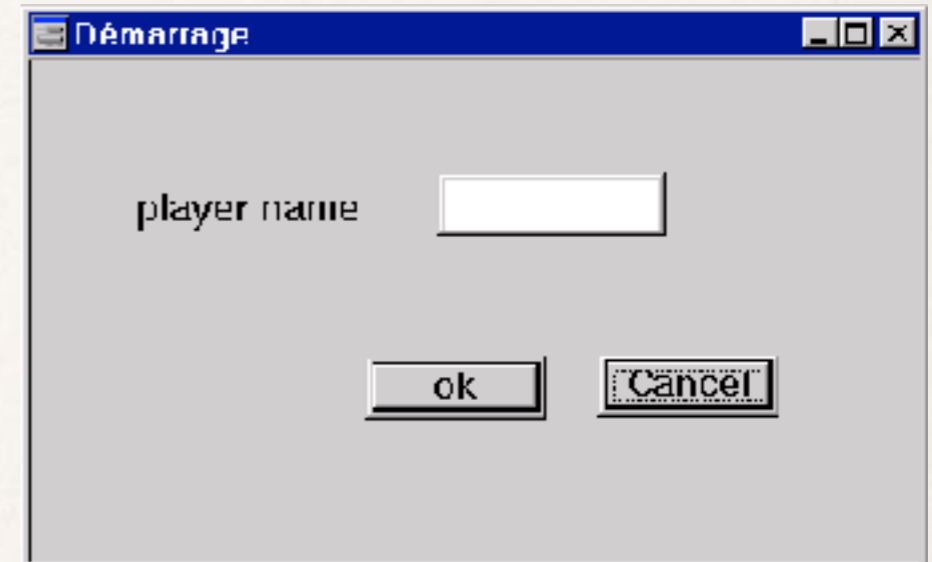
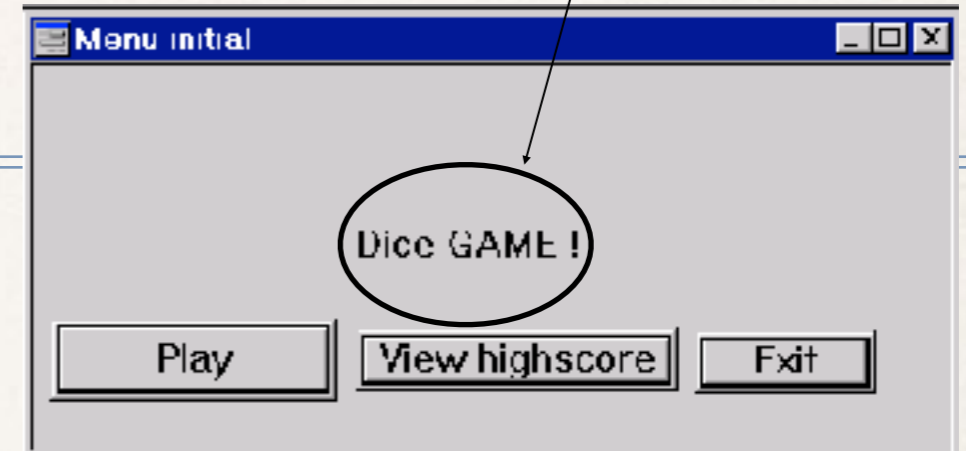
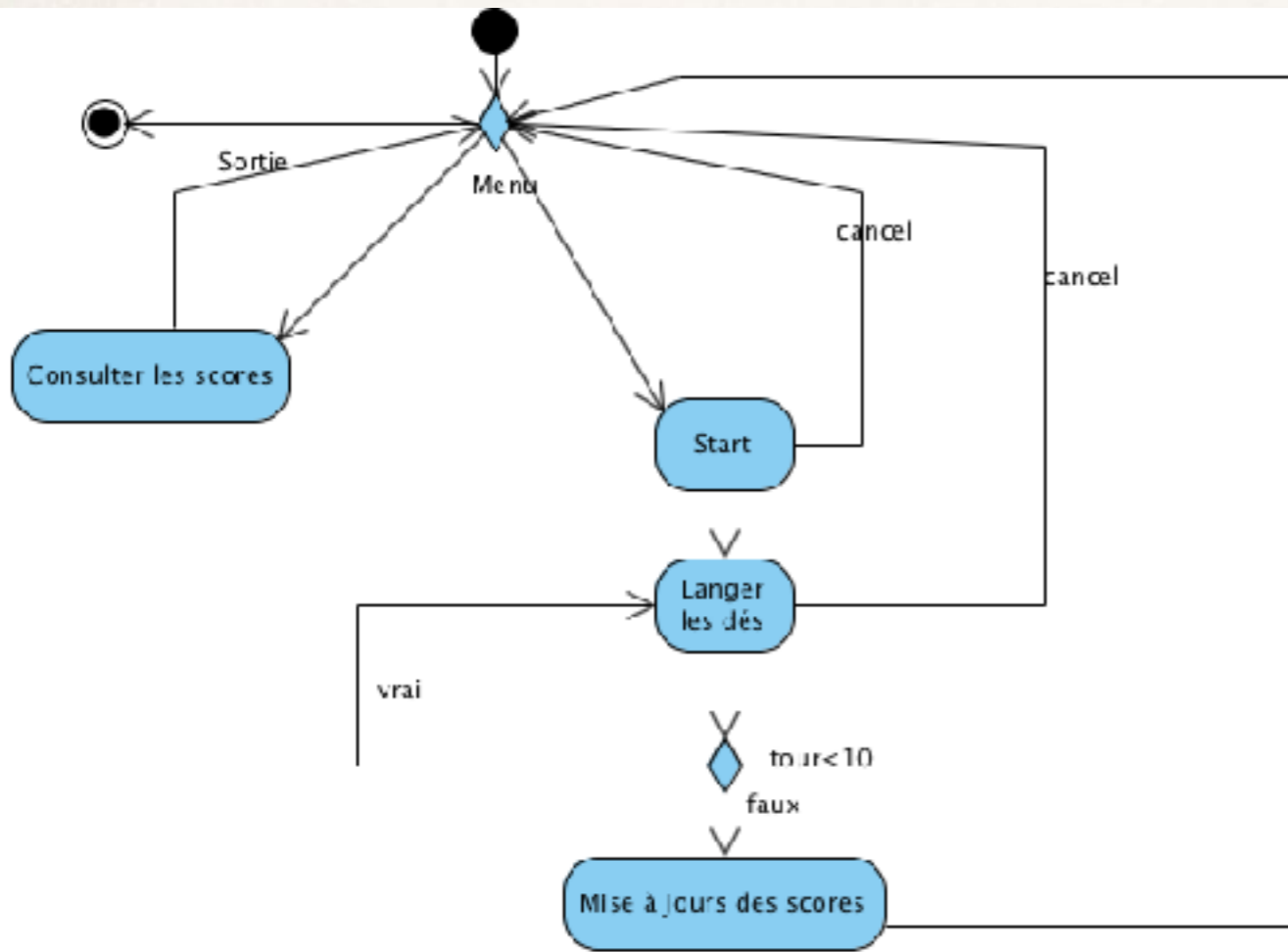
---



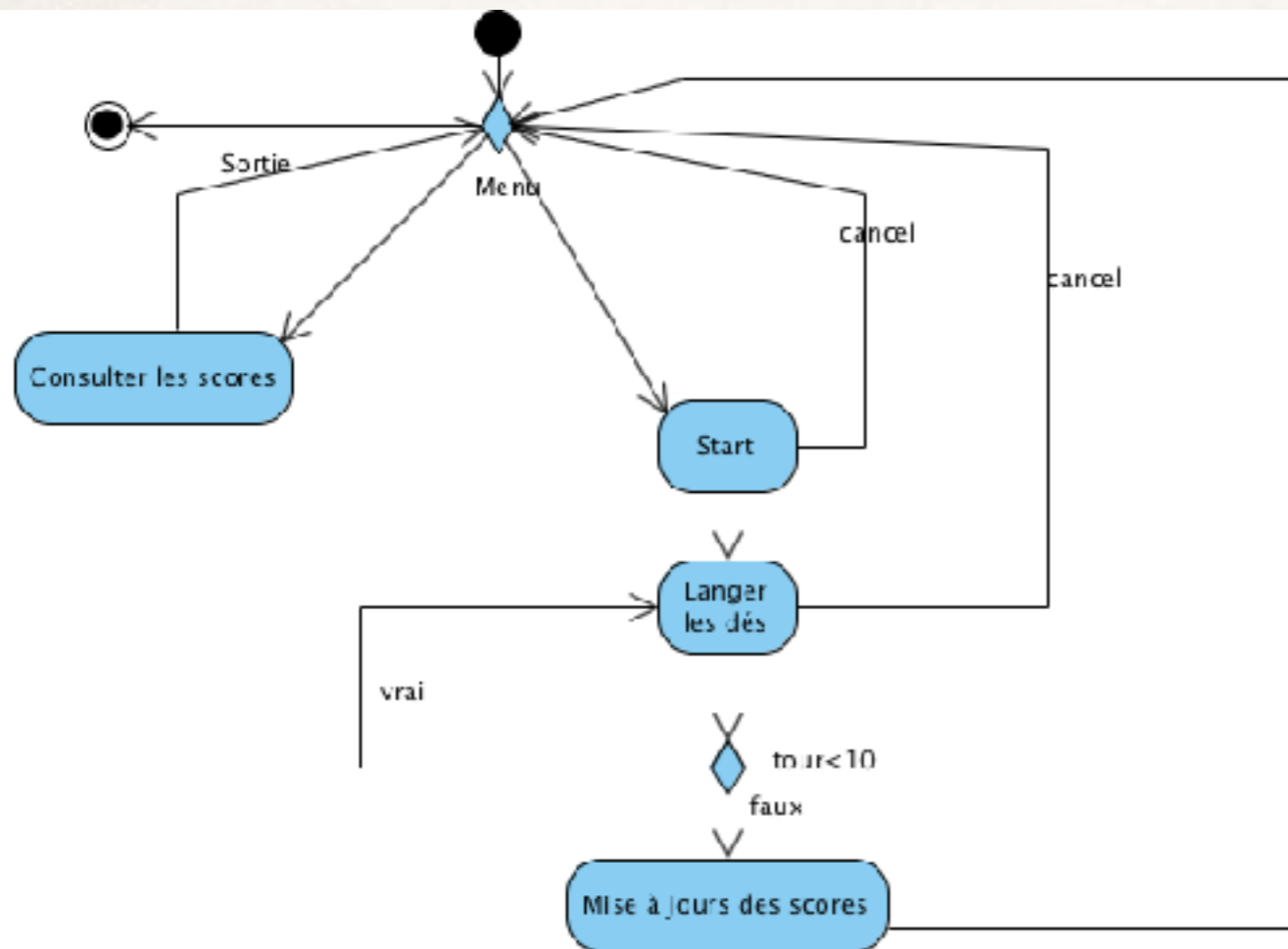
- ❖ Ok, les fonctionnalités sont là ...
- ❖ et sont conformes au descriptif associé au use case !

# Test du système

Je l'ai oublié  
celui là !



# Test du système



■ Tester tous les chemins possibles !

■ Ex:

- 1/ Start
- 2/ roll
- 3/ cancel
- 4/ highscore
- 5/ exit



# Problème rencontré

---

- ❖ Scénario 1 :
  - start, roll\*, highscore, quit : OK
- ❖ Scénario 2:
  - highscore, : ko ! Bug
  - Pb design :
    - DiceGame crée Highscore (start)
    - Si Highscore avant start : bug

# Debriefing de cette application (1)

---

- ❖ Analyse des besoins
  - Use-case + description
  - diagramme d'activités
  - Prototypage UI
- ❖ Analyse
  - Dynamique : Sequence, state
  - Statique : Class Diagram

# Debriefing de cette application (2)

---

- ❖ Conception

- Architecture design (layer)

- diagramme de Packages, diagramme de déploiement

- Classes techniques pour assurer le découpage :

- Pattern d'architecture MVC, ...

# Debriefing de cette application (3)

---

## \* Codage

- Simple conversion du design vers Java (*la partie persistance n'a pas été étudiée*)
- Possibilité de construire pour chaque représentation UML, une traduction vers n'importe quel langage cible.
- Utilisation des outils pour round-trip engineering
- PB au codage : Bien remettre à jour les document analyse / design !!!

# Conclusion sur cette application

---

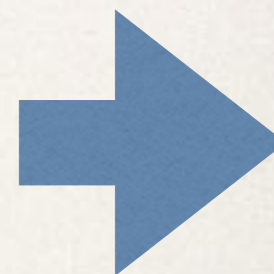
- ❖ **Découpage en phases:**

- Analyse des besoins, analyse, conception, réalisation, test Système
- Mais les « tests » sous-tendent toutes les phases....

- ❖ **Dans chaque phase:**

- Différents points de vue sur le même problème :
  - Vue statique, dynamique, fonctionnelle, architecturale

# Check List



---

*Date*

# Cohérence/couverture

---

- Diagramme Use-cases / Activity
  - Toute activité est assignable à un use-case
  - Tous les use-cases sont réalisés dans les diagrammes d'activités

# Séquence/Class diagram

---

- Tous les objets d'un diagramme de séquence ont un type : Classe du diagramme de classe
- Les «relations» induites par un diagramme de séquence existent ou peuvent être dérivées du diagramme de classe !
- Les messages échangés sont des méthodes du diagrammes de classes !



# Class diagram / séquence

---

- La dynamique des relations apparaît dans au moins 1 diagramme de séquence ou d'activités
- Tout changement d'attributs est représenté dans au moins 1 diag d'activités ou de séquence
- Toute création ou destruction d'objet apparaît dans au moins 1 diag dynamique!

# Class/Package (Design)

---

- Chaque classe est affectée à un package, package lui-même partie intégrante de l'architecture Sinon la classe ne fait pas partie de l'architecture !

# Conclusion générale sur l'application «Dice»

---

- ❖ Différence avec une application juste codée ?
- ✓ Elle est documentée, les choix sont justifiés, les tests sont inclus, ...
- ✓ Elle est «évolutive» :
  - Changement d'IHM, de modèle de persistance
  - Responsabilité unique des objets (à étudier)
  - ....

# Pas mieux

---

[Message de service] Chers 4A, getters, setters et autres abominations Java n'ont RIEN A FOUTRE dans un modèle d'architecture. #Abstraction

# Conclusion

---

- ❖ La suite de ce cours doit vous aider à produire des applications de qualité en améliorant vos «développements» de l'analyse à la mise en oeuvre, tel est **notre** objectif.



# Bibliographie

---

- ❖ Unified Modeling Language; Pascal Molli; Université Nancy1, Loria  
[molli@loria.fr](mailto:molli@loria.fr), [www.loria.fr/~molli](http://www.loria.fr/~molli)
- ❖ Craig Larman, UML2 et les Design Patterns