

Genie Logiciel

Jacques Lonchamp

SEPTIEME PARTIE

Une étude de cas UML

1. Le problème

C'est un problème très simple, afin qu'il puisse être traité complètement. Il est tiré de l'ouvrage 'Applying UML and Patterns' de Craig Larman (et des transparents de Pascal Molli, UHP Nancy).

Il s'agit d'un jeu de dés. Le joueur lance 10 fois 2 dés. Si le total des 2 dés fait 7, il marque 10 points à son score. En fin de partie, son score est inscrit dans le tableau des 'high scores'.

2. L'analyse des besoins (définition des fonctionnalités attendues)

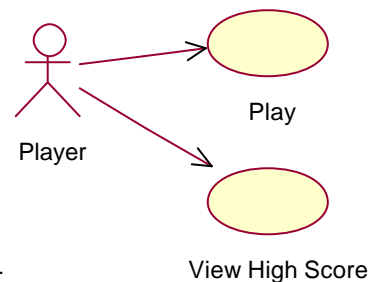
2.1. Un premier cas d'utilisation

Cas Play

Description : le joueur lance 10 fois les dés ;
à chaque fois que le total fait 7, ajout de 10pts.

Cas View High Score

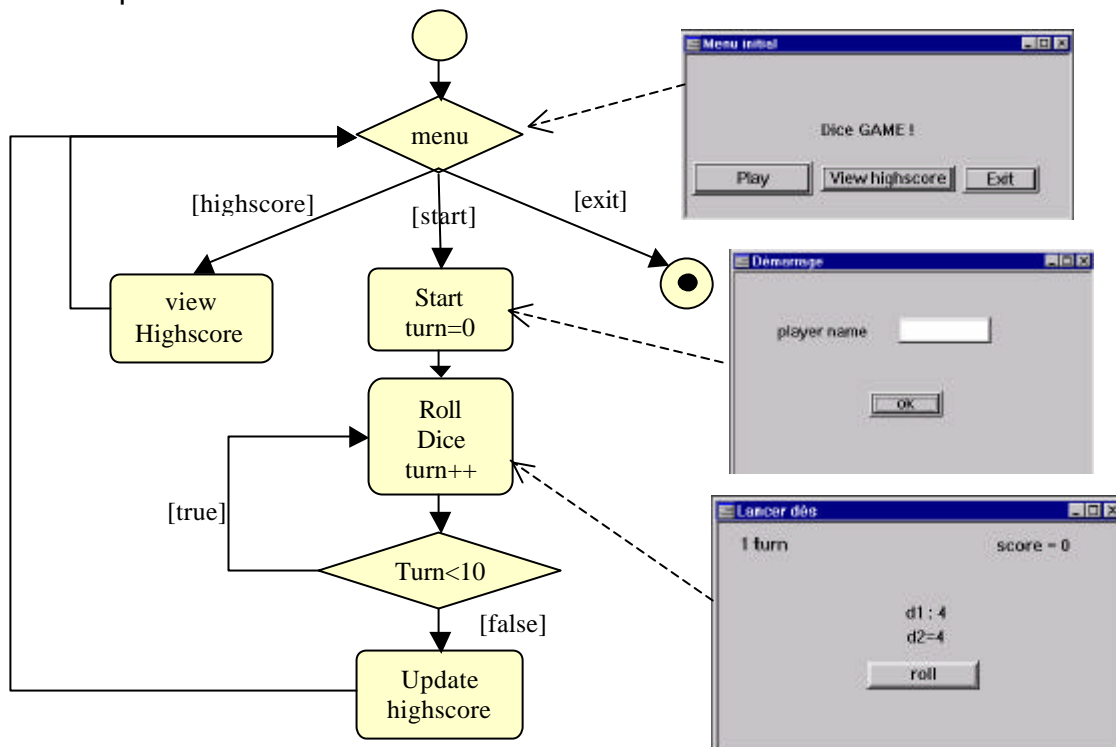
Description : le joueur consulte en lecture seulement les high scores.



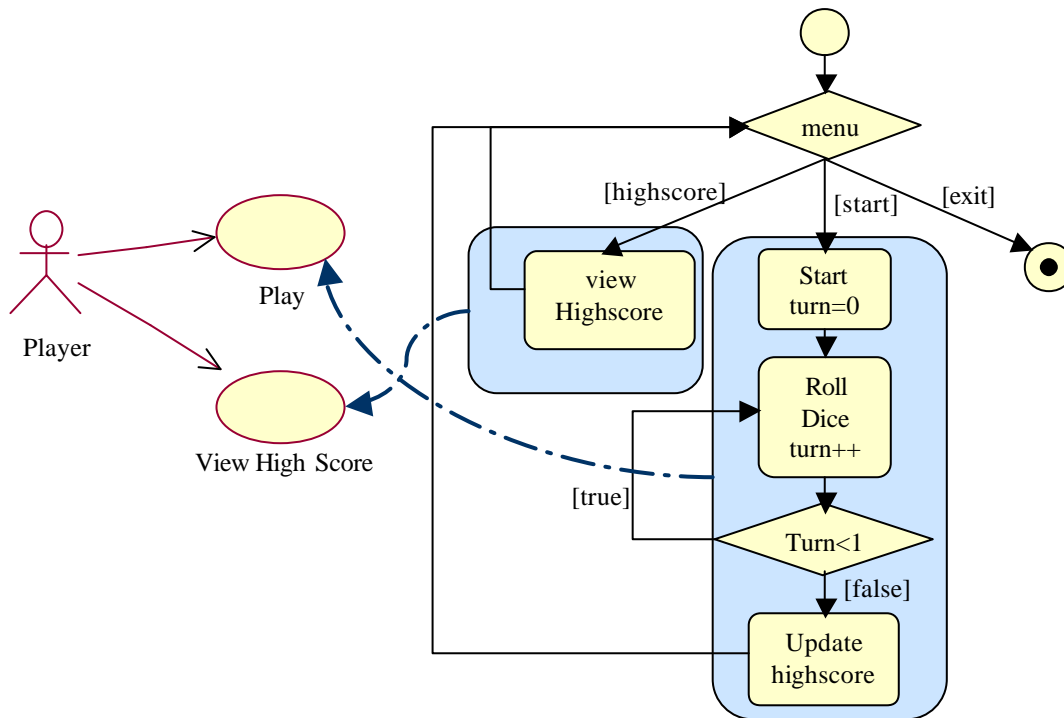
2.2. Un premier diagramme d'activités

Sa construction permet de décrire l'organisation générale des traitements et permet de dialoguer avec les 'clients'.

On peut compléter ce diagramme par des maquettes d'écran associées aux différentes opérations.



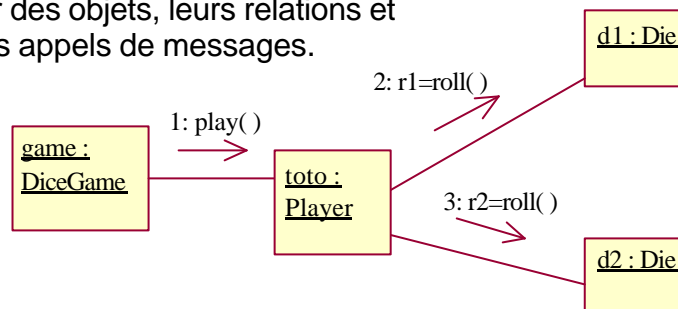
Il doit exister une certaine cohérence entre ces diagrammes. On peut vérifier que tous les cas se retrouvent quelque part dans le diagramme d'activités.



3. L'analyse (définition des classes du domaine et de leur dynamique)

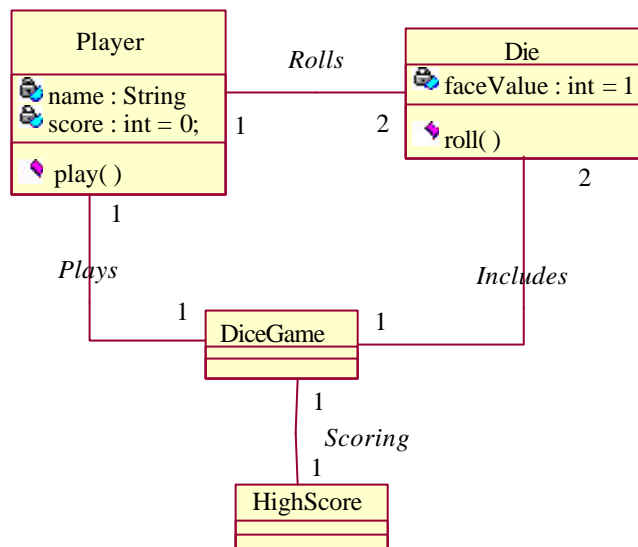
3.1. Un premier diagramme de collaboration

Il permet de visualiser des objets, leurs relations et l'ordonnancement des appels de messages.

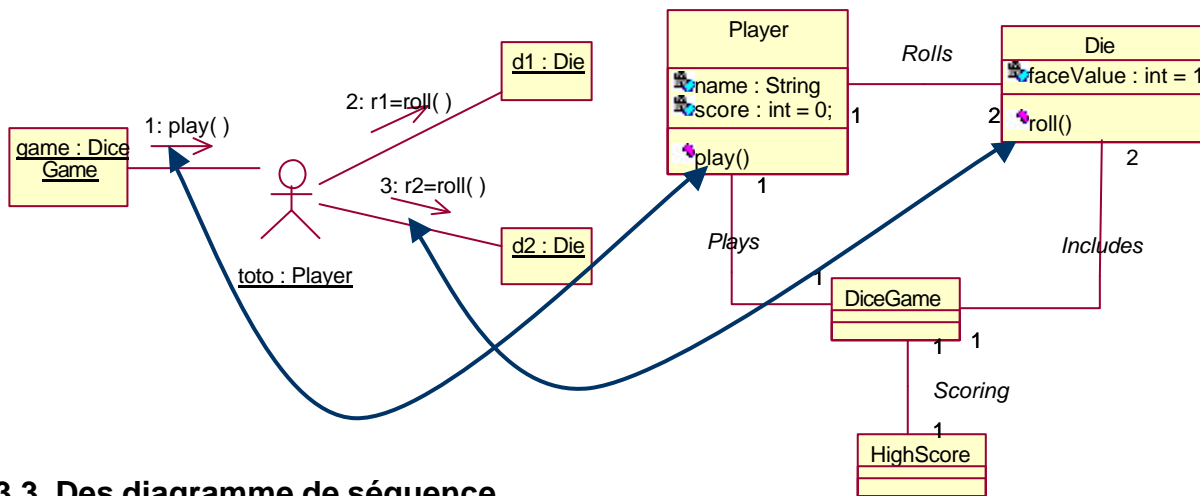


3.2. Un premier diagramme de classes

On complète les classes et on représente leurs associations statiques et dynamiques ; on définit les attributs des classes et les cardinalités des associations.

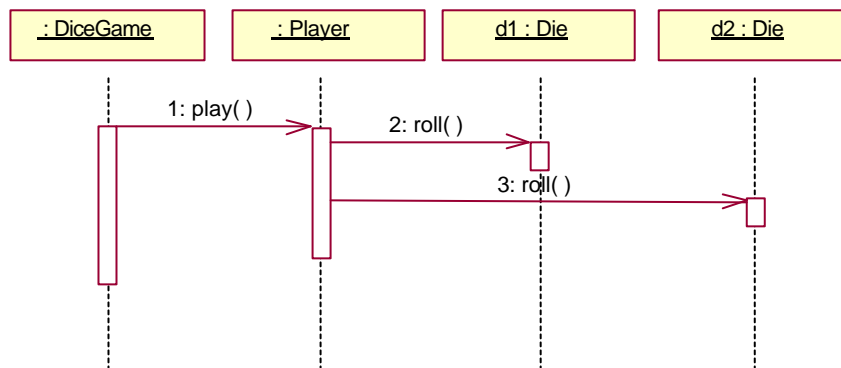


Ici aussi il faut vérifier la cohérence des diagrammes.

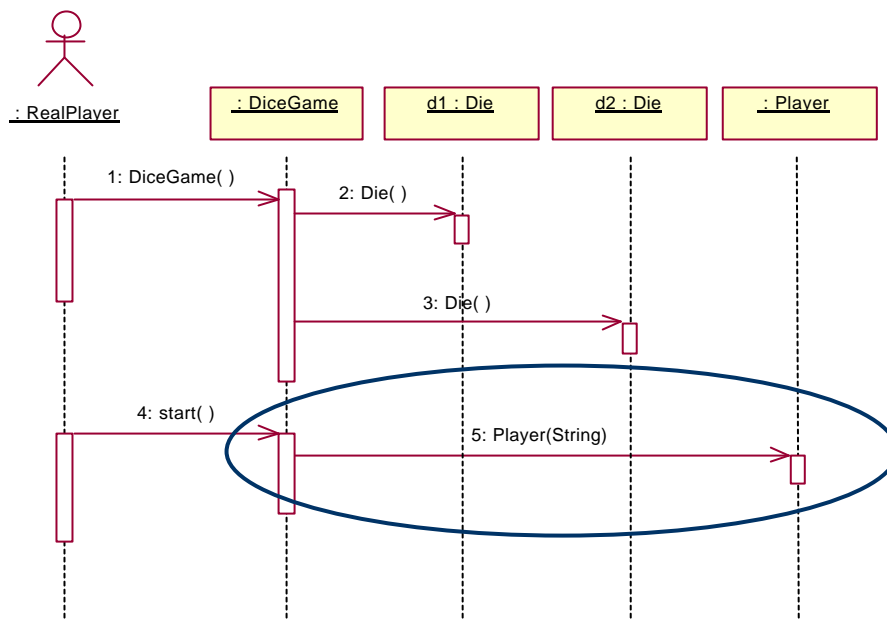


3.3. Des diagramme de séquence

Ils modélisent la dynamique (~ comme diagrammes de collaboration) en se focalisant sur l'enchaînement des messages.

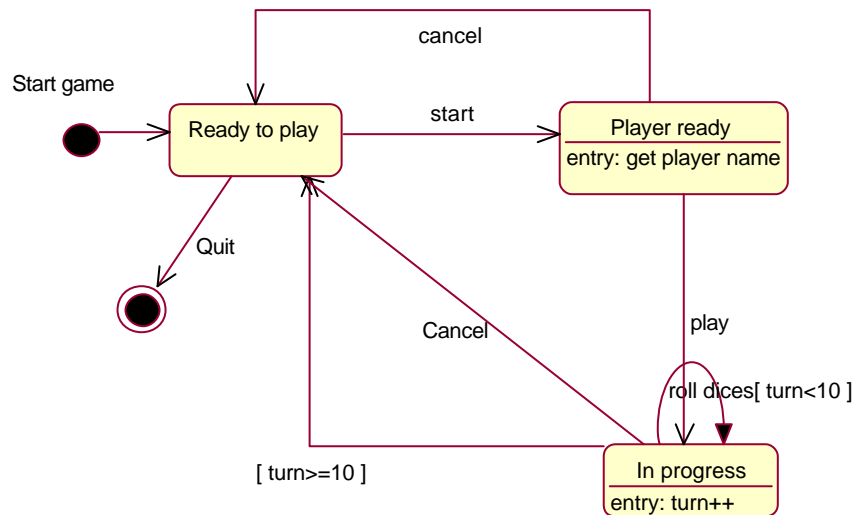


On peut aller plus loin en modélisant la création des objets en début de partie. A noter que le joueur (:Player) n'est créé qu'au démarrage de la partie. C'est un choix qui découle du diagramme d'activité. On aurait tout aussi bien pu créer le joueur au moment où l'on crée le jeu. Le petit avantage est que l'on peut changer de nom entre 2 parties ...

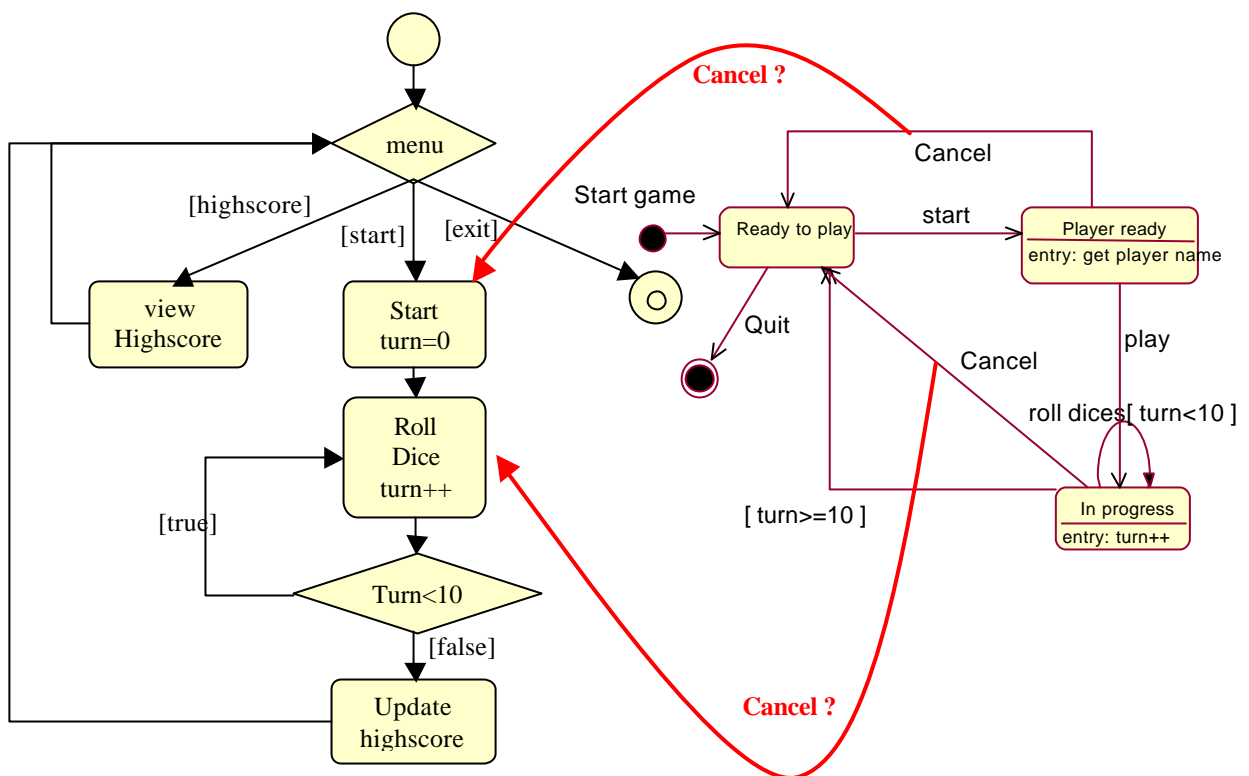


3.4. Un diagramme d'états

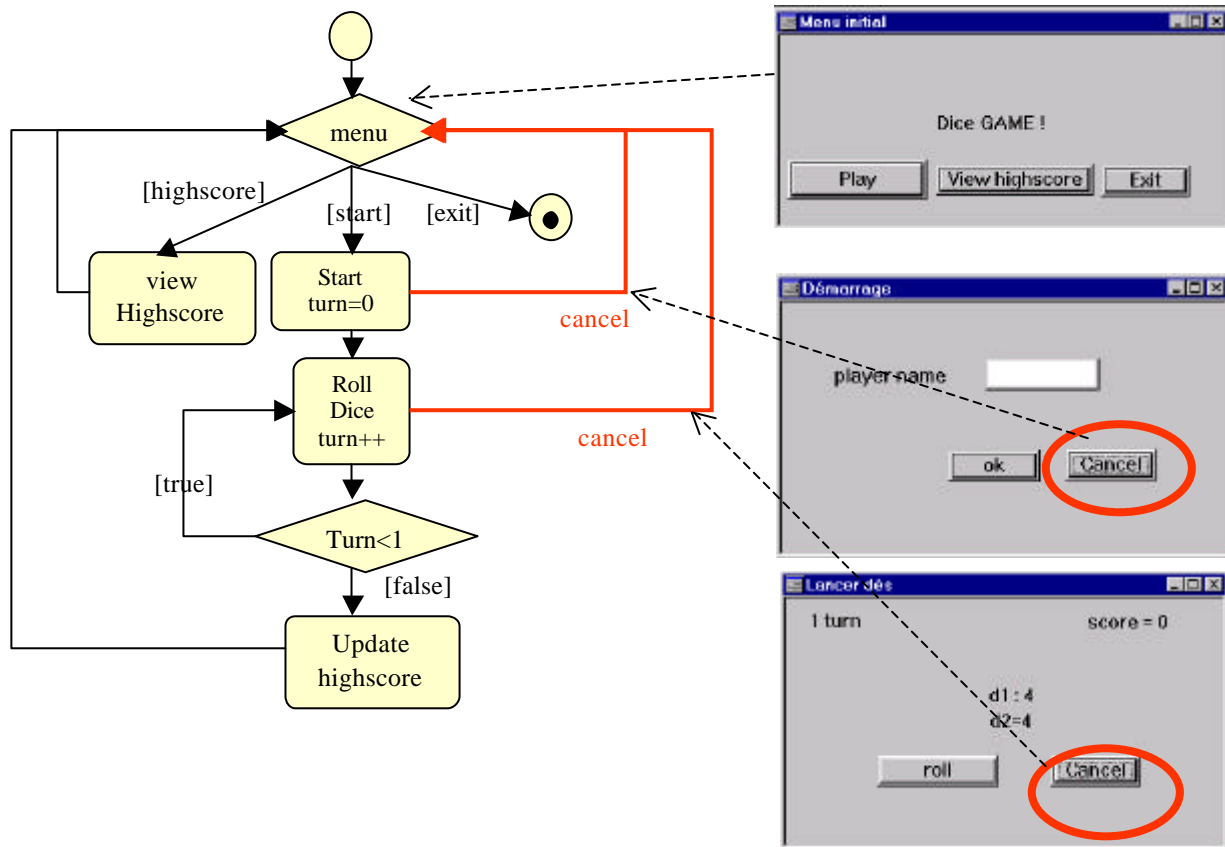
Il a pour objet de déterminer les états et transitions d'état d'un objet. Ici le diagramme d'états d'une partie (diceGame).



On détecte certaines incohérences.

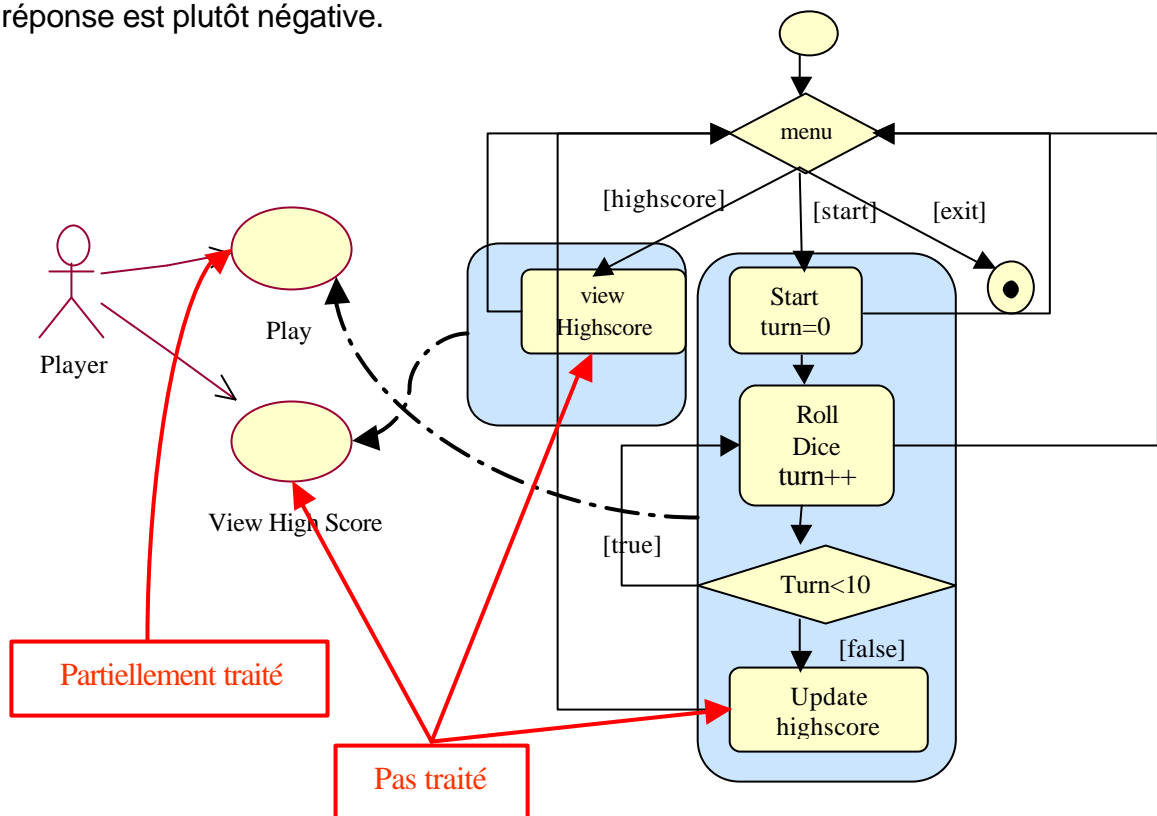


Ces incohérences mises en évidence par le diagramme d'états conduisent à modifier les diagrammes précédents ainsi que les maquettes d'écrans envisagées lors de l'analyse des besoins.

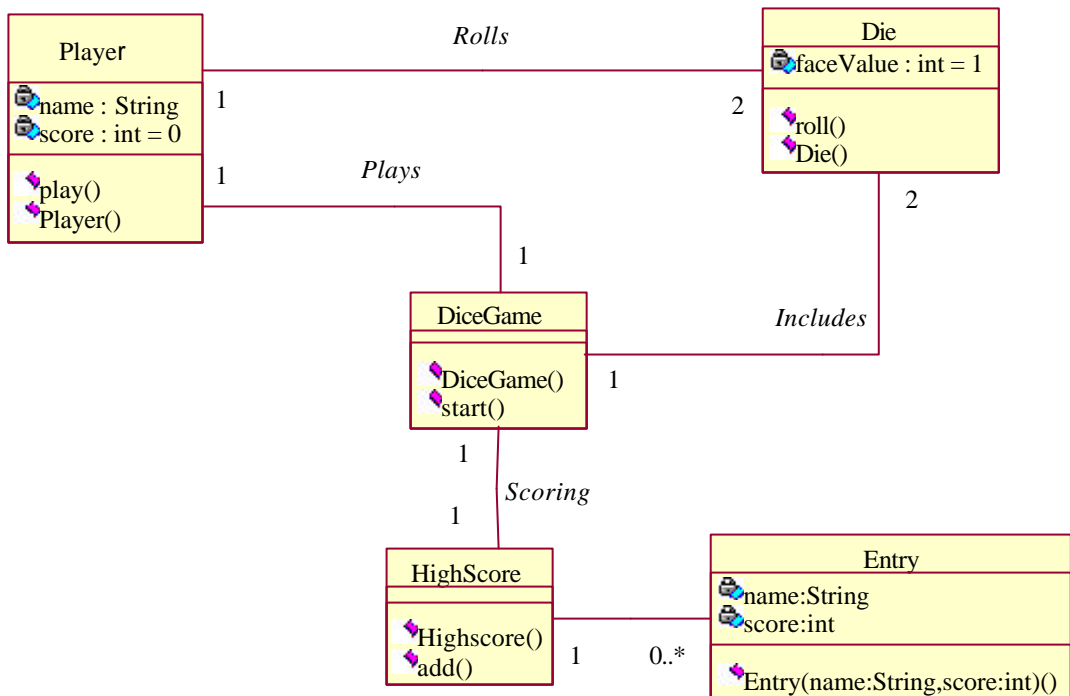
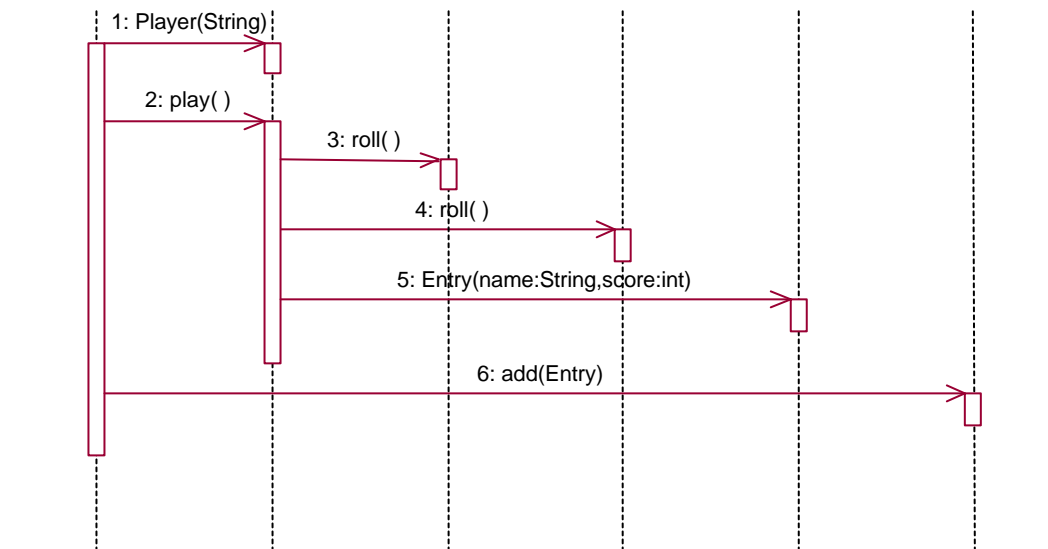
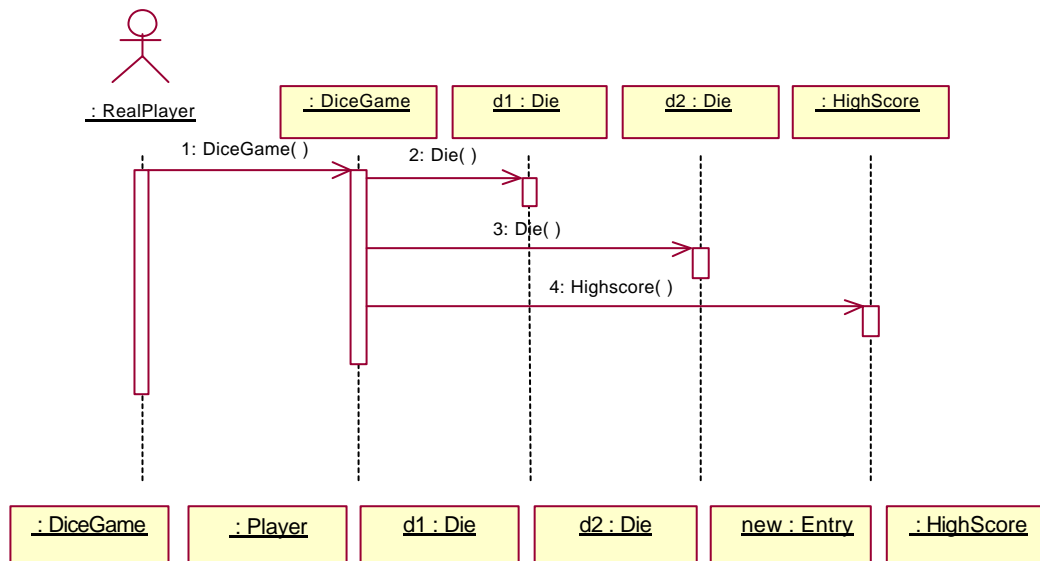


3.5. Le problème de l'arrêt de l'analyse

Il faut se demander si les cas d'utilisation sont couverts correctement par l'analyse. La réponse est plutôt négative.



Il faut donc reprendre les schémas pour gérer le highscore (création et mise à jour).

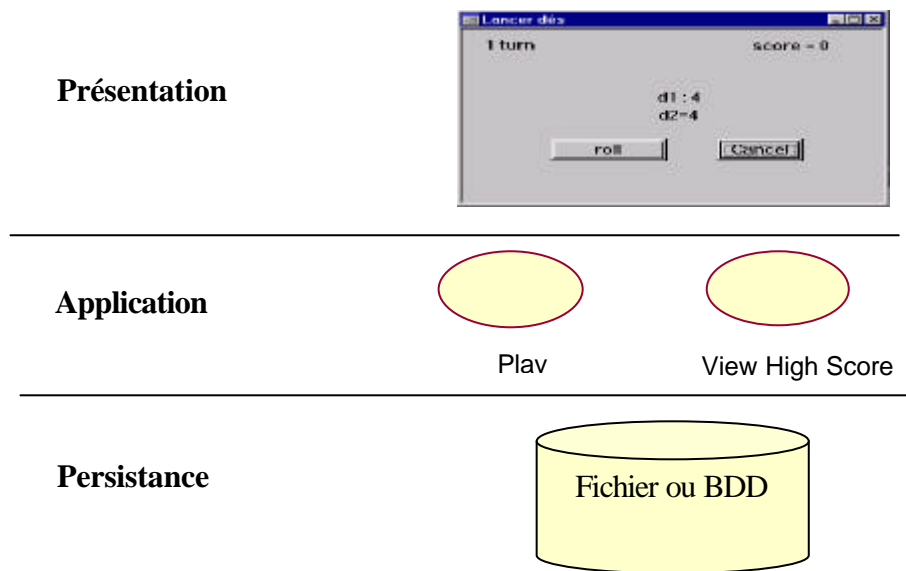


L'analyse des cas est à peu près complète. On pourrait encore raffiner un peu la dynamique du cancel et les diagrammes de séquence du jeu. Passons à la phase de conception de la solution.

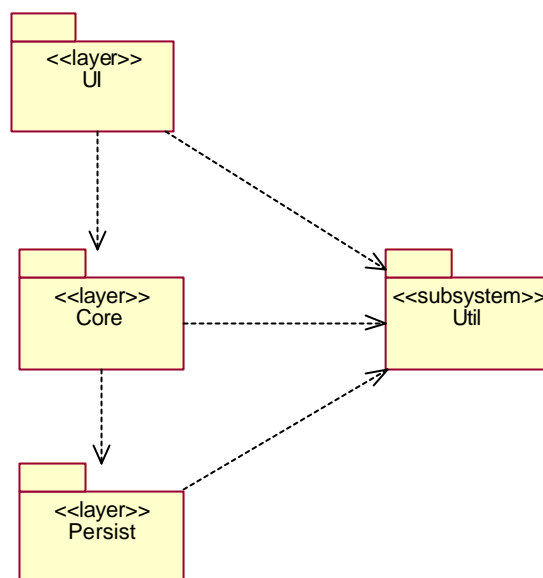
4. La conception

4.1 L'architecture générale

On fait le choix d'une architecture en couches (pattern architectural Layer vu en cours) comportant très classiquement 3 couches.



Il correspond à cette architecture des paquetages et des dépendances entre paquetages. Le paquetage Util regroupe les services qui ne se rattachent pas clairement aux niveaux.

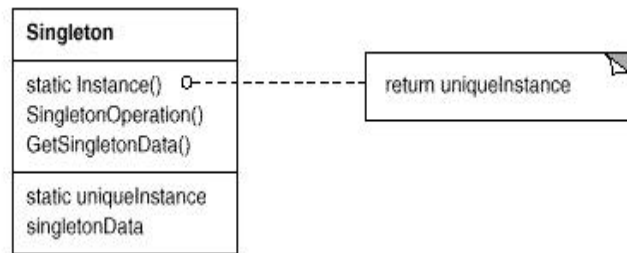


4.2. Conception du niveau applicatif 'Core'

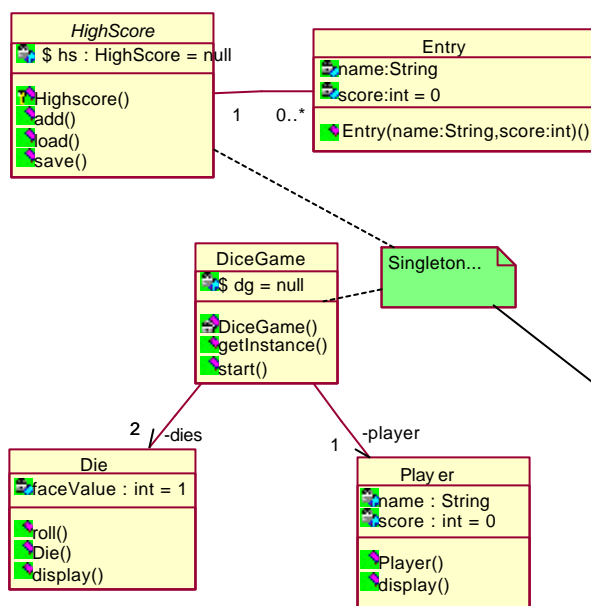
On cherche à appliquer différents 'design patterns' permettant de mieux structurer les classes.

a) le pattern singleton

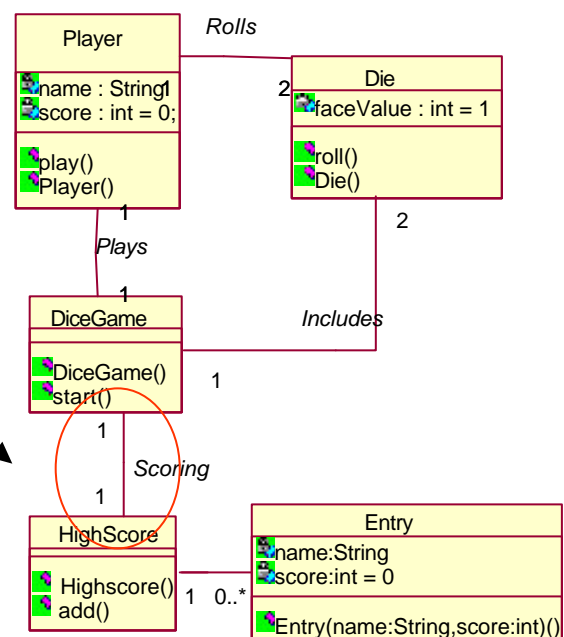
Assure qu'une classe à une instance unique qu'elle garde dans une variable de classe (static uniqueInstance) et donne un point d'accès depuis la classe à cette instance (méthode de classe static Instance() ou getInstance()).



Conception



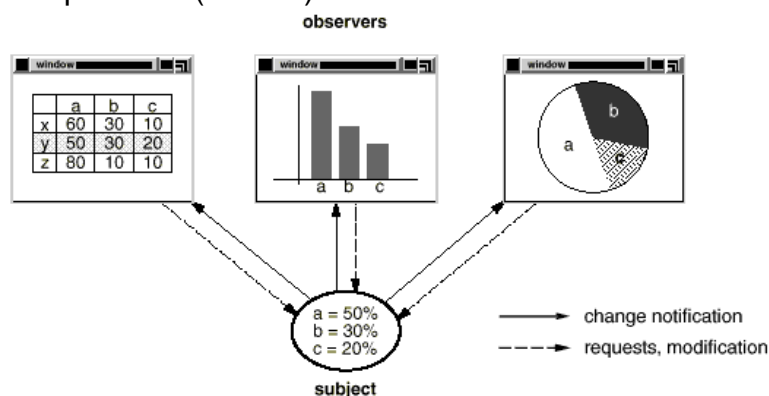
Analyse

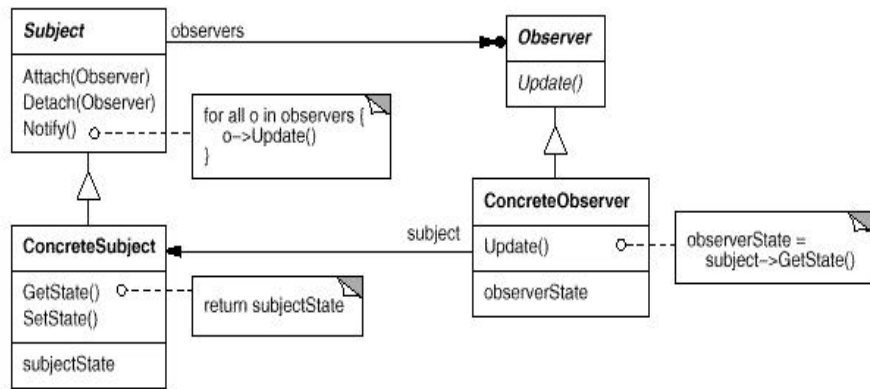


Sont également ajoutées des méthodes pour sauvegarder les high score (load, save) et pour afficher les dés et joueurs (display).

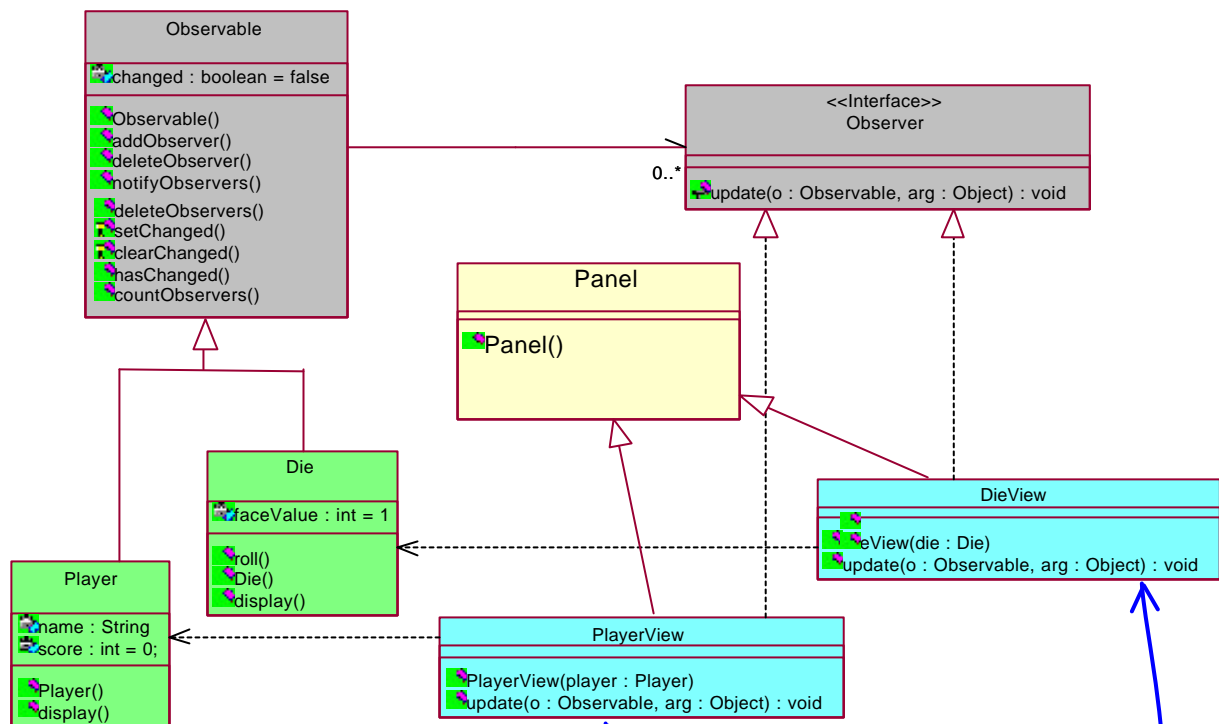
b) le pattern observer

Un objet (le sujet) est lié à plusieurs objets (les observateurs ou observers) pas nécessairement connus à la création du programme. Toute modification est automatiquement répercutée (notifiée) à tous les observateurs.





Ici, les dés et les joueurs sont les sujets (classe Observable) et les vues sur les dés et sur les joueurs sont les observateurs (interface Observer).

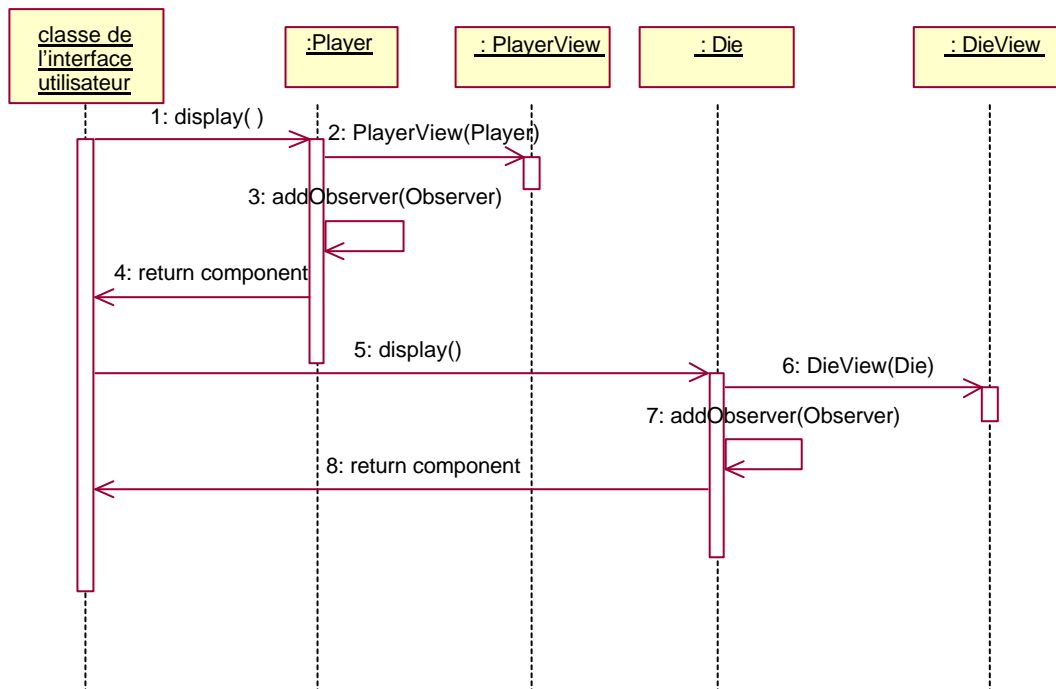


Les vues sont les éléments d'affichage des objets. Ils doivent évoluer dès que l'objet observé change. D'où le recours au pattern Observer avec notifyObservers et update().

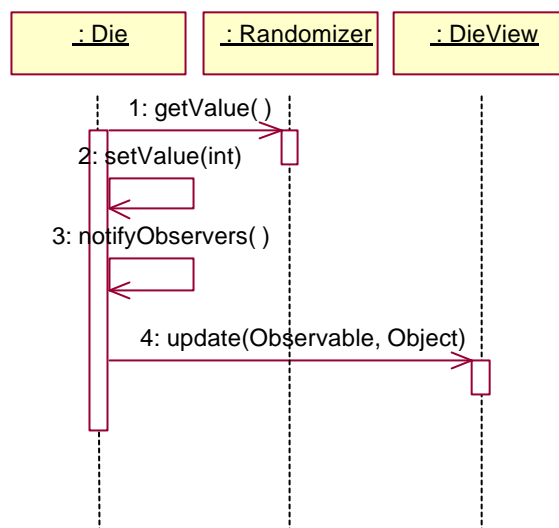
Les vues sont des panneaux (Panel) au sens de l'interface utilisateur (ici awt de Java), c'est à dire une zone d'une fenêtre.



La mise en place de cette structure nécessite les échanges suivants :



et la propagation d'un événement (ici une nouvelle valeur pour un dé) nécessite :



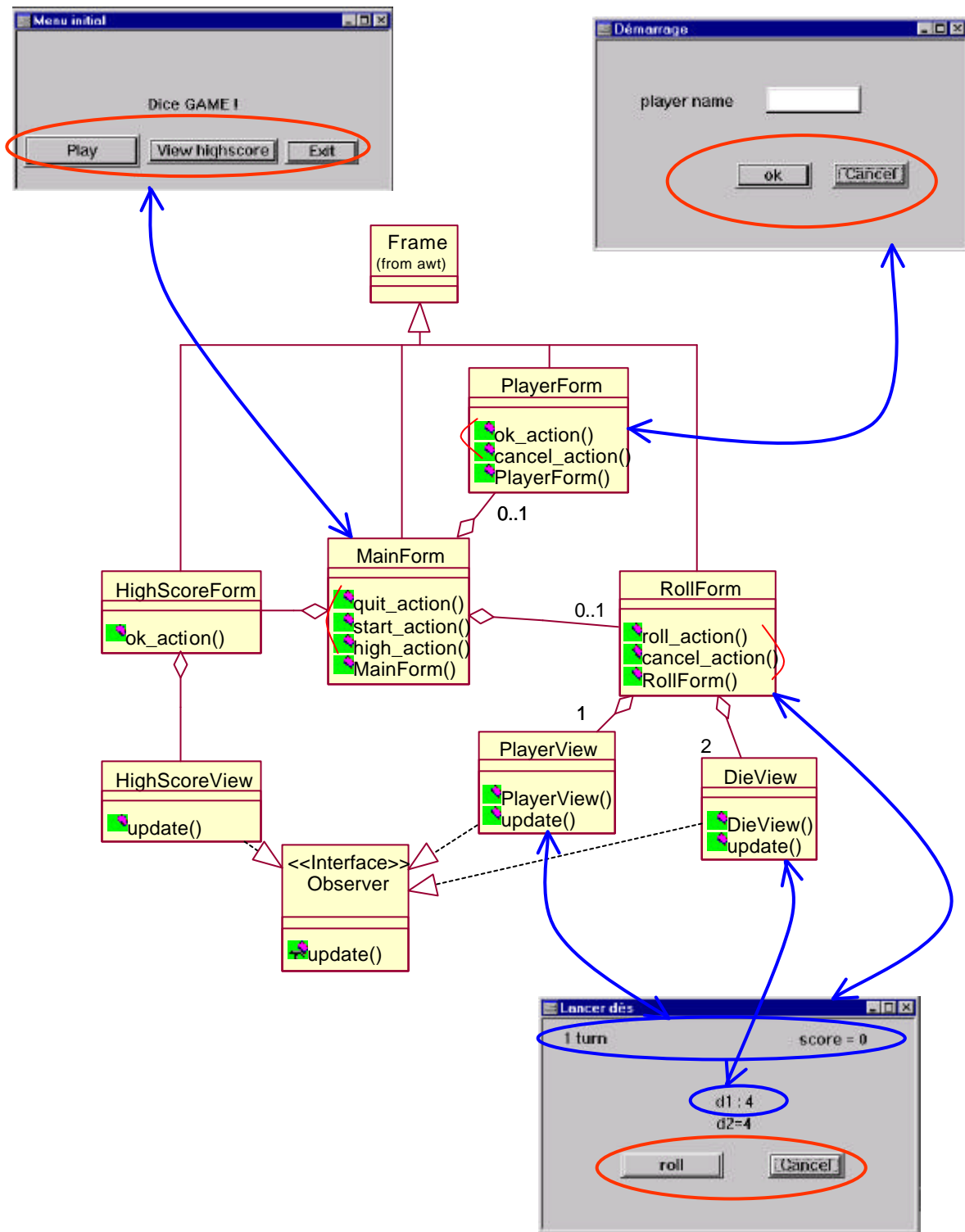
où Randomizer est une classe qui rend une valeur au hasard (pour le tirage du dé entre 1 et 6).

Il faut noter que les interfaces utilisateurs (comme awt en Java) se chargent de propager les *événements* sur les éléments d'interface (bouton appuyé, valeur saisie, clic ou déplacement de souris, fermeture de fenêtre, ...) vers les classes applicatives.

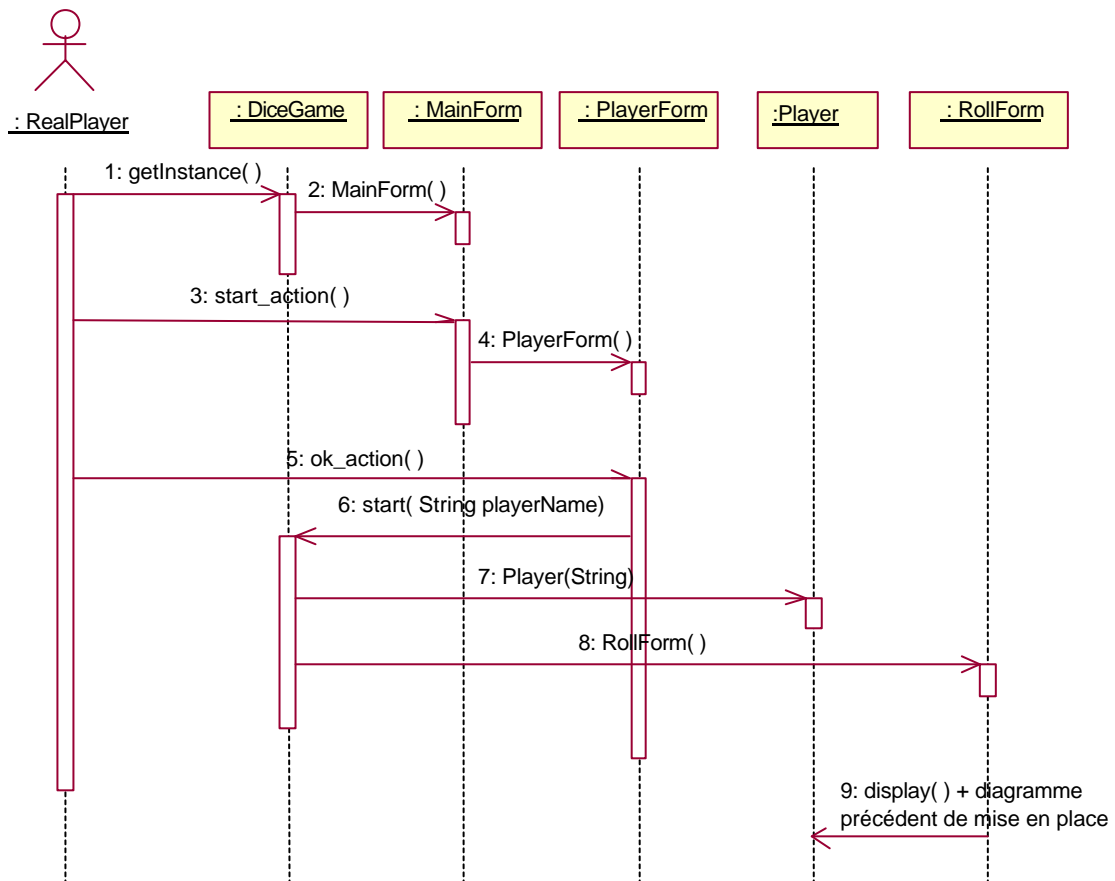
Le pattern Observer *fait le travail opposé* : il notifie les changements d'état des classes applicatives aux éléments de l'interface utilisateur.

4.3. Conception du niveau interface utilisateur 'UI'

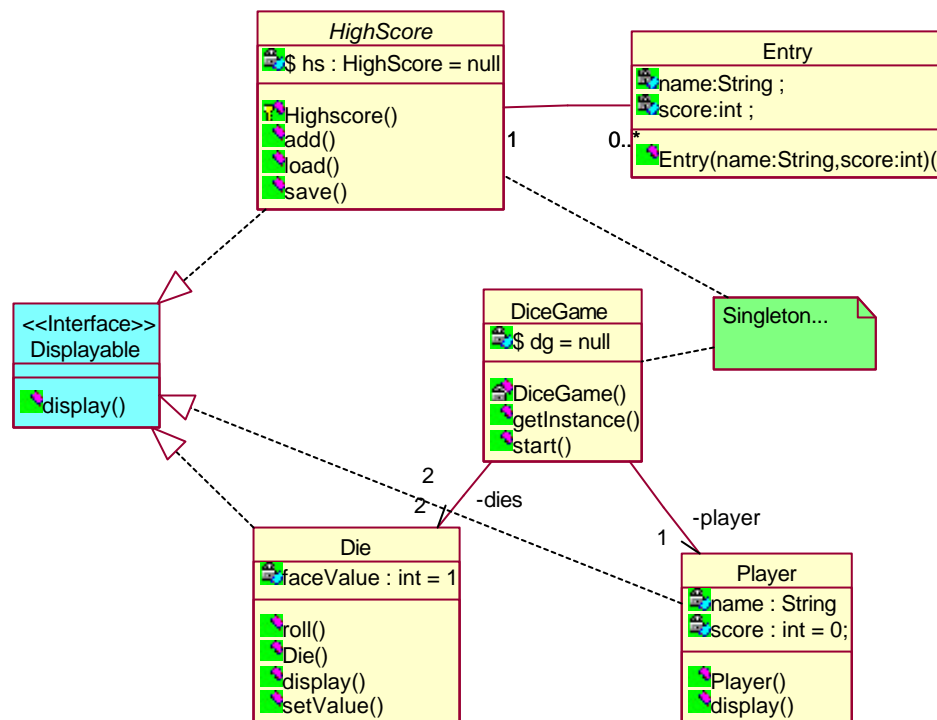
Il faut définir les fenêtres graphiques ('forms') contenant éventuellement les panneaux vues. Toutes les forms héritent de la classe awt Frame.



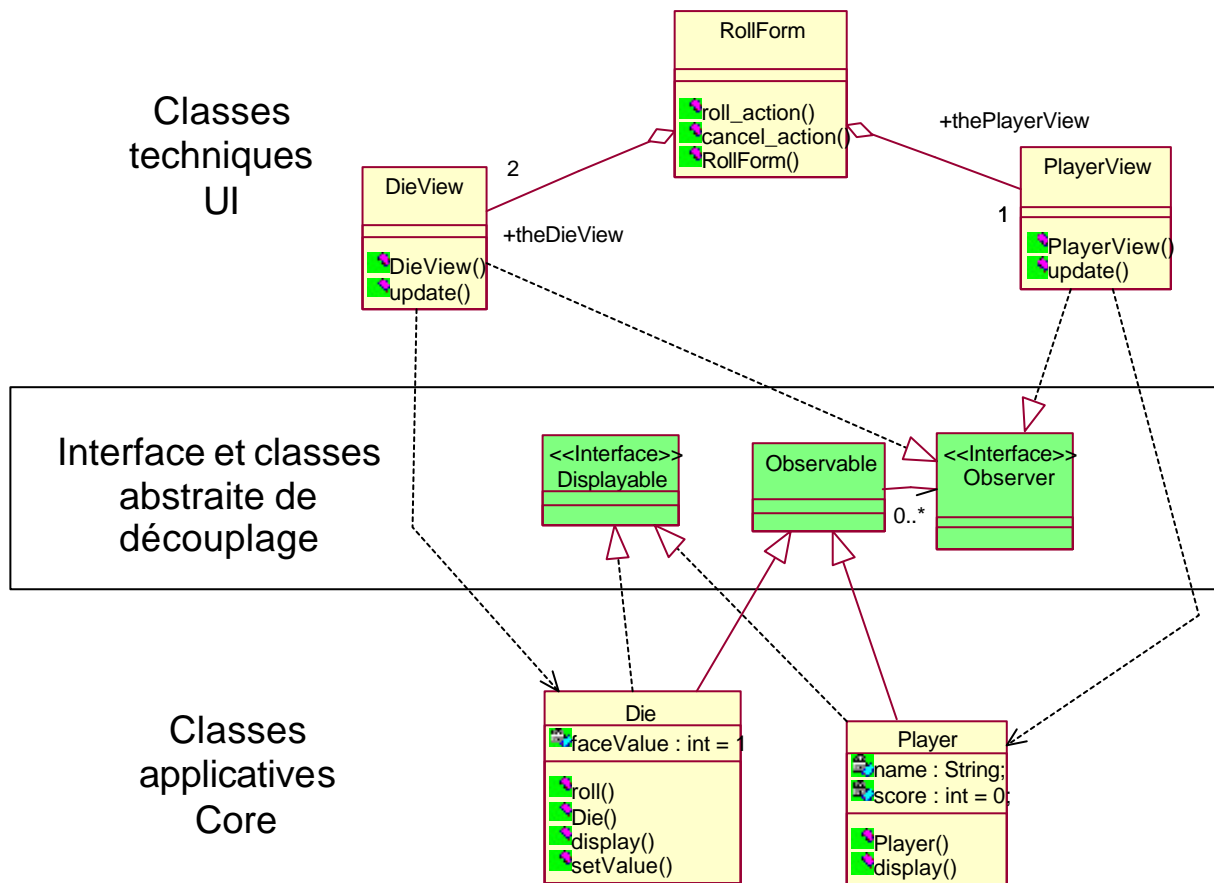
Mise en place de l'interface utilisateur et démarrage du jeu :



Amélioration du découplage UI / Core : introduction d'un Interface Displayable.

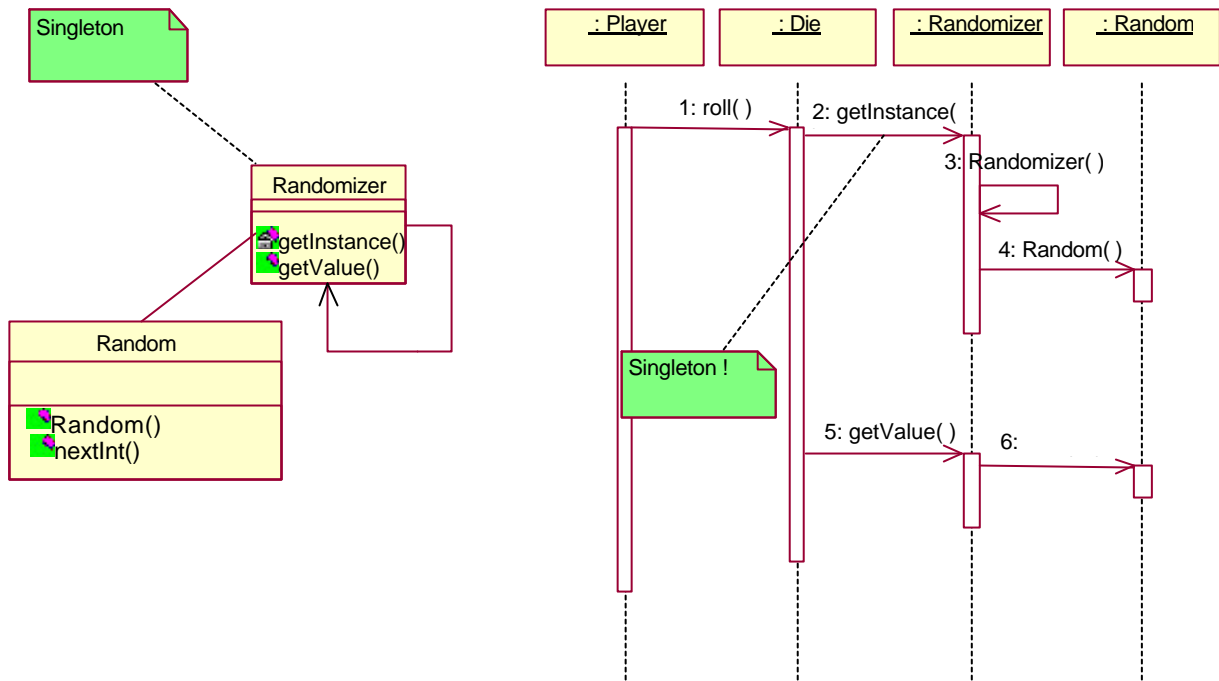


D'où l'architecture en couche avec des interfaces et classes abstraites de découplage :



4.4 Le paquetage Util

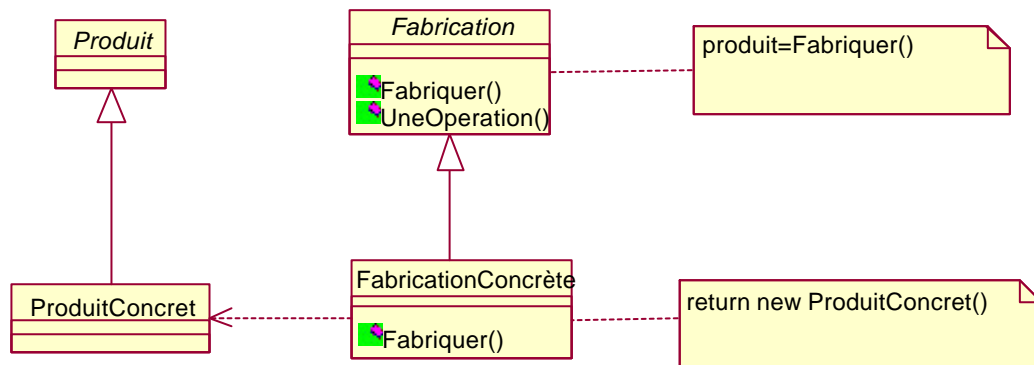
Il comprend la classe `Randomizer` qui utilise la classe Java `Random`. Encore un singleton !



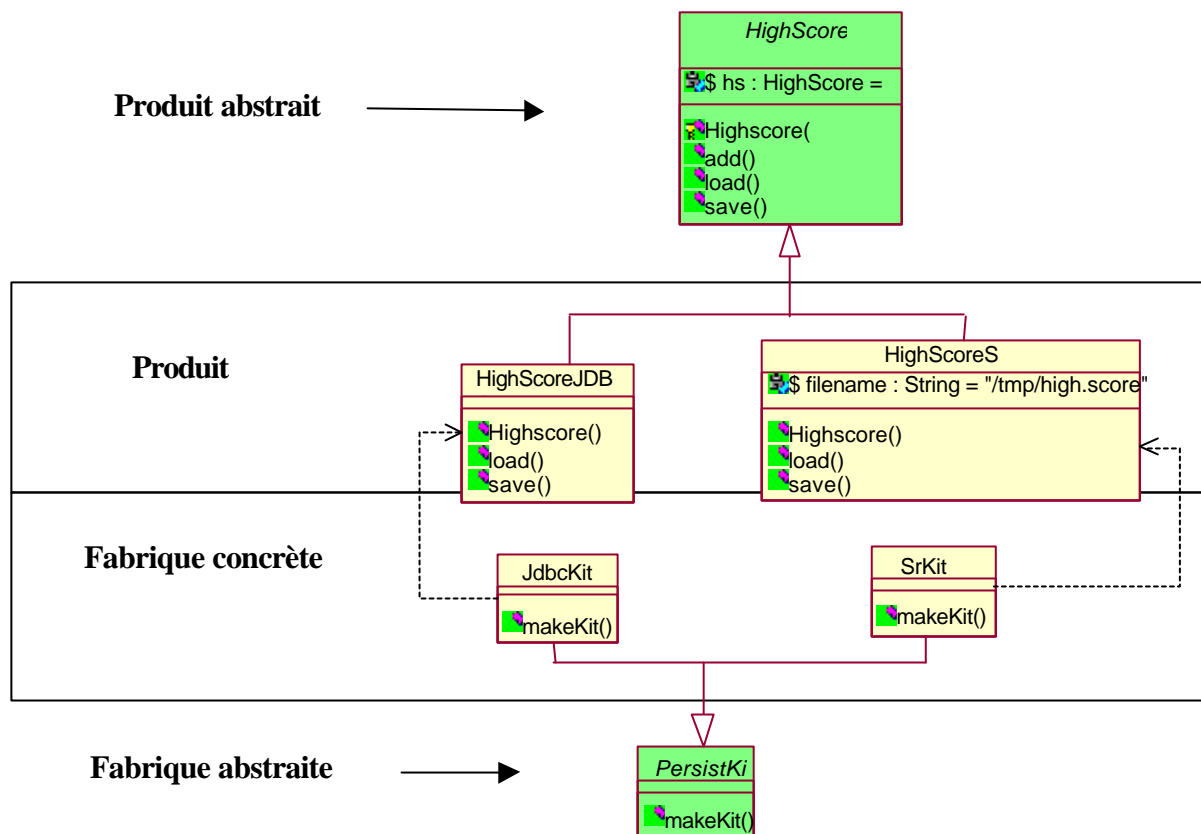
4.5 Le niveau persistance Persist

Il contient les classes techniques de persistance. L'objectif est d'assurer l'indépendance Core/Persist afin de pouvoir utiliser plusieurs types de persistance.

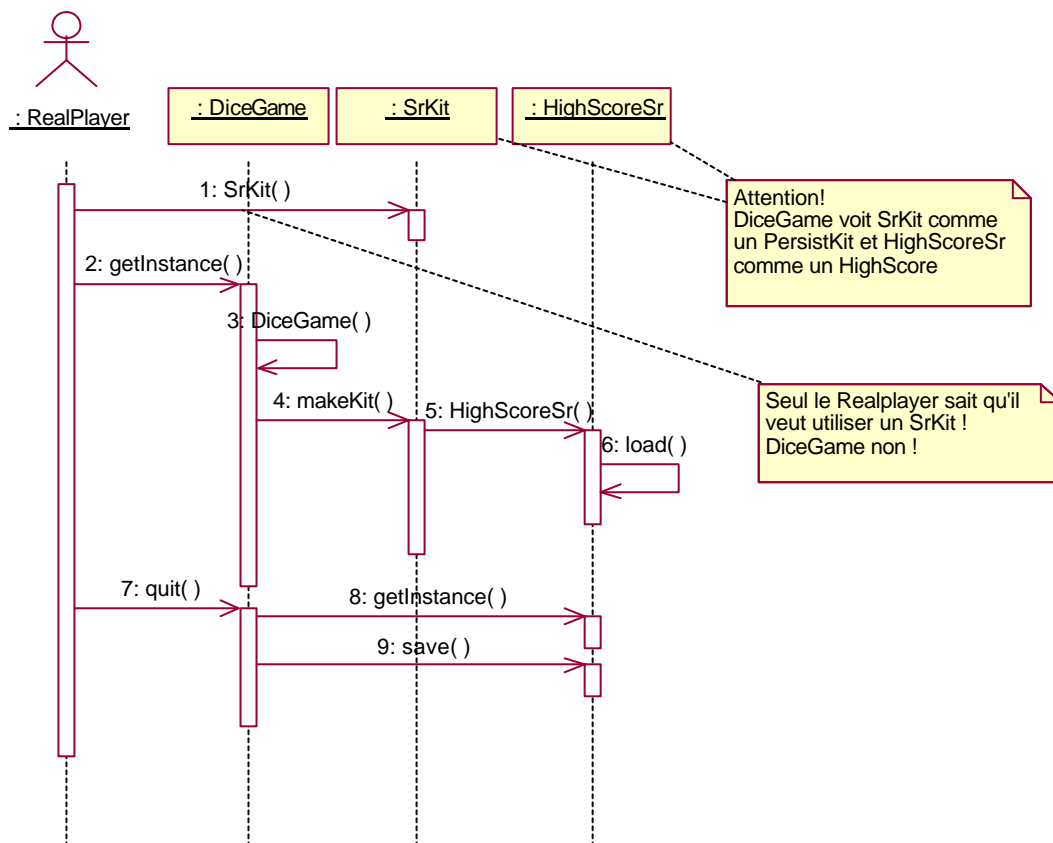
Par exemple, la sérialisation (persistance d'objets liés dans un fichier) et l'utilisation d'une base de données relationnelle (via JDBC). Pour cela, on fait appel au pattern Factory. Ce pattern sert quand une classe peut créer des objets (ProduitConcret) de différentes classes. Une interface unique (Fabrication) est implantée par différentes classes concrètes pour chaque type d'objet à créer (FabricationConcrète). La classe qui veut créer les objets peut ne travailler qu'avec les interfaces.



Ici, les fabriques concrètes créent un highscore sérialisable ou un highscore jdbc.



La dynamique de la couche de persistance est donc la suivante :



Dans le cas sérialisable, la persistance se propage automatiquement de l'objet racine (highscore) à tous les objets dépendants (entries).

```

class HighScoreSr extends HighScore implements Serializable {
    ...
    public void save() throws Exception {
        FileOutputStream ostream = new FileOutputStream(filename);
        ObjectOutputStream p = new ObjectOutputStream(ostream);

        p.writeObject(this); // ecrit le highscore et toutes les entrees
        p.flush();
        ostream.close();      // ferme le fichier de serialisation
    }

    public void load() throws Exception {
        FileInputStream istream = new FileInputStream(filename);
        ObjectInputStream q = new ObjectInputStream(istream);

        HighScoreSr hsr = (HighScoreSr)q.readObject();
    }
}
  
```

Dans le cas de JDBC, une table doit être créée dans un SGBD relationnel. A la création de HighScoreJDBC il y a connexion au SGBD via JDBC.

Un save consiste à faire des « inserts » pour chaque « entry ».

Un load consiste à faire un `Select * from ...`, à parcourir le résultat et à créer les objets « entry » correspondants.


```

public class HighScoreJDBC extends HighScore {
    public static final String url="jdbc:odbc:dice";
    Connection con=null;

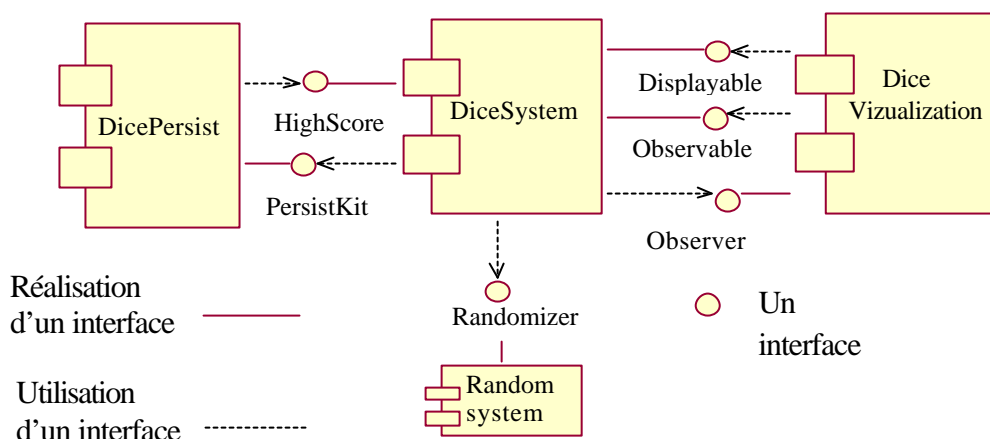
    public HighScoreJDBC() {
        try {
            //charge le driver JDBC
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con=DriverManager.getConnection(url, "toto","");
        } catch (Exception e) {
            e.printStackTrace();
            new Error("Cannot access Database at"+url);
        }
        hs=this;          // enregistrement de l'instance unique!
        this.load();
    }
    public void load() {
        try {
            Statement select=con.createStatement();
            ResultSet result=select.executeQuery ("SELECT Name,Score FROM
                HighScore");
            while (result.next()) {
                this.add(new Entry(result.getString(1), result.getInt(2)));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void save() {
        try {
            for (Enumeration e = this.elements() ; e.hasMoreElements() ;) {
                Entry entry=(Entry)e.nextElement();
                Statement s=con.createStatement();
                s.executeUpdate("INSERT INTO HighScore (Name,Score)"+
                    "VALUES('"+entry.getName()+"', "+
                    entry.getScore()+")");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

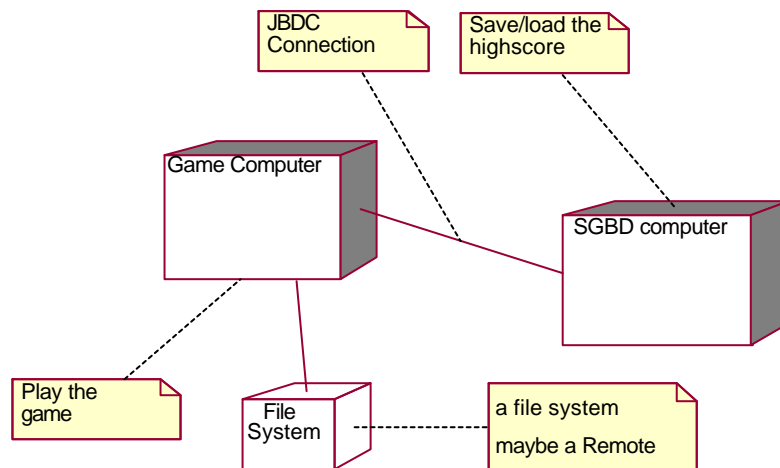
```

Remarque : highscore est une collection d'entrées (d'où le this.elements()).

4.6 Le diagramme de composants



4.7 Le diagramme de déploiement



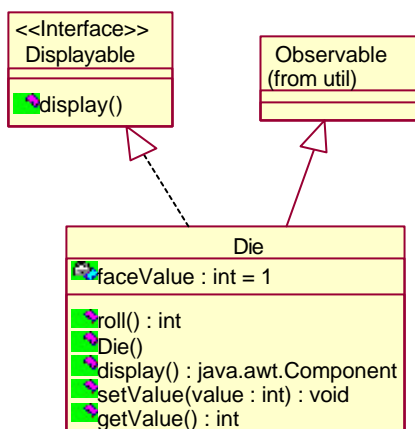
5. L'implantation

5.1. Génération du code

Traduire la conception détaillée dans un langage à objets (Java, C++, Smalltalk, ...) ou non (C, VB, ...). C'est évidemment plus direct vers un langage à objets !

Exemple :

Traduction en Java des composants :



```
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;

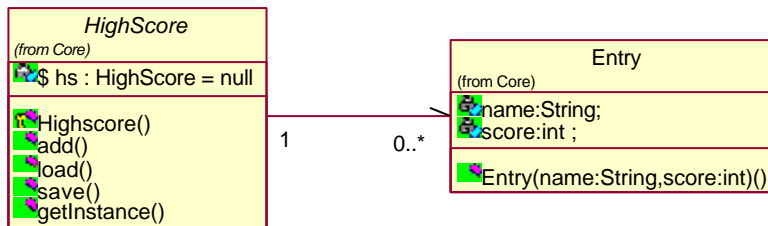
    public int roll() {
        setValue(Randomizer.getInstance().
            getValue());
        return getValue();
    }

    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() {
```

Traduction en Java des associations :

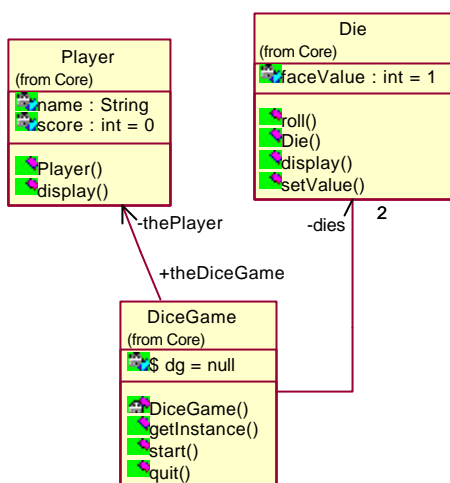


```

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 entries=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() { // implantation de Displayable
        Component c=new HighScoreView(this);
        this.addObserver((java.util.Observer)c);
        return c;
    }
    public static HighScore getInstance() {
  
```

Les environnements de conception (ateliers) UML donnent des outils de génération de code ('forward engineering'), de rétro conception à partir du code ('reverse engineering'). La combinaison des deux étant appelée 'round trip engineering'. L'objectif est d'assurer en permanence la cohérence de l'analyse, de la conception et du code.

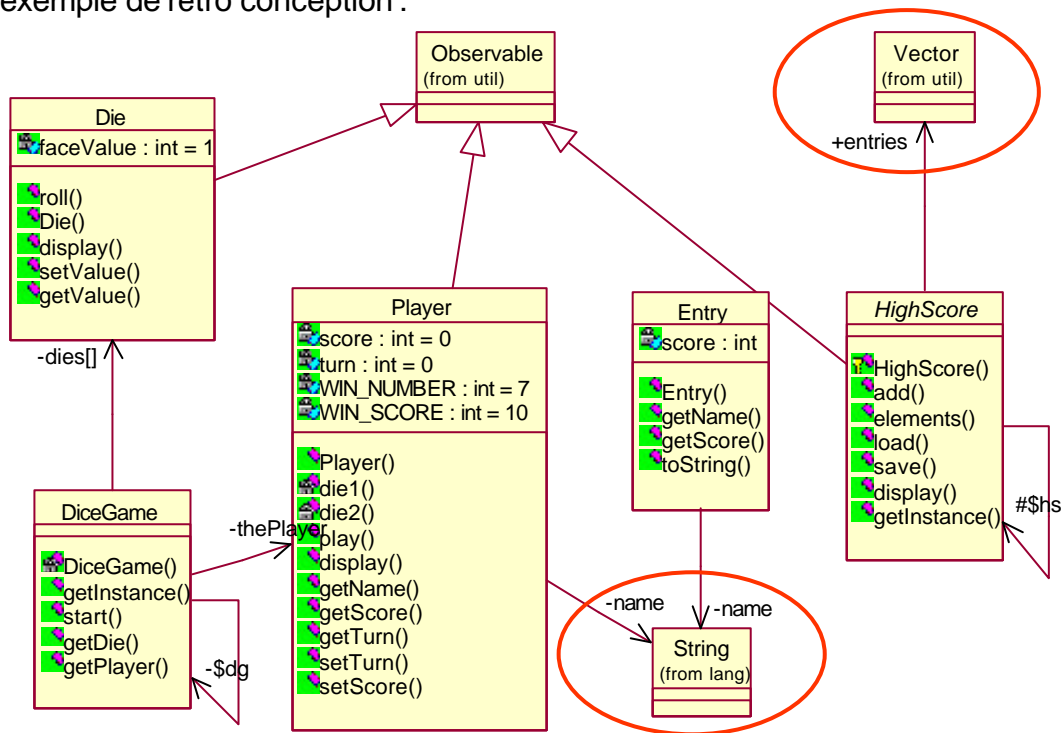
Un exemple de génération :



```

// Source file: c:/UML/Core/DiceGame.java
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
     */
  
```

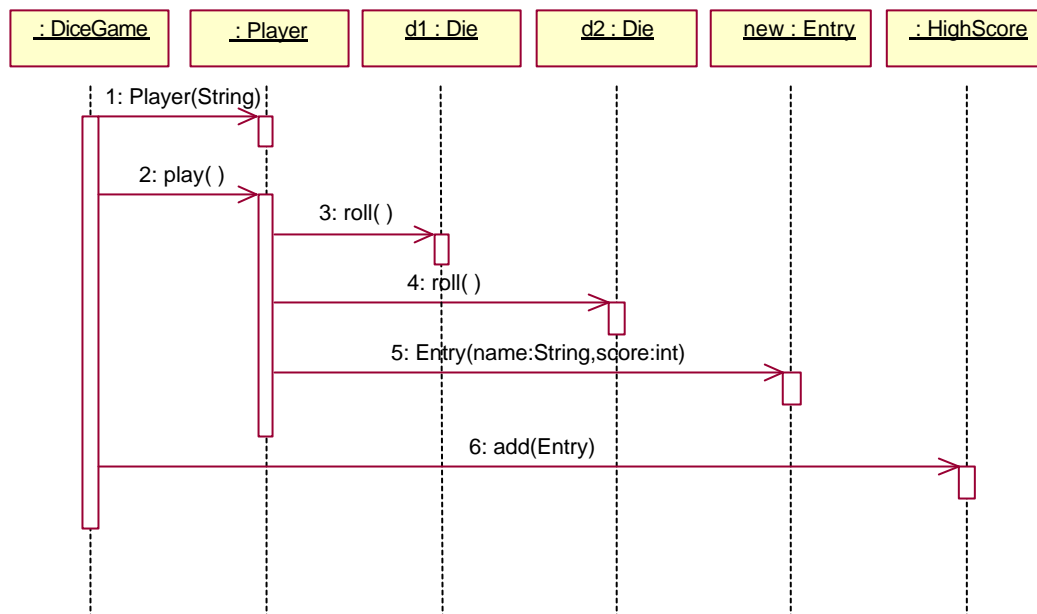
Un exemple de rétro conception :



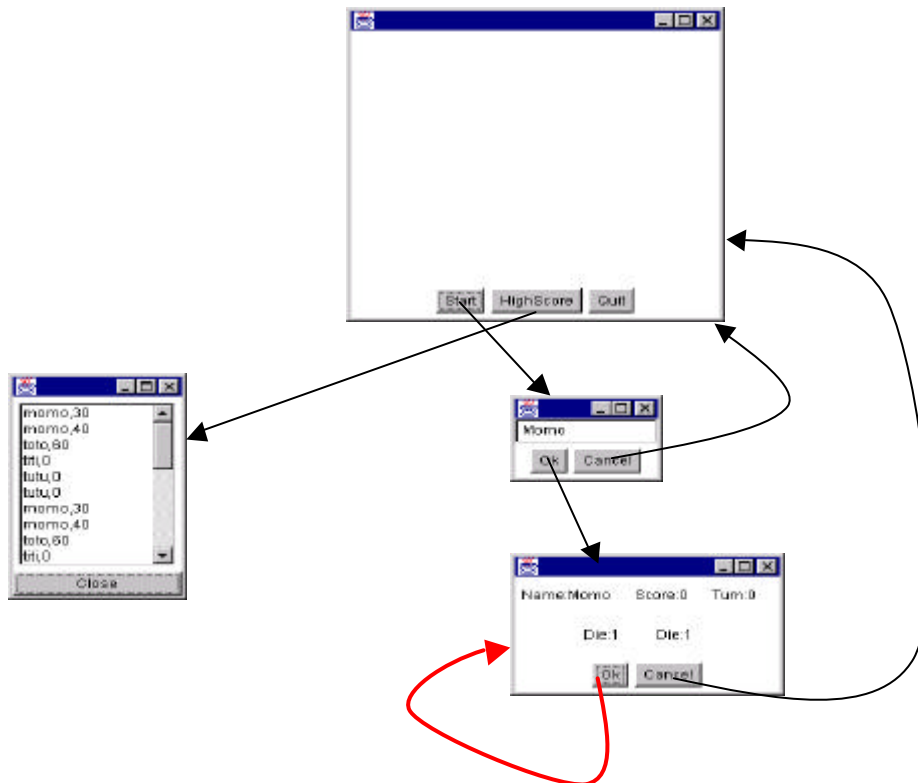
Rien n'est possible pour les aspects dynamiques. Il s'agit donc d'une aide limitée.

5.2. Dernières améliorations

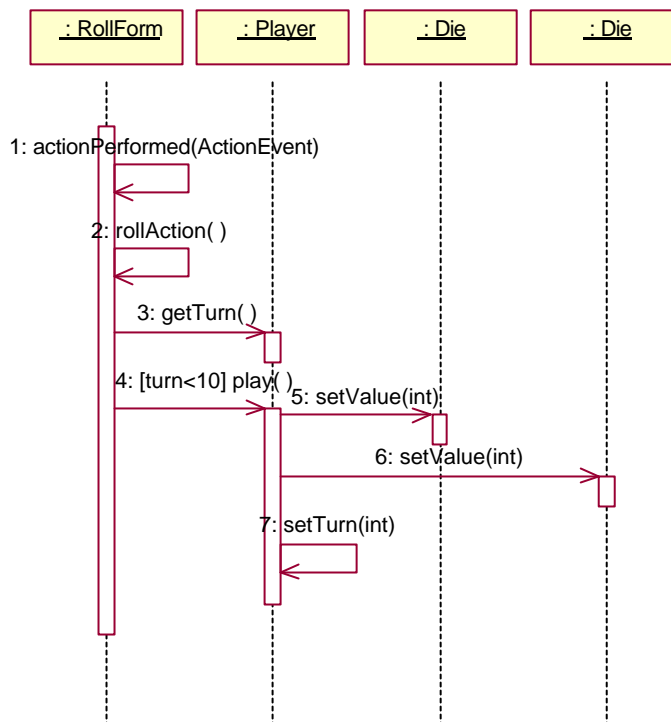
A la fin de l'analyse on avait signalé des manques du côté de la dynamique du cancel et de la gestion du jeu qui était définie par :



Rien n'ayant été fait, il faut reprendre ces aspects en liaison avec la conception détaillée actuelle. Le schéma précédent ne gère pas les 10 tours de jeu et la fin du jeu. La version remaniée doit correspondre au fonctionnement ci dessous.



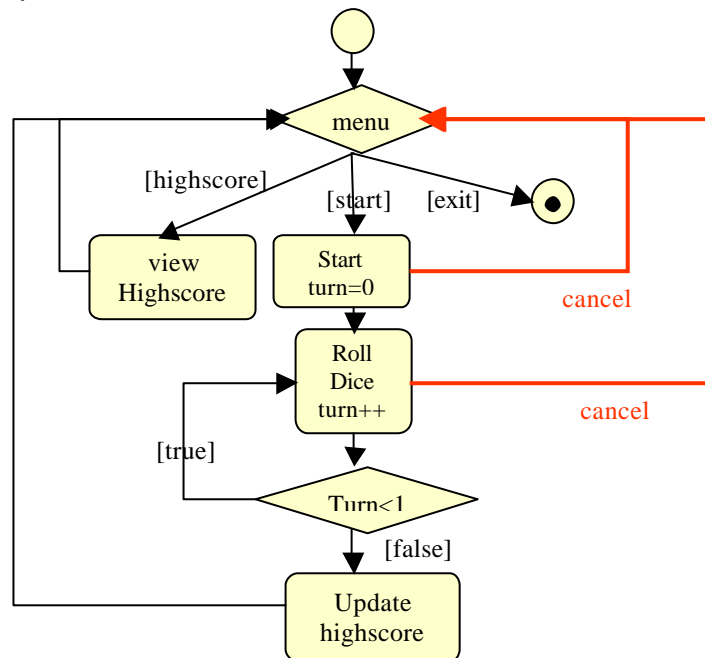
Soit en termes de diagramme de séquences :



6. Les tests

Il faut d'abord tester toutes les méthodes, classe par classe (tests unitaires). Puis il faut tester les composants (tests d'intégration), et enfin, le système tout entier (test d'acceptation au regard des use case et du diagramme d'activités initial définissant les besoins).

Vis à vis des cas d'utilisation et de leur description tout semble correct.
 Vis à vis du diagramme d'activités, il faut tester tous les chemins possibles.
 Regardons ce point.



Le chemin normal start, roll*, highscore, exit marche correctement.

Par contre le chemin highscore, exit plante ! Il s'agit d'un problème de (mauvaise) conception. En effet, c'est DiceGame qui crée Highscore lors du start. Il faut corriger !

Enfin, le chemin highscore, start, roll* doit voir le highscore (dont la fenêtre est déjà ouverte) évoluer dynamiquement si le MVC fonctionne correctement. C'est bon !

7. Conclusions

L'analyse des besoins a utilisé

- cas d'utilisations + descriptions,
- diagramme d'activités,
- prototypage de l'interface utilisateur.

L'analyse a utilisé

- pour la dynamique : les diagrammes de collaborations, séquences, états,
- pour la statique : les diagrammes de classes.

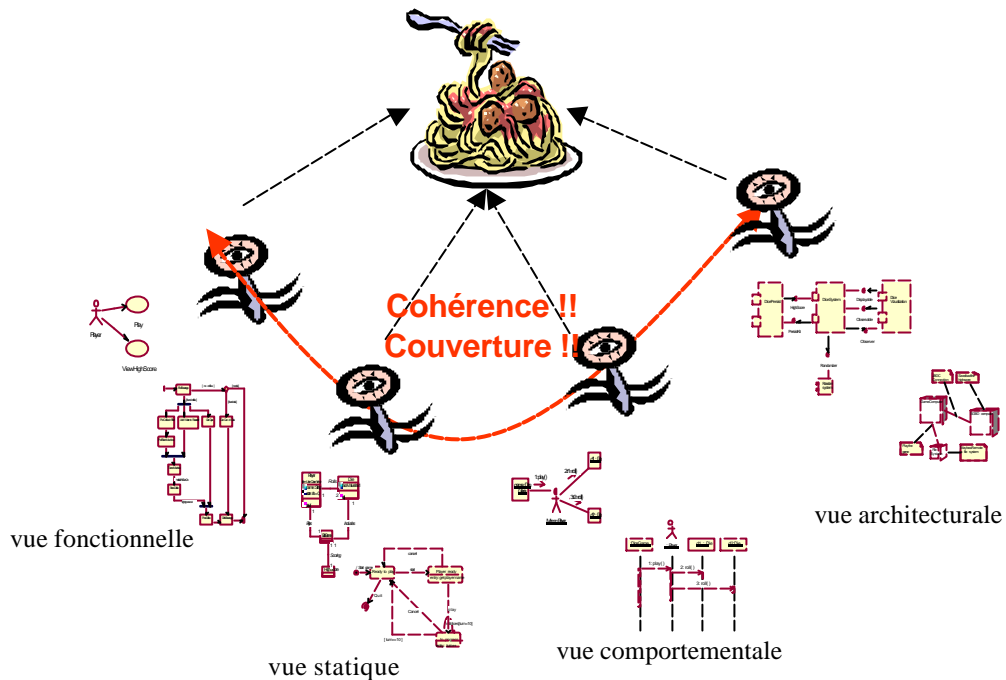
La conception a utilisé :

- le pattern architectural Layer
- les diagrammes de paquetages, de composants, de déploiement,
- les design patterns Singleton, Observer, Factory,
- des classes techniques pour la persistance et l'interface utilisateur.

La réalisation a utilisé les outils de génération et de rétro conception. Il est indispensable de mettre à jour la documentation d'analyse et de conception.

Les tests ont utilisé les cas d'utilisation (couverture des fonctionnalités) et le diagramme d'activité (conformité). Bien entendu sur des grosses applications les démarches de test sont plus complexes (intégration, non régression, ...).

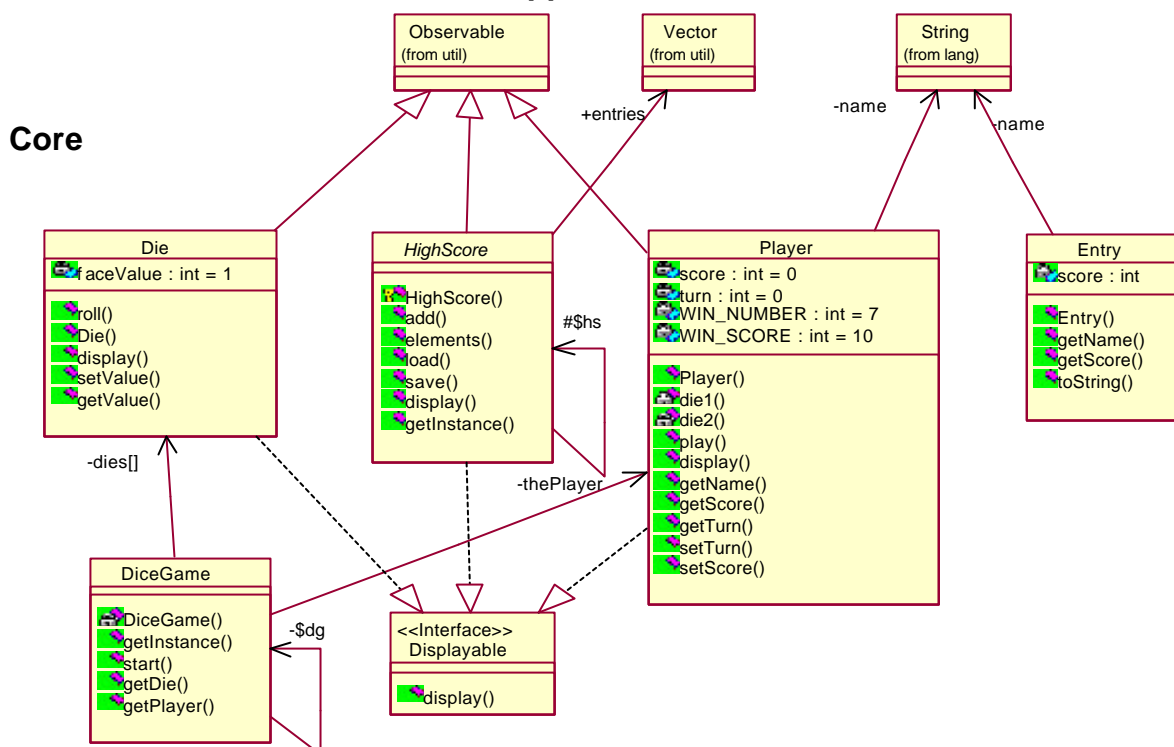
UML permet d'avoir plusieurs points de vue à chaque étape et de réfléchir en termes de couverture et de cohérence. Le travail est facilité avec un atelier UML.



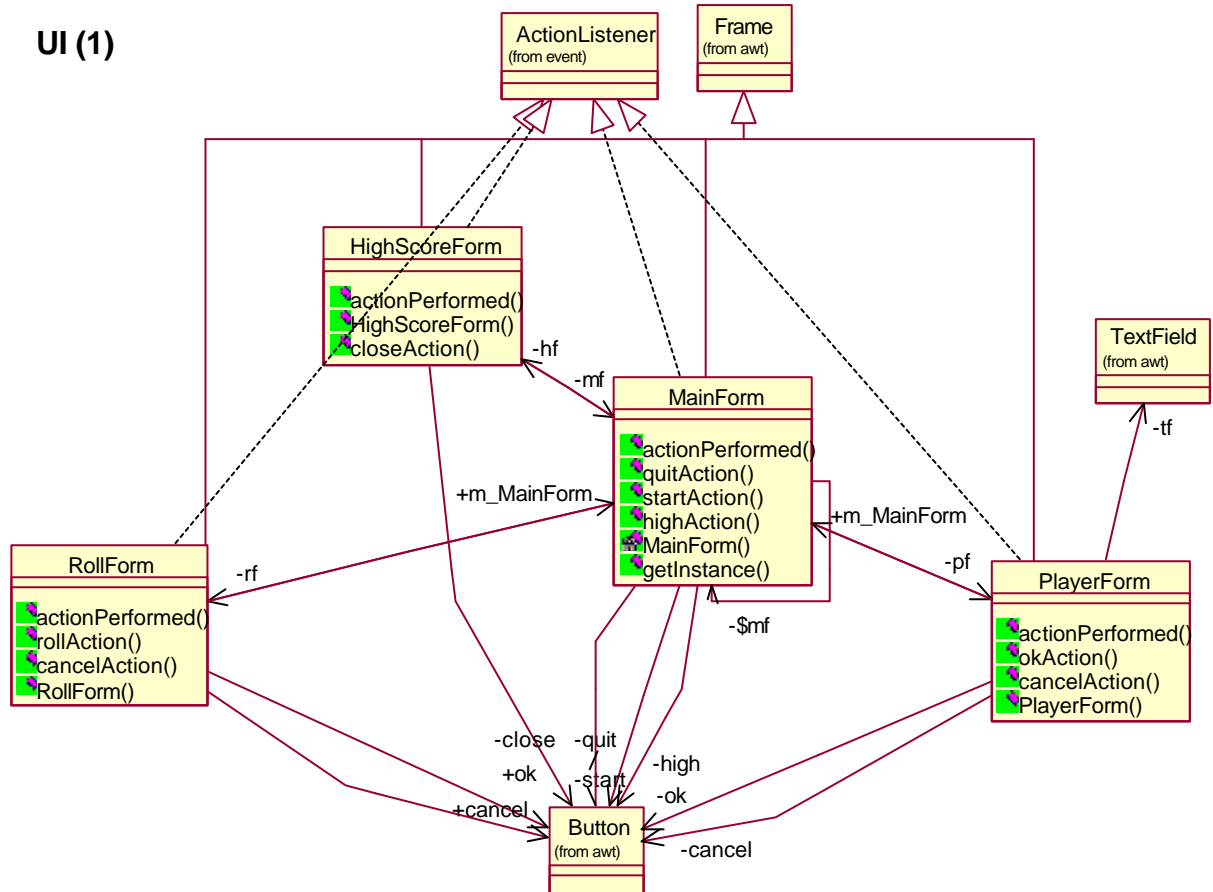
UML permet de documenter les choix d'analyse et de conception.

Grâce au processus de développement maîtrisé, le produit est conforme à ce qui était prévu au départ. Grâce aux patterns, le produit est évolutif (on peut facilement modifier l'interface ou la persistance). Grâce à Java, le produit est portable (système, SGBD).

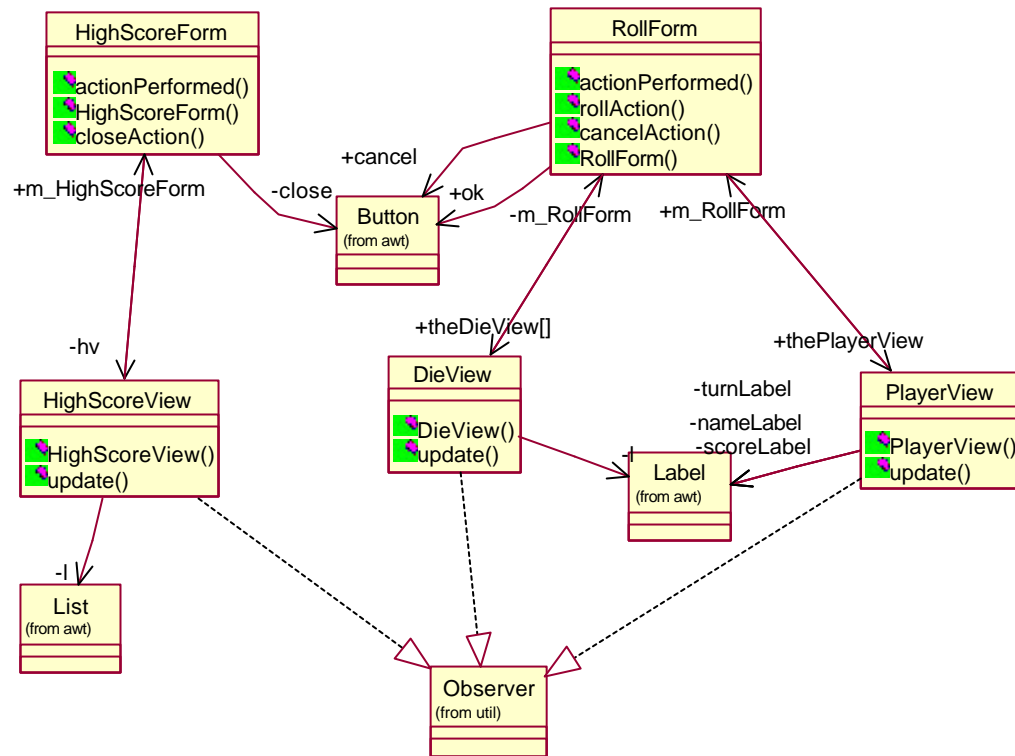
ANNEXE : structuration finale de l'application



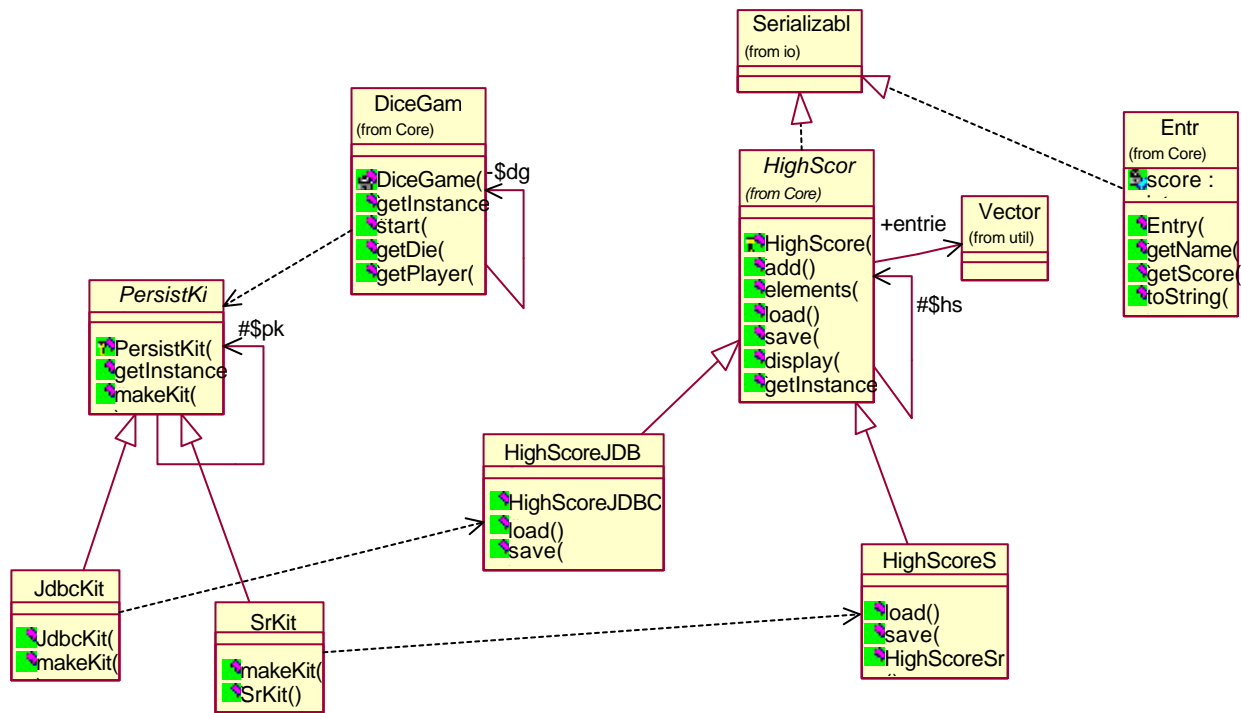
UI (1)



UI (2)



Persist



Util

