Au programme

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille – Sciences et Technologies

Université de Lille

Faculté des sciences et technologies
Département Informatique

Présentation

l'UE COO est le prolongement immédiat de l'UE POO du S4 informatique.

cf. `http://portail.fil.univ-lille1.fr/portail/ls4/poo`

après la programmation... la conception

comment utiliser les concepts objets pour construire des logiciels.

fonctionnement

- équipe pédagogique
  - Quentin Baert, Bilel Derberl, Jérémy Lictevout, Lucien Mousin, Clément Quinton, Jean-François Roos, Yves Roos, Romain Rouvoy, Jean-Christophe Routier
- 1h30 de Cours (8 séances), 2h TD, 1h30 TP par semaine
  - préparer les TD, être **acteur** !
- évaluation continue :
  - TP à rendre
  - première session : $\frac{TP+sup(DS1+2*DS2,3*DS2)}{4}$
  - seconde session : note TP conservée, $\frac{TP+3\times DS3}{4}$

`http://portail.fil.univ-lille1.fr/portail/ls5/coo`

A l'issue de ce module vous devriez...

- **maîtriser les concepts de la programmation objet :**
  - maitriser les différentes manières de mettre en œuvre le polymorphisme ;
  - connaître quelques principes de conception objet et savoir les mettre en œuvre ;
- **être en mesure de concevoir une application avec une approche objet :**
  - savoir faire une analyse objet d'un problème complexe et faire les choix de conception appropriés en utilisant de manière pertinente les concepts et principes des langages à objets ;
  - connaître les principaux patrons de conception (« *design patterns* »), être en mesure d'identifier des situations où ils s'appliquent et les mettre en œuvre ;
  - adopter une méthodologie de conception rigoureuse : documenter un programme, écrire et utiliser des tests ;

Pré-requis objet

- classes et objets
  - instances, constructeurs, méthodes, attributs
  - UML comme notation : diagrammes de classes
- modélisation objet d'un problème
  - analyse d'un problème simple, identification des classes et leur conception,
- **polymorphisme des objets**
  - les interfaces, l'héritage,
  - mécanismes de **lookup** et **liaison tardive** (late-binding)

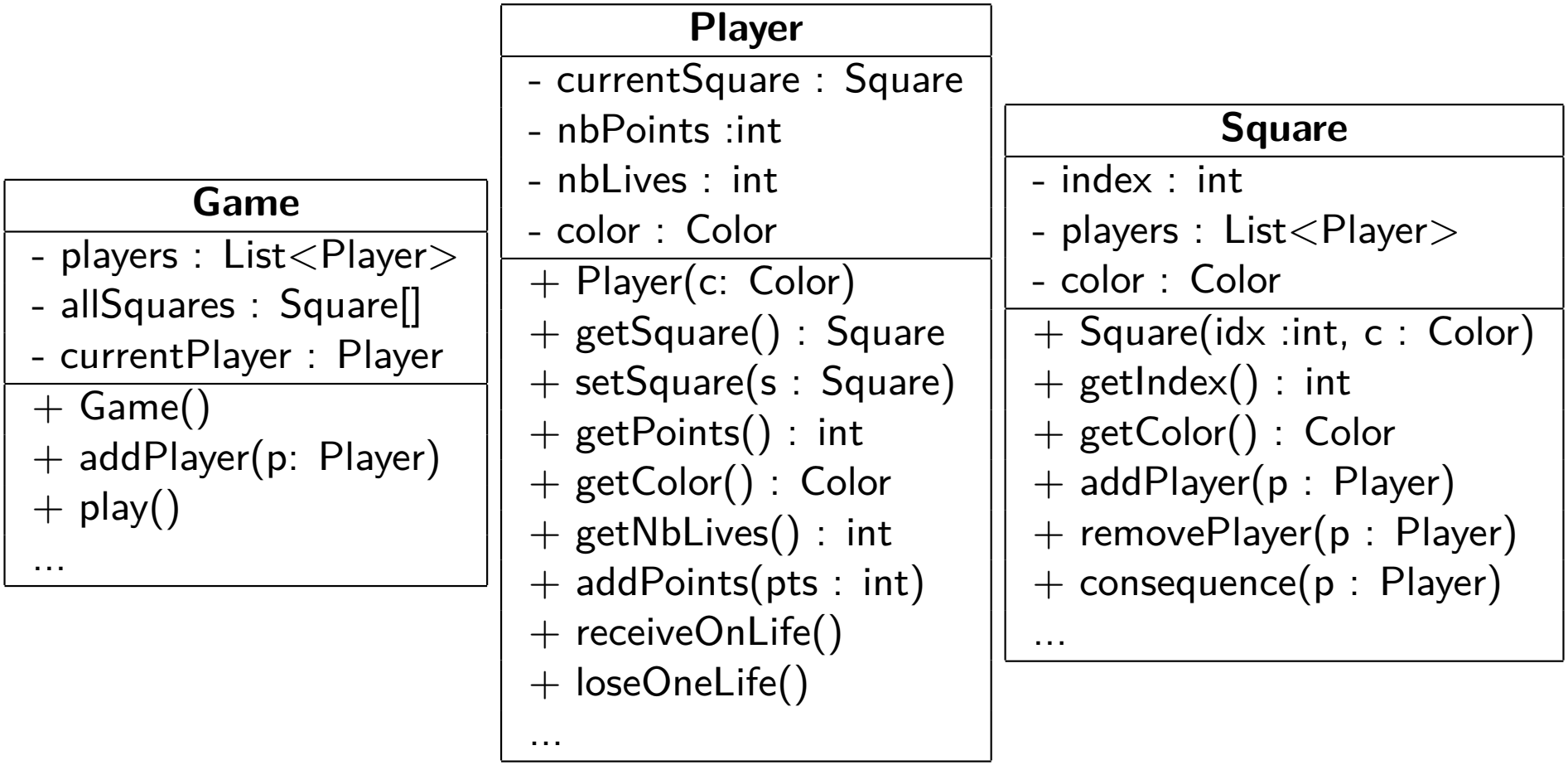
Pré-requis java

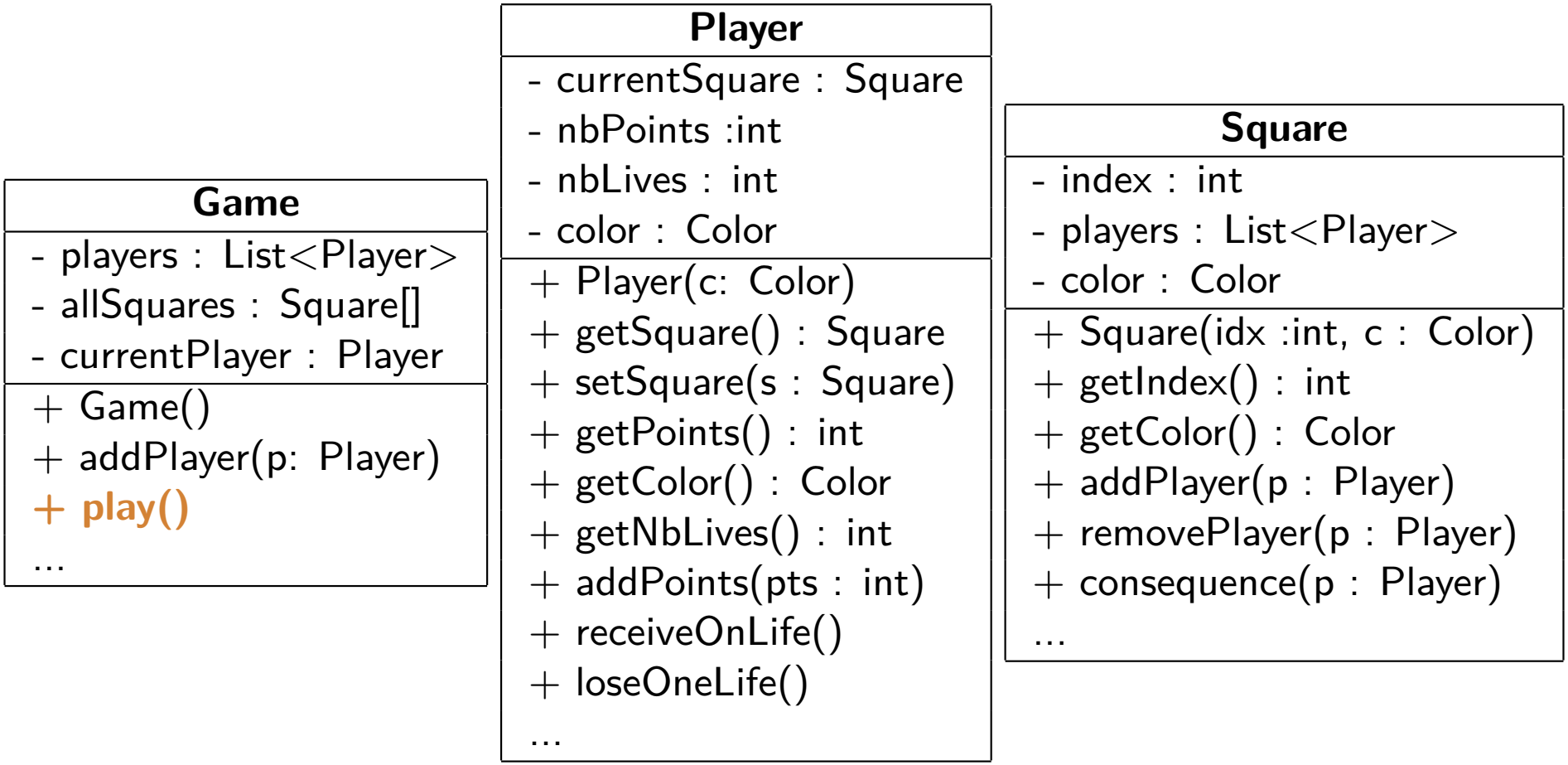
- `class`, `this`, `private`, `public`
  - `toString`, `equals`/`==`
- `static`, `enum`
- `interface`, `extends`, `super`
- exceptions : capture et levée
  - `try ... catch`, `throws`/`throw`
- `java.util` :
  - `Collection<E>`, `Iterator`, `Iterable`
  - `List`/`ArrayList`/`LinkedList`, `Set`,
  - `Map`/`HashMap`, `hashCode`

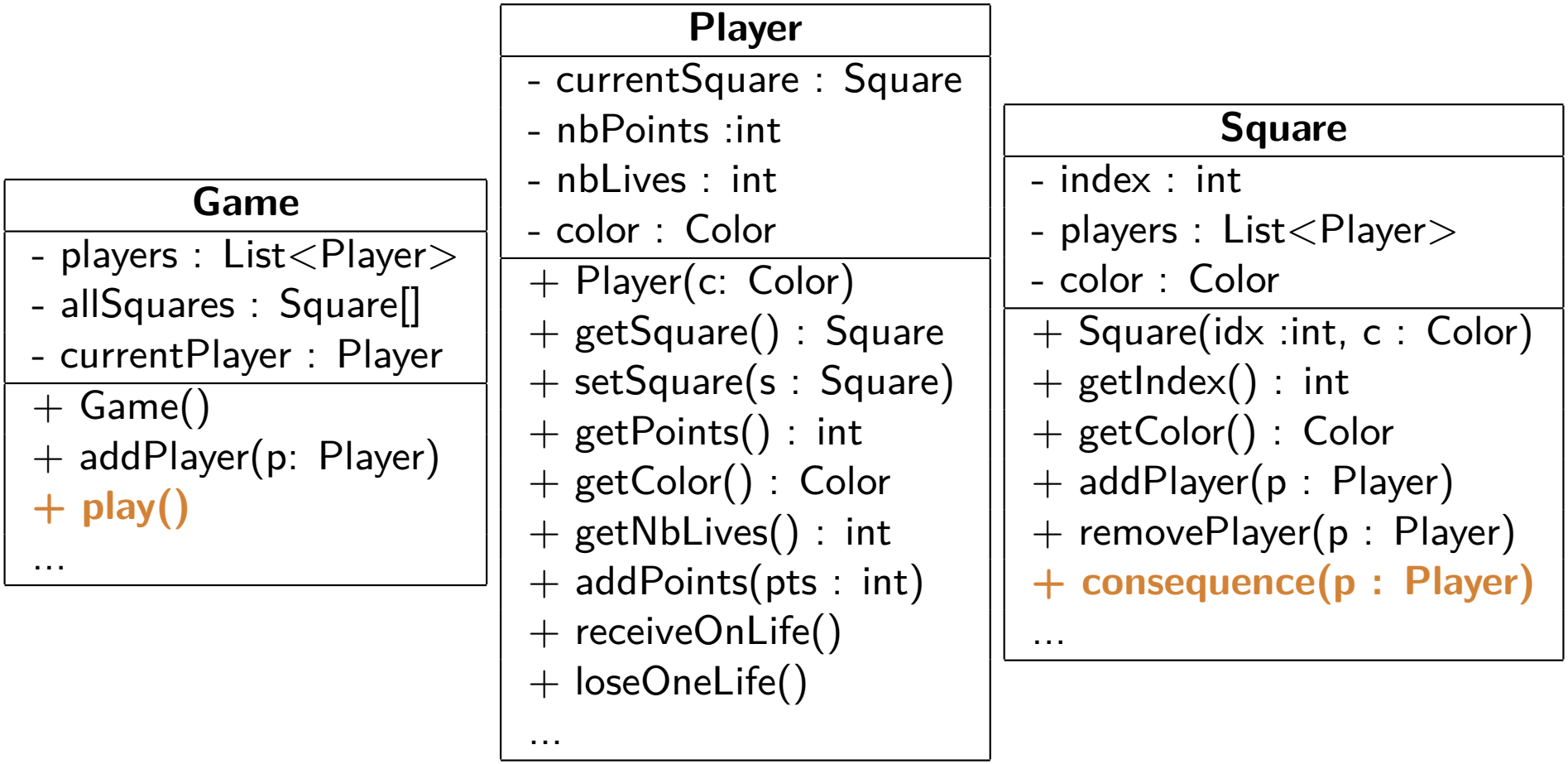
- `package`, `import`
- `javac`, `java`, `jar`, `javadoc`, `classpath`, **tests** (JUnit)
  ↪ "Le TP4" de POO à (re)faire (voir semainier)

Un jeu de plateau simple :

- une piste constituée de cases numérotées de différentes couleurs ;
- les joueurs sont représentés chacun par un pion de couleurs différentes, ces couleurs sont éventuellement les mêmes que celles des cases. Chaque joueur a 3 vies en début de partie ;
- les joueurs jouent chacun leur tour en lançant deux dés et en avançant d'autant de cases ;
- lorsqu'un joueur s'arrête sur une case, la conséquence est qu'il marque des points et la case produit un effet. Les points marqués sont doublés si la couleur du joueur est la même que celle de la case. Le nombre de points marqués et l'effet varient en fonction de la couleur de la case. Certaines cases font marquer plus de points, d'autre perdre des vies, etc.
- le jeu se termine quand un joueur a marqué plus qu'un nombre de points fixé à l'avance ou quand tous les joueurs sauf un ont perdu toutes leurs vies.


Game
- players : List<Player>
- allSquares : Square[]
- currentPlayer : Player
+ Game()
+ addPlayer(p: Player)
+ play()
...
Player
- currentSquare : Square
- nbPoints :int
- nbLives : int
- color : Color
+ Player(c: Color)
+ getSquare() : Square
+ setSquare(s : Square)
+ getPoints() : int
+ getColor() : Color
+ getNbLives() : int
+ addPoints(pts : int)
+ receiveOnLife()
+ loseOneLife()
...
Square
- index : int
- players : List<Player>
- color : Color
+ Square(idx :int, c : Color)
+ getIndex() : int
+ getColor() : Color
+ addPlayer(p : Player)
+ removePlayer(p : Player)
+ consequence(p : Player)
...


Game
- players : List<Player>
- allSquares : Square[]
- currentPlayer : Player
+ Game()
+ addPlayer(p: Player)
+ play()
...
Player
- currentSquare : Square
- nbPoints :int
- nbLives : int
- color : Color
+ Player(c: Color)
+ getSquare() : Square
+ setSquare(s : Square)
+ getPoints() : int
+ getColor() : Color
+ getNbLives() : int
+ addPoints(pts : int)
+ receiveOnLife()
+ loseOneLife()
...
Square
- index : int
- players : List<Player>
- color : Color
+ Square(idx :int, c : Color)
+ getIndex() : int
+ getColor() : Color
+ addPlayer(p : Player)
+ removePlayer(p : Player)
+ consequence(p : Player)
...


Game
- players : List<Player>
- allSquares : Square[]
- currentPlayer : Player
+ Game()
+ addPlayer(p: Player)
+ play()
...
Player
- currentSquare : Square
- nbPoints :int
- nbLives : int
- color : Color
+ Player(c: Color)
+ getSquare() : Square
+ setSquare(s : Square)
+ getPoints() : int
+ getColor() : Color
+ getNbLives() : int
+ addPoints(pts : int)
+ receiveOnLife()
+ loseOneLife()
...
Square
- index : int
- players : List<Player>
- color : Color
+ Square(idx :int, c : Color)
+ getIndex() : int
+ getColor() : Color
+ addPlayer(p : Player)
+ removePlayer(p : Player)
+ consequence(p : Player)
...

dans la classe Game :

```
/** ... */
public void play() {
   Iterator<Player>  playerIt = this.players.iterator();
   boolean finished = false;
   while (!finished) {
      if (!playerIt.hasNext()) {
         playerIt = this.player.iterator();
      }
      this.currentPlayer = playerIt.next();
      int dice = this.currentPlayer.throwDice();
      int currentSquareIndex = this.currentPlayer.getSquare().getIndex();
      int nextSquareIndex = this.allSquares[currentSquareIndex+dice];
      Square nextSquare = this.allSquares[nextSquareIndex];
      this.currentPlayer.getSquare().removePlayer(this.currentPlayer);
      this.currentPlayer.setSquare(nextSquare);
      nextSquare.addPlayer(this.currentPlayer);
      nextSquare.consequence(this.currentPlayer);
      if (this.currentPlayer.getNbLives() < 0) {
        this.players.remove(this.currentPlayer);
      }
      finished = this.currentPlayer.getNbPoints() > 1000
                || this.players.size() == 1;
   }
   Player winner;
   if (this.players.size() == 1) {
      winner = this.players.get(0);
   } else {
      winner = this.currentPlayer;
   }
   System.out.println("the winner is "+winner);
}
```

arghhhh...

```
public void play() {
  Iterator<Player> playerIt = this.players.iterator();
  boolean finished = false;
  while (!finished) {

    if (!playerIt.hasNext()) {
      playerIt = this.player.iterator()
    }
    this.currentPlayer = playerIt.next();

    int dice = this.currentPlayer.throwDice();
    int currentSquareIndex = this.currentPlayer.getSquare().getIndex();
    int nextSquareIndex = this.allSquares[currentSquareIndex+dice];
    Square nextSquare = this.allSquares[nextSquareIndex];

    this.currentPlayer.getSquare().removePlayer(this.currentPlayer);
    this.currentPlayer.setSquare(nextSquare);
    nextSquare.addPlayer(this.currentPlayer);

    nextSquare.consequence(this.currentPlayer);

    if (this.currentPlayer.getNbLives() < 0) {
      this.players.remove(this.currentPlayer);
    }
    finished = this.currentPlayer.getNbPoints() > Game.WINNING_POINTS
             || this.players.size() == 1;
  }

  Player winner;
  if (this.players.size() == 1) {
    winner = this.players.get(0);
  } else {
    winner = this.currentPlayer;
  }
  System.out.println("the winner is "+ winner);
}
```

```
public void play() {
  Iterator<Player> playerIt = this.players.iterator();
  boolean finished = false;
  while (!finished) {
  // compute next player
    if (!playerIt.hasNext()) {
      playerIt = this.player.iterator()
    }
    this.currentPlayer = playerIt.next();
  // compute new player's square
    int dice = this.currentPlayer.throwDice();
    int currentSquareIndex = this.currentPlayer.getSquare().getIndex();
    int nextSquareIndex = this.allSquares[currentSquareIndex+dice];
    Square nextSquare = this.allSquares[nextSquareIndex];
  // move player
    this.currentPlayer.getSquare().removePlayer(this.currentPlayer);
    this.currentPlayer.setSquare(nextSquare);
    nextSquare.addPlayer(this.currentPlayer);
  // apply new square's consequence
    nextSquare.consequence(this.currentPlayer);
  // end of game is reached ???
    if (this.currentPlayer.getNbLives() < 0) {
      this.players.remove(this.currentPlayer);
    }
    finished = this.currentPlayer.getNbPoints() > Game.WINNING_POINTS
               || this.players.size() == 1;
  }
  // who is the winner ?
  Player winner;
  if (this.players.size() == 1) {
    winner = this.players.get(0);
  } else {
    winner = this.currentPlayer;
  }
  System.out.println("the winner is "+ winner);
}
```

re arghhhh...

décomposer

il faut

dé-com-po-ser

responsabilité unique

une méthode, une fonctionnalité

```
public void play() {
  Iterator<Player> playerIt = this.players.iterator();
  boolean finished = false;
  while (!finished) {
  // compute next player
    if (!playerIt.hasNext()) {
      playerIt = this.player.iterator()
    }
    this.currentPlayer = playerIt.next();
  // compute new player's square
    int dice = this.currentPlayer.throwDice();
    int currentSquareIndex = this.currentPlayer.getSquare().getIndex();
    int nextSquareIndex = this.allSquares[currentSquareIndex+dice];
    Square nextSquare = this.allSquares[nextSquareIndex];
  // move player
    this.currentPlayer.getSquare().removePlayer(this.currentPlayer);
    this.currentPlayer.setSquare(nextSquare);
    nextSquare.addPlayer(this.currentPlayer);
  // apply new square's consequence
    nextSquare.consequence(this.currentPlayer);
  // end of game is reached ???
    if (this.currentPlayer.getNbLives() < 0) {
      this.players.remove(this.currentPlayer);
    }
    finished = this.currentPlayer.getNbPoints() > Game.WINNING_POINTS
               || this.players.size() == 1;
  }
  // who is the winner ?
  Player winner;
  if (this.players.size() == 1) {
    winner = this.players.get(0);
  } else {
    winner = this.currentPlayer;
  }
  System.out.println("the winner is "+ winner);
}
```

```
public void play() {

  while (!finished) {
  // compute next player



  // compute new player's square



  // move player



  // apply new square's consequence
    nextSquare.consequence(this.currentPlayer);
  // end of game is reached ???


    finished =

  }
  // who is the winner ?



}
```

et donc...

```
public void play() {
  while (! this.isFinished()) {                    // end of game is reached ???
    this.currentPlayer = this.nextPlayer();        // compute next player
    Square nextSquare = this.computeNextSquare(); // compute new player's square
    this.moveCurrentPlayer(nextSquare);            // move player
    nextSquare.consequence(this.currentPlayer);    // apply new square's consequence
  }
  this.displayWinner();                            // who is the winner ?
}
```

une seule responsabilité :
gérer la « boucle principale » du jeu

avec :

```
protected boolean isFinished() {
  if (this.currentPlayer.getNbLives() < 0) {
    this.players.remove(this.currentPlayer);
  }
  return this.currentPlayer.getNbPoints() > WINNING_POINTS || this.players.size() == 1;
}
```

responsabilité : déterminer si le jeu est fini

```
protected Player nextPlayer() { ... }
```

responsabilité : choisir le prochain joueur

```
protected Square computeNextSquare() {
  int dice = this.currentPlayer.throwDice();
  int currentSquareIndex = this.currentPlayer.getSquare().getIndex();
  return this.allSquares[currentSquareIndex+dice];
}
```

responsabilité : calculer nouvelle case du joueur

```
protected void moveCurrentPlayer(Square nextSquare) {
  this.currentPlayer.getSquare().removePlayer(this.currentPlayer);
  this.currentPlayer.setSquare(nextSquare);
  nextSquare.addPlayer(this.currentPlayer);
}
```

responsabilité : déplacer le joueur

```
public void displayWinner() {
  Player winner;
  if (this.players.size() == 1) {
    winner = this.players.get(0);
  } else {
    winner = this.currentPlayer;
  }
  System.out.println("the winner is "+ winner);
}
```

responsabilité : afficher le vainqueur

cool...

consequence

- *lorsqu'un joueur s'arrête sur une case, la conséquence est qu'il marque des points et la case produit un effet. Les points marqués sont doublés si la couleur du joueur est la même que celle de la case. Le nombre de points marqués et l'effet varient en fonction de la couleur de la case. Certaines cases font marquer plus de points, d'autre perdre des vies, etc.*

Si la couleur de la case est :

- *bleu*, le joueur marque 100 points et reçoit un bonus entre 10 et 50 points
- *jaune*, le joueur marque 50 points et gagne une vie
- *orange*, la conséquence est le double de celle d'une case jaune
- *rouge*, le joueur marque -100 points et perd une vie

```
nextSquare.consequence(this.currentPlayer);
```

`consequence` doit (« contrat »)

- ajouter au joueur le nombre points correspondant à la case (points doublés si les couleurs correspondent)
- afficher le nombre de points gagnés
- appliquer l'effet correspondant à la couleur de la case
- indiquer si le joueur a perdu toutes ses vies

dans Square :

```java
/** add points to the player and apply square effect. Points and effect depend
 * on square color. Won points are displaid and player is notified if (s)he has
 * lost all its lives.
 * @param player the player reaching this square
 */
public void consequence(Player player) {
  // points management
  int points = this.getNbPoints();
  if (player.getColor().equals(this.color)) { // double points if same color
    points = 2 * points;
  }
  player.addPoints(points);
  System.out.println(player+" gets "+points);

 // apply the square effect
  if (this.color == Color.BLUE) {
    player.addPoints(alea.nextInt(40)+10);
  } else if (this.color.equals(Color.YELLOW)) {
    player.receiveOneLife();
  } else if (this.color == Color.ORANGE) {
    player.receiveOneLife();
    player.receiveOneLife();
  } else if (this.color == Color.RED) {
    player.loseOneLife();
  }

  // has the player lost all its lives ?
  if (player.getNbLives() <= 0) {
    System.out.println(player +" is eliminated, no more life");
  }
}
```

arghhhh...

évolution

ajout d'une nouvelle couleur de case

si la couleur de la case est :

- *noir, le joueur perd deux vies*

plusieurs comportements ⇒ plusieurs types

polymorphisme + late-binding

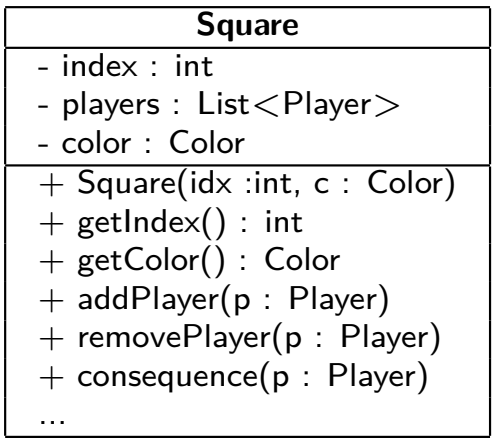
différents types de `Square`

Square
- index : int - players : List<Player> - color : Color
+ Square(idx :int, c : Color) + getIndex() : int + getColor() : Color + addPlayer(p : Player) + removePlayer(p : Player) + consequence(p : Player) ...

Squares

différents types de cases :

- différentes classes: `BlueSquare`, `YellowSquare`, etc.
- doivent posséder la même interface publique
  = répondent aux mêmes messages
  ⇒ doivent partager un type commun (`Square`)
- partagent des comportements (`getColor`, `addPlayer`,...)
- partagent des attributs (`index`, `players`, `color`)
- diffèrent sur les comportements de `consequence`
  pas de « comportement par défaut »

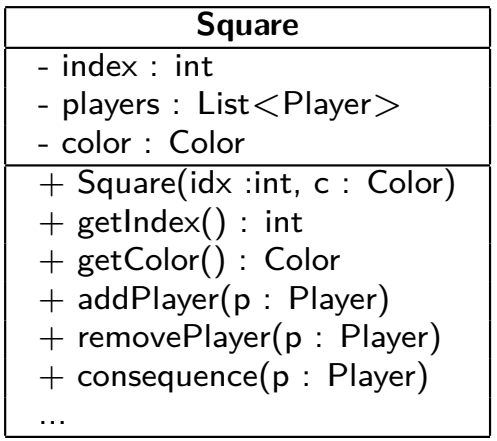


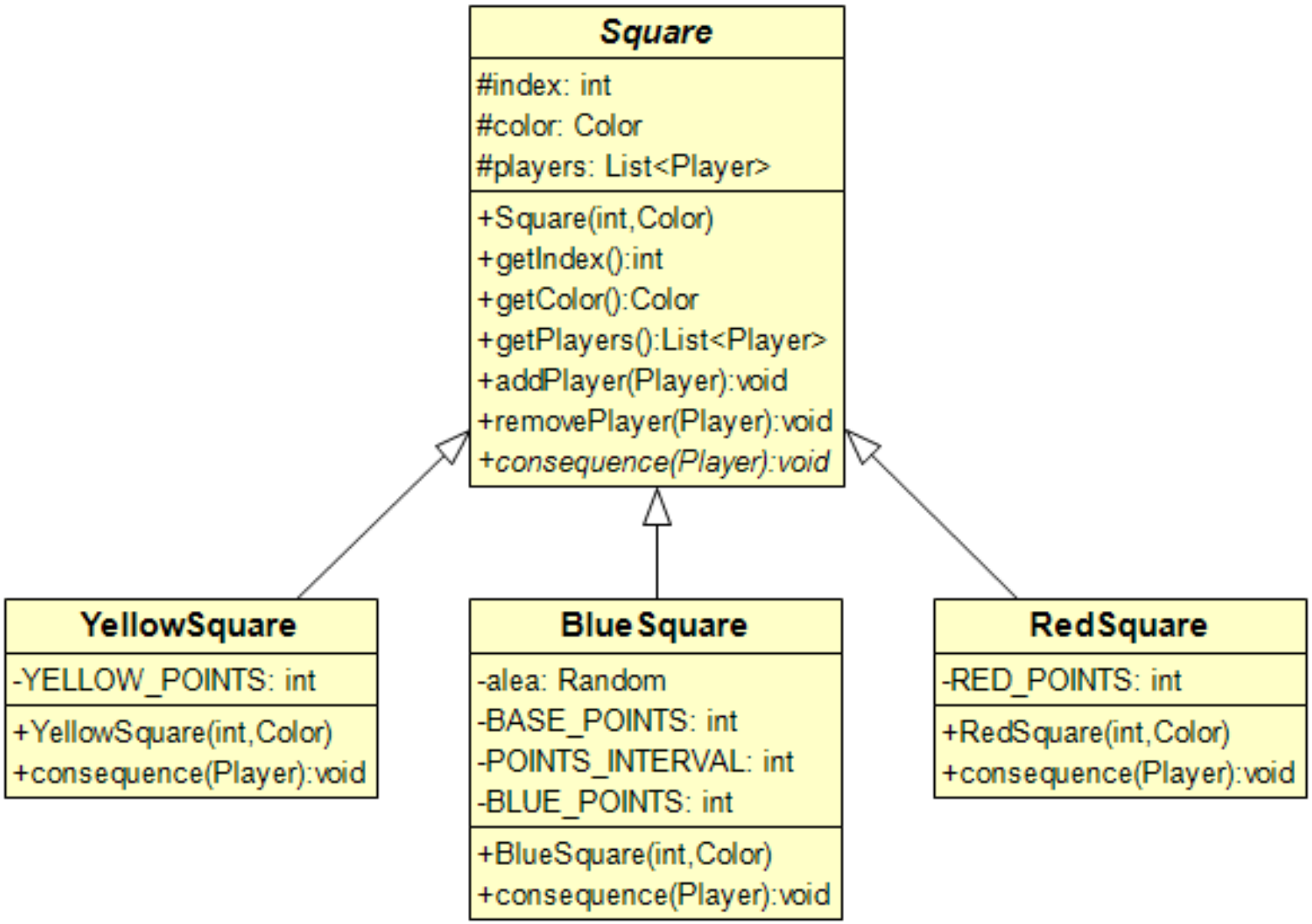
Squares

différents types de cases :

- différentes classes: `BlueSquare`, `YellowSquare`, etc.
- doivent posséder la même interface publique
  = répondent aux mêmes messages
  ⇒ doivent partager un type commun (`Square`)
- partagent des comportements (`getColor`, `addPlayer`,...)
- partagent des attributs (`index`, `players`, `color`)
- diffèrent sur les comportements de `consequence`
  pas de « comportement par défaut »

classe abstraite + héritage



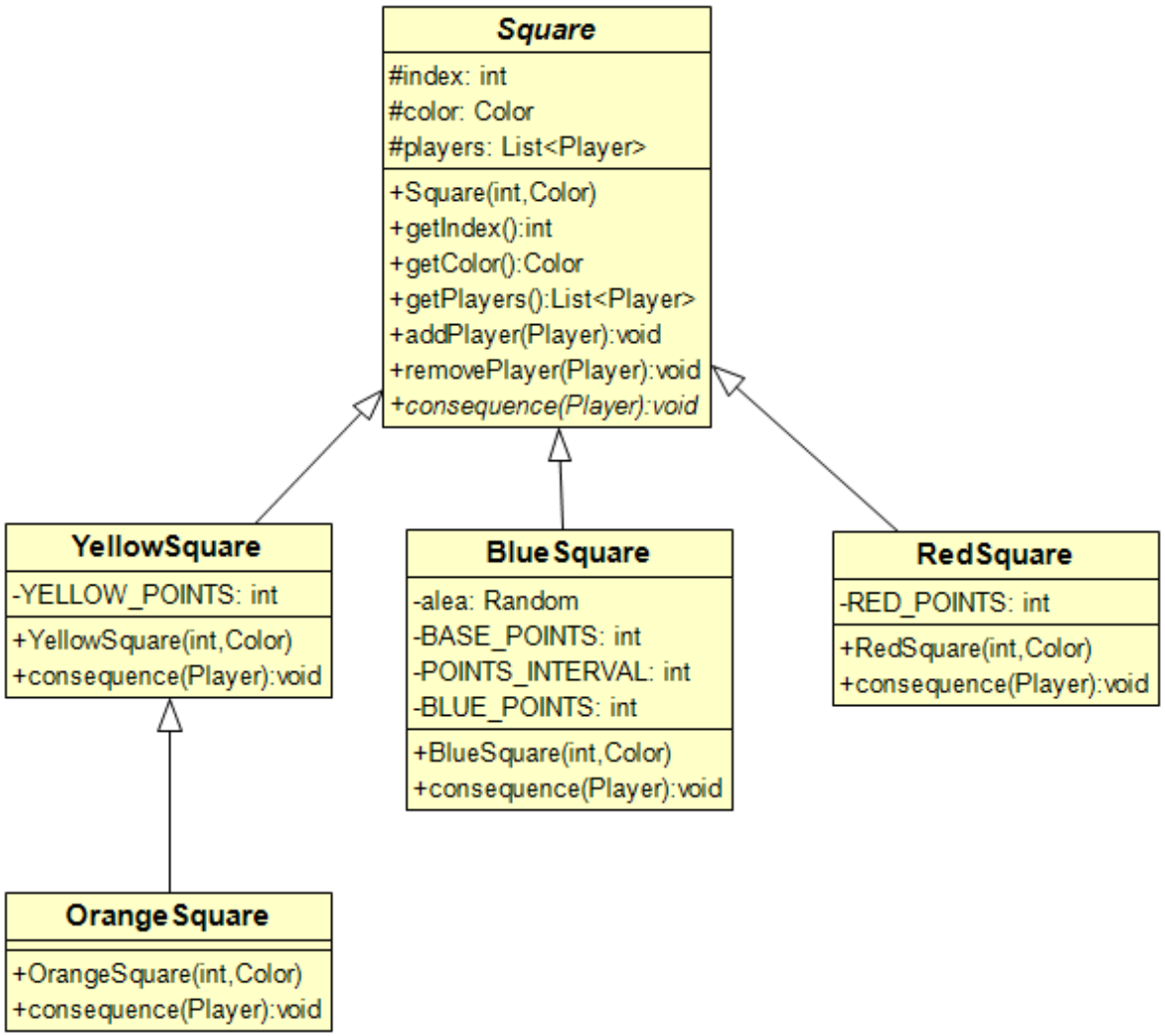

Square
#index: int
#color: Color
#players: List<Player>
+Square(int,Color)
+getIndex():int
+getColor():Color
+getPlayers():List<Player>
+addPlayer(Player):void
+removePlayer(Player):void
+consequence(Player):void
YellowSquare
-YELLOW_POINTS: int
+YellowSquare(int,Color)
+consequence(Player):void
BlueSquare
-alea: Random
-BASE_POINTS: int
-POINTS_INTERVAL: int
-BLUE_POINTS: int
+BlueSquare(int,Color)
+consequence(Player):void
RedSquare
-RED_POINTS: int
+RedSquare(int,Color)
+consequence(Player):void

```java
public class YellowSquare extends Square {
  private static final int YELLOW_POINTS = 50;

  public YellowSquare(int index) {
    super(index, Color.YELLOW);
  }

  public void consequence(Player player) {
    // points management
    int points = YELLOW_POINTS;
    if (player.getColor() == this.color) {
      points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);

    // apply the square effect
    player.receiveOneLife();

    // has the player lost all its lives ?
    if (player.getNbLives() <= 0) {
      System.out.println(player +" is ...")
    }
  }

}
```

```java
public class BlueSquare extends Square {
  private static final Random alea = new Random();
  private static final int BLUE_POINTS = 100;

  public BlueSquare(int index) {
    super(index, Color.BLUE);
  }

  public void consequence(Player player) {
    // points management
    int points = BLUE_POINTS;
    if (player.getColor() == this.color) {
      points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);

    // apply the square effect
    player.addPoints(alea.nextInt(40)+10);

    // has the player lost all its lives ?
    if (player.getNbLives() <= 0) {
      System.out.println(player +" is ...");
    }
  }

}
```

si la couleur de la case est :

- *orange, la conséquence est le double de celle d'une case jaune*



si la couleur de la case est :

- *orange, la conséquence est le double de celle d'une case jaune*

```java
public class YellowSquare extends Square {
  private static final int YELLOW_POINTS = 50;

  public YellowSquare(int index) {
    super(index, Color.YELLOW);
  }

  public void consequence(Player player) {
    // points management
    int points = YELLOW_POINTS;
    if (player.getColor() == this.color) {
      points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);

    // apply the square effect
    player.receiveOneLife();

    // has the player lost all its lives ?
    if (player.getNbLives() <= 0) {
      System.out.println(player +" is ...")
    }
  }
}
```

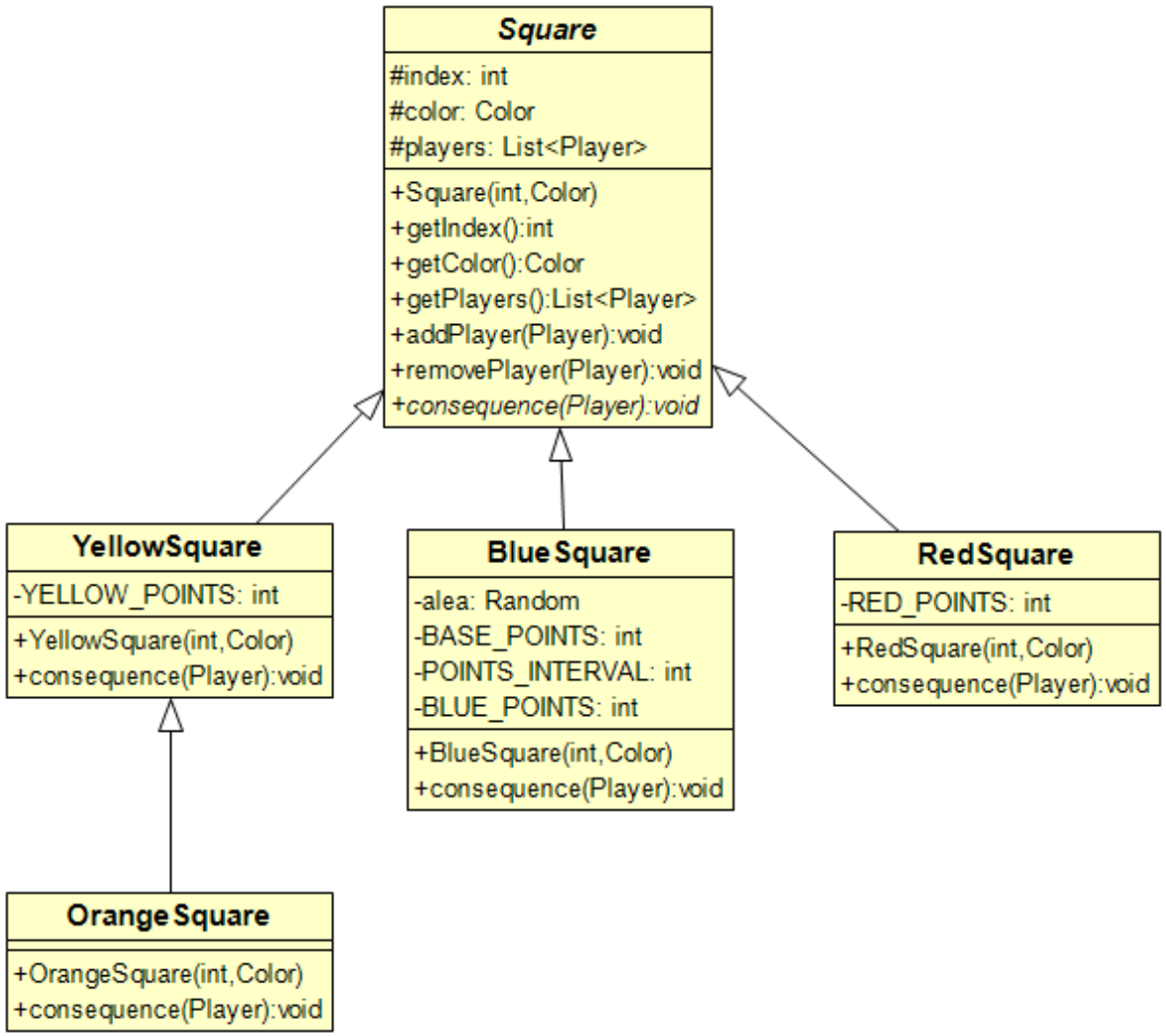
```java
public class Orange Square extends Square {
  private static final Random alea = new Random();
  private static final int ORANGE_POINTS = 100;

  public OrangeSquare(int index) {
    super(index, Color.ORANGE);
  }

  public void consequence(Player player) {
    // points management
    int points = ORANGE_POINTS;
    if (player.getColor() == this.color) {
      points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);

    // apply the square effect
    player.receiveOneLife();
    player.receiveOneLife();
    // has the player lost all its lives ?
    if (player.getNbLives() <= 0) {
      System.out.println(player +" is ...");
    }
  }
}
```

arghhhh...

évolution du code de `YellowSquare` ?

si la couleur de la case est :

- *orange, la conséquence est le double de celle d'une case jaune*



utiliser `super`

```
public class OrangeSquare extends YellowSquare {


    public OrangeSquare(int index) {
      super(index);
      this.color = Color.ORANGE;
    }

    public void consequence(Player player) {
      super.consequence(player);
      super.consequence(player);
    }

}
```

respect du contrat :

« *la conséquence d'une case orange est le double de celle d'une case jaune* »

late binding

```
public void play() {
    ...
    Square nextSquare = this.computeNextSquare();
    ...
    nextSquare.consequence(this.currentPlayer);
    ...
}
```

quel code est appelé ?

late binding

```
public void play() {
    ...
    Square nextSquare = this.computeNextSquare();
    ...
    nextSquare.consequence(this.currentPlayer);
    ...
}
```

quel code est appelé ?

si `nextSquare` fait référence à un objet

- `YellowSquare` ?
- `BlueSquare` ?

late binding

```
public void play() {
    ...
    Square nextSquare = this.computeNextSquare();
    ...
    nextSquare.consequence(this.currentPlayer);
    ...
}
```

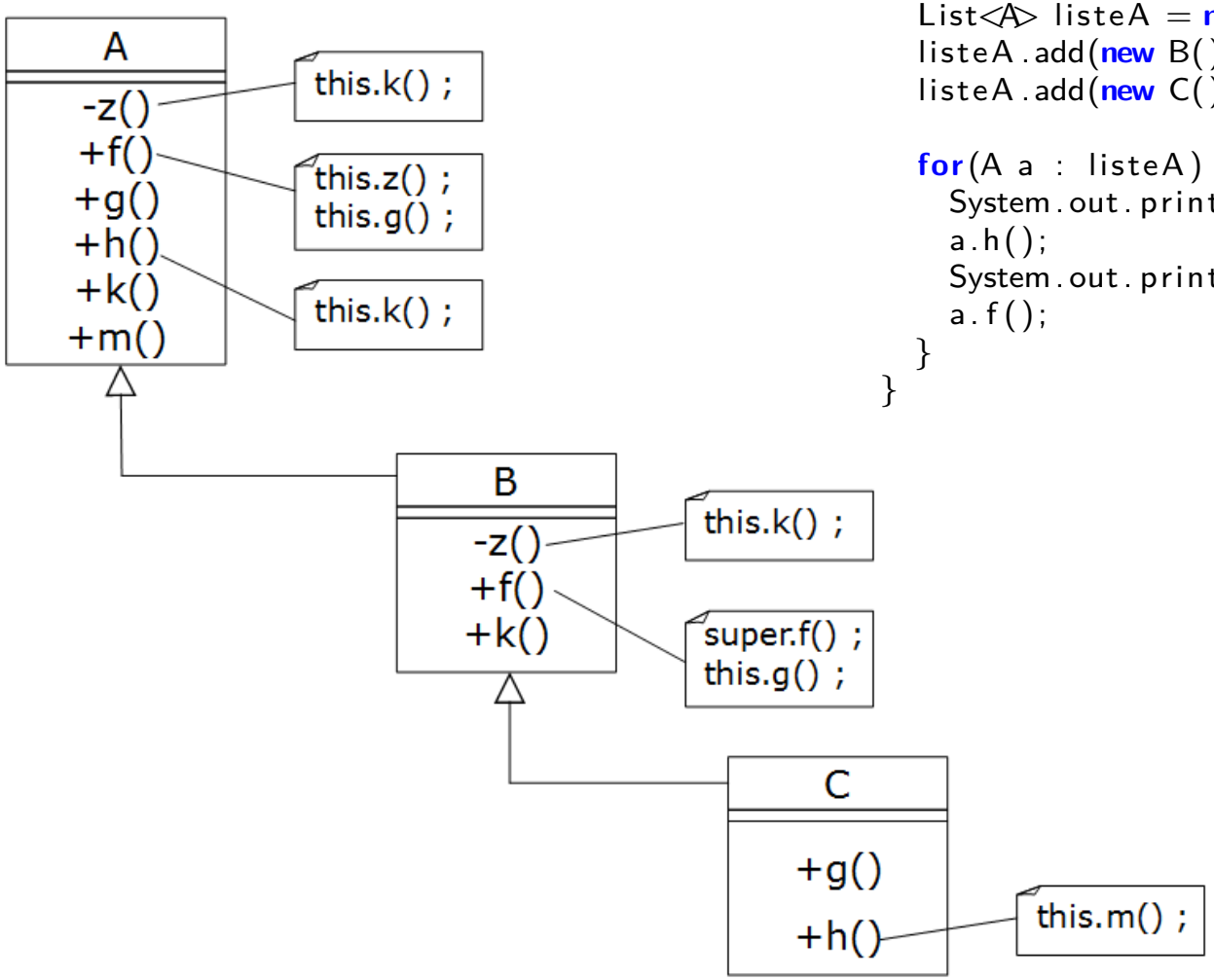
quel code est appelé ?

si `nextSquare` fait référence à un objet

- `YellowSquare` ?
- `BlueSquare` ?

late-binding

lookup




```java
public static void main(String[] args) {
  List<A> listeA = new ArrayList<A>();
  listeA.add(new B());
  listeA.add(new C());

  for(A a : listeA) {
    System.out.println("- appel h() -");
    a.h();
    System.out.println("- appel f() -");
    a.f();
  }
}
```

lookup

- `this` est une référence vers l'objet qui reçoit le message : celui qui invoque la méthode

Recherche ("*lookup*") :

- méthode publique :
  - la recherche commence dans la **classe de l'objet invoquant**
  - si aucune définition de la méthode n'est trouvée, on continue la recherche dans la super-classe
  - et ainsi de suite

  la recherche de la méthode est donc **dynamique**
- si le message porte sur une méthode **privée** de la classe du receveur :
  - on prend la définition de la méthode dans la classe où est réalisée l'envoi de message. Dans ce cas le choix est **statique**.

lookup avec super

- super est une référence vers l'objet qui reçoit le message ("invoquant")

`super == this`

Mais, recherche de méthode (*lookup*) avec super **différente** :

- méthode publique :
  - la recherche commence dans la **super-classe de la classe définissant la méthode** utilisant `super`
  - le processus de chaînage reste ensuite le même à partir de cette classe
- `super` **ne fait pas** commencer la recherche de méthode dans la super-classe de l'objet

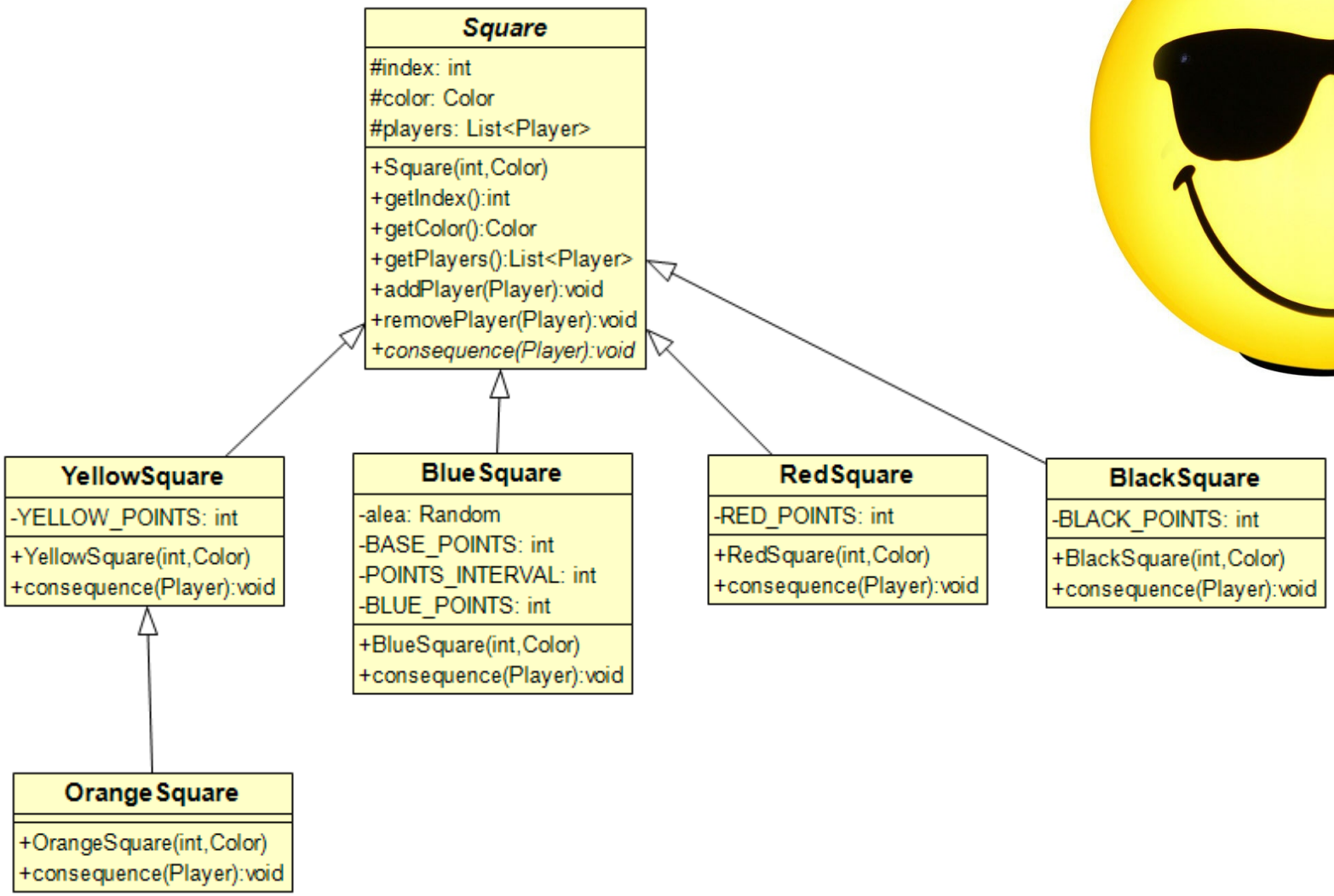
this est dynamique, **super** est statique

évolution

ajout d'une nouvelle couleur de case

si la couleur de la case est :

- *noir, le joueur perd deux vies*


Square
#index: int
#color: Color
#players: List<Player>
+Square(int,Color)
+getIndex():int
+getColor():Color
+getPlayers():List<Player>
+addPlayer(Player):void
+removePlayer(Player):void
+consequence(Player):void
YellowSquare
-YELLOW_POINTS: int
+YellowSquare(int,Color)
+consequence(Player):void
BlueSquare
-alea: Random
-BASE_POINTS: int
-POINTS_INTERVAL: int
-BLUE_POINTS: int
+BlueSquare(int,Color)
+consequence(Player):void
RedSquare
-RED_POINTS: int
+RedSquare(int,Color)
+consequence(Player):void
BlackSquare
-BLACK_POINTS: int
+BlackSquare(int,Color)
+consequence(Player):void
OrangeSquare
+OrangeSquare(int,Color)
+consequence(Player):void

Principe ouvert-fermé

Open Close Principle

> Un code doit être ouvert aux extensions et fermé aux modifications.

il doit être possible d'étendre l'application sans modifier le code existant

```
public abstract class Square {
  ...
  public abstract void consequence(Player player);
}
```

```
public class YellowSquare extends Square {
  ...

  public void consequence(Player player) {
    // points management
    int points = 50;
    if (player.getColor().equals(this.color)) {
      points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);

    // apply the square effect
    player.receiveOneLife();

    // has the player lost all its lives ?
    if (player.getNbLives() <= 0) {
      System.out.println(player +" is ...")
    }
  }
}
```

```
public class BlueSquare extends Square} {
  ...

  public void consequence(Player player) {
    // points management
    int points = 100;
    if (player.getColor().equals(this.color)) {
      points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);

    // apply the square effect
    player.addPoints(alea.nextInt(40)+10);

    // has the player lost all its lives ?
    if (player.getNbLives() <= 0) {
      System.out.println(player +" is ...");
    }
  }
}
```

duplication de code...

\+ similaire dans `BlackSquare`, `RedSquare`

dans le **contrat** de `consequence`

- *ajouter au joueur le nombre points correspondant à la case (points doublés si les couleurs correspondent)*
- *afficher le nombre de points gagnés*
- *appliquer l'effet correspondant à la couleur de la case*
- *indiquer si le joueur a perdu toutes ses vies*

seuls le nombre de points et l'application de l'effet changent d'un type à l'autre

? Comment le prendre en compte et garantir que le contrat sera toujours respecté ?

y compris pas les classes non encore écrites...

il faut séparer ce qui change de ce qui ne varie pas

Comment ?

il faut séparer **ce qui change** de ce qui ne varie pas

Comment ? `BlueSquare`

```
public void consequence(Player player) {
   // points management
   int points = 100;
   if (player.getColor().equals(this.color)) {
      points = 2 * points;
   }
   player.addPoints(points);
   System.out.println(player+" gets "+points);
   // apply the square effect
   player.addPoints(alea.nextInt(40)+10);;
   // has the player lost all its lives ?
   if (player.getNbLives() <= 0) {
      System.out.println(player +" is ...");
   }
}
```

il faut séparer **ce qui change** de ce qui ne varie pas

Comment ? `YellowSquare`

```
public void consequence(Player player) {
    // points management
    int points =  50;
    if (player.getColor().equals(this.color)) {
       points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);
    // apply the square effect
    player.receiveOneLife();
    // has the player lost all its lives ?
    if (player.getNbLives() <= 0) {
       System.out.println(player +" is ...");
    }
}
```

il faut séparer ce qui change de **ce qui ne varie pas**

Comment ? **tous les Square**

```
public void consequence(Player player) {
    // points management
    int points = **************;
    if (player.getColor().equals(this.color)) {
        points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);
    // apply the square effect
    ************************;
    // has the player lost all its lives ?
    if (player.getNbLives() <= 0) {
        System.out.println(player +" is ...");
    }
}
```

il faut séparer ce qui change de **ce qui ne varie pas**

Comment ? **tous les Square**

```
public void consequence(Player player) {
   // points management
   int points = this.getNbPoints();
   if (player.getColor().equals(this.color)) {
      points = 2 * points;
   }
   player.addPoints(points);
   System.out.println(player+" gets "+points);
   // apply the square effect
   **************************;
   // has the player lost all its lives ?
   if (player.getNbLives() <= 0) {
      System.out.println(player +" is ...");
   }
}
```

il faut séparer ce qui change de **ce qui ne varie pas**

Comment ? **tous les Square**

```
public void consequence(Player player) {
    // points management
    int points = this.getNbPoints();
    if (player.getColor().equals(this.color)) {
        points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);
    // apply the square effect
    this.applyEffect(player);
    // has the player lost all its lives ?
    if (player.getNbLives() <= 0) {
        System.out.println(player +" is ...");
    }
}
```

utiliser les « outils » à disposition pour consolider le contrat

```
public abstract class Square {
   ...
   public  void consequence(Player player) {
                        // points management
      int points = this.getNbPoints();
      if (player.getColor().equals(this.color)) {
         points = 2 * points;
      }
      player.addPoints(points);
      System.out.println(player+" gets "+points);
                        // apply the square effect
      this.applyEffect(player);
                        // has the player lost all its lives ?
      if (player.getNbLives() <= 0) {
         System.out.println(player +" is ...");
      }
   }

         abstract int getNbPoints();
            abstract void applyEffect(Playerplayer);
}
```

utiliser les « outils » à disposition pour consolider le contrat

```
public abstract class Square {
   ...
   public  void consequence(Player player) {
                        // points management
      int points = this.getNbPoints();
      if (player.getColor().equals(this.color)) {
         points = 2 * points;
      }
      player.addPoints(points);
      System.out.println(player+" gets "+points);
                        // apply the square effect
      this.applyEffect(player);
                        // has the player lost all its lives ?
      if (player.getNbLives() <= 0) {
         System.out.println(player +" is ...");
      }
   }

    public abstract int getNbPoints();
    protected abstract void applyEffect(Playerplayer);
}
```

utiliser les « outils » à disposition pour consolider le contrat

```
public abstract class Square {
   ...
   public final void consequence(Player player) {
                        // points management
      int points = this.getNbPoints();
      if (player.getColor().equals(this.color)) {
         points = 2 * points;
      }
      player.addPoints(points);
      System.out.println(player+" gets "+points);
                        // apply the square effect
      this.applyEffect(player);
                        // has the player lost all its lives ?
      if (player.getNbLives() <= 0) {
         System.out.println(player +" is ...");
      }
   }

    public abstract int getNbPoints();
    protected abstract void applyEffect(Playerplayer);
}
```

```java
public class YellowSquare extends Square {
  ...

  public int  getNbPoints() {
    return 50;
  }

  protected void applyAffect(Player player) {
    player.receiveOneLife();
  }
}
```

```java
public class BlueSquare extends Square {
    ...
    public int  getNbPoints() {
      return 100;
    }

    protected void applyEffect(Player player) {
      player.addPoints(alea.nextInt(40)+10);
    }
}
```

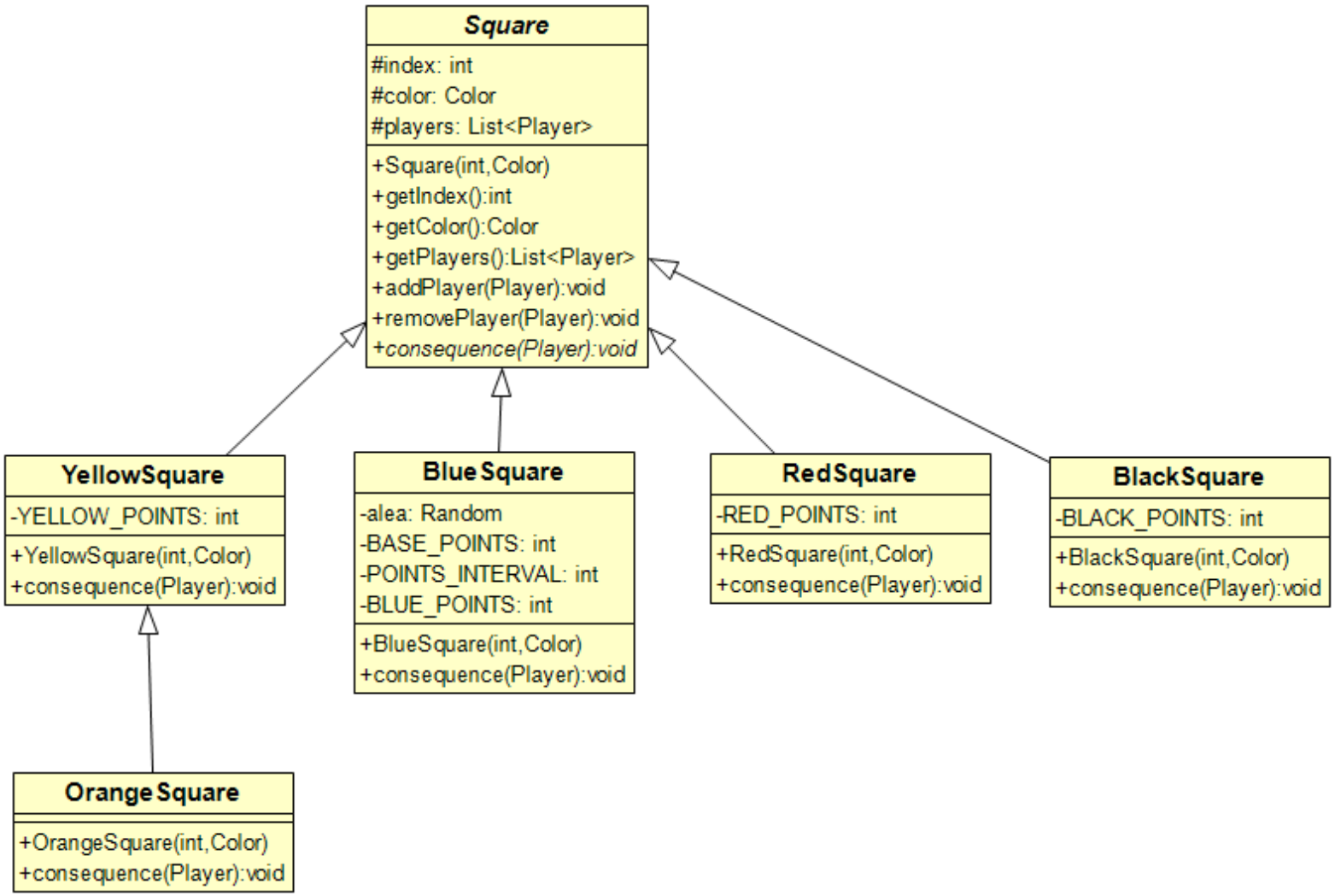
```java
public class OrangeSquare extends YellowSquare {
    ...
    public int  getNbPoints() {
      return 2 * super.getNbPoints();
    }

    protected void applyEffect(Player player) {
      super.applyEffect(player);
      super.applyEffect(player);
    }
}
```

tests

tout code doit être testé

un code non testé n'a pas de valeur

pour rappel


Square
#index: int
#color: Color
#players: List<Player>
+Square(int,Color)
+getIndex():int
+getColor():Color
+getPlayers():List<Player>
+addPlayer(Player):void
+removePlayer(Player):void
+consequence(Player):void
YellowSquare
-YELLOW_POINTS: int
+YellowSquare(int,Color)
+consequence(Player):void
BlueSquare
-alea: Random
-BASE_POINTS: int
-POINTS_INTERVAL: int
-BLUE_POINTS: int
+BlueSquare(int,Color)
+consequence(Player):void
RedSquare
-RED_POINTS: int
+RedSquare(int,Color)
+consequence(Player):void
BlackSquare
-BLACK_POINTS: int
+BlackSquare(int,Color)
+consequence(Player):void
OrangeSquare
+OrangeSquare(int,Color)
+consequence(Player):void

Spécification :

- *l'effet appliqué par la conséquence d'une case jaune est que le joueur gagne une vie*

```
import static org.junit.Assert.*;
import org.junit.Test;

public class YellowSquareTest {
  @Test
  public void consequenceAppliesEffectTest() {
    YellowSquare yellow = new YellowSquare(1);
    Player p = new Player(Color.GREEN);
    int lives = p.getNbLives();
    yellow.consequence(p);
    assertEquals(1, p.getNbLives() - lives);
  }
}
```

indépendances des tests

Tester les méthodes `setSquare` et `getSquare` de `Player`.

Spécification :

> *Après l'exécution de la méthode `setSquare`, la méthode `getSquare` doit fournir la case affectée.*

indépendances des tests

Tester les méthodes `setSquare` et `getSquare` de `Player`.

Spécification :

> *Après l'exécution de la méthode `setSquare`, la méthode `getSquare` doit fournir la case affectée.*

quelle classe de `Square` pour le test ?

indépendances des tests

Tester les méthodes `setSquare` et `getSquare` de `Player`.

Spécification :

> *Après l'exécution de la méthode `setSquare`, la méthode `getSquare` doit fournir la case affectée.*

quelle classe de `Square` pour le test ?

utilisation de « **Mock** »

tests et classe abstraite

- Comment tester les méthodes de `Square` ?
  Comment tester `consequence` ? (effet appliqué et points ajoutés)

tests et classe abstraite

- Comment tester les méthodes de `Square` ?
  Comment tester `consequence` ? (effet appliqué et points ajoutés)
- dans les tests de `YellowSquare`, `BlueSquare`, etc. ?
  Pb : répétitions des tests...

tests et classe abstraite

- Comment tester les méthodes de `Square` ?
  Comment tester `consequence` ? (effet appliqué et points ajoutés)
- dans les tests de `YellowSquare`, `BlueSquare`, etc. ?
  Pb : répétitions des tests...
- dans `SquareTest`
  Pb : création des instances ?

tests et classe abstraite

- Comment tester les méthodes de `Square` ?
  Comment tester `consequence` ? (effet appliqué et points ajoutés)
- dans les tests de `YellowSquare`, `BlueSquare`, etc. ?
  Pb : répétitions des tests...
- dans `SquareTest`
  Pb : création des instances ?

utilisation de « **Mock** »

tests et héritage

`YellowSquare` doit passer avec succès les tests de `Square` ?

tests et héritage

`YellowSquare` doit passer avec succès les tests de `Square` ?

- hiérarchie de classes de tests

tests et héritage

`YellowSquare` doit passer avec succès les tests de `Square` ?

- hiérarchie de classes de tests
  `YellowSquareTest` hérite de `SquareTest`

tests et héritage

`YellowSquare` doit passer avec succès les tests de `Square` ?

- hiérarchie de classes de tests
  `YellowSquareTest` hérite de `SquareTest`
  `YellowSquareTest` en succès si passe ses propres tests + les tests hérités de `SquareTest`

tests et héritage

`YellowSquare` doit passer avec succès les tests de `Square` ?

- hiérarchie de classes de tests
  `YellowSquareTest` hérite de `SquareTest`
  `YellowSquareTest` en succès si passe ses propres tests + les tests hérités de `SquareTest`
- comment gérer les instances ?

tests et héritage

`YellowSquare` doit passer avec succès les tests de `Square` ?

- hiérarchie de classes de tests
  `YellowSquareTest` hérite de `SquareTest`
  `YellowSquareTest` en succès si passe ses propres tests + les tests hérités de `SquareTest`
- comment gérer les instances ?
  méthode de fabrique (« *factory method* ») + `@Before`

exceptions

- `removePlayer` de `Square` lève une exception `UnknownPlayerException` si le joueur à supprimer n'est pas sur la case

création d'exception

```
package squaregame;
public class UnknownPlayerException extends Exception {
  public UnknownPlayerException(String msg) {
    super(msg);
  }
}
```

removePlayer

- `throws`
- `throw new ...`

capture (ou pas)

- `try ... catch`
- ou `throws` ...
- `RuntimeException` ?

tests et exceptions

```
@Test(expected=UnkownPlayerException.class)
```