BLG 468E Recitation

Object Oriented Modelling & Design

Designing Monopoly Game

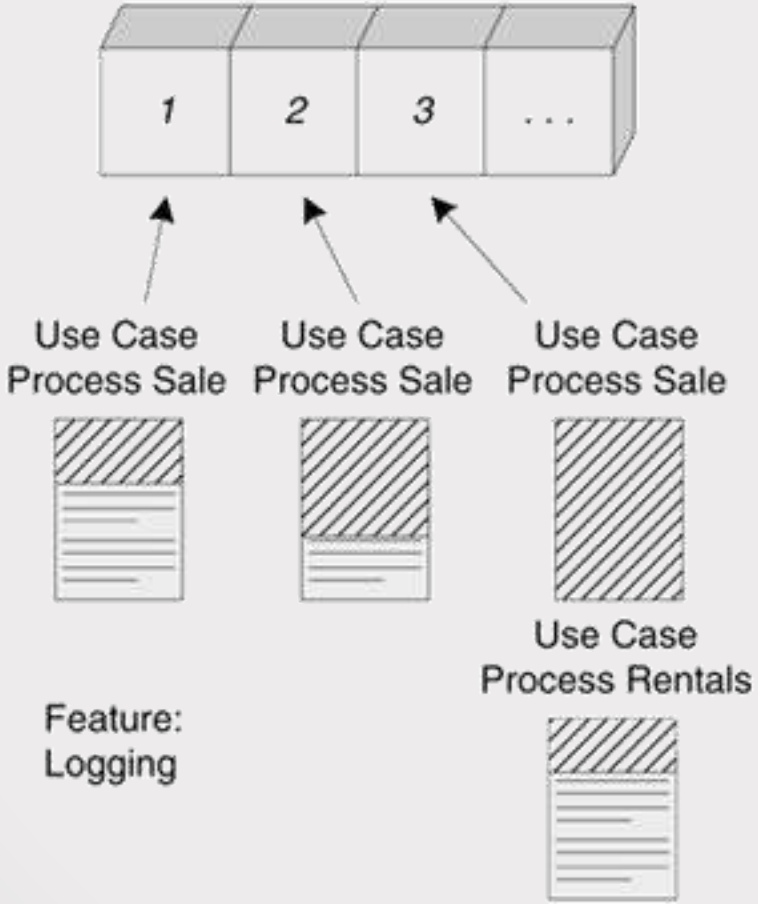
Mehmet Tahir SANDIKKAYA

March 21st, 2016

Use Case

- Two to eight players can play.
- A game is played as a series of rounds. During a round, each player takes one turn. In each turn, a player advances his piece clockwise around the board a number of squares equal to the sum of the number rolled on two six-sided dice.
- Play the game for only 20 rounds.
- After the dice are rolled, the name of the player and the roll are displayed. When the player moves and lands on a square, the name of the player and the name of the square that the player landed on are displayed.
- In iteration-1 there is no money, no winner or loser, no properties to buy or rent to pay, and no special squares of any kind.
- Each square has a name. Every player begins the game with their piece located on the square named "Go." The square names will be Go, Square 1, … Square 39.
- Run the game as a simulation requiring no user input, other than the number of players.

How to split the use cases?

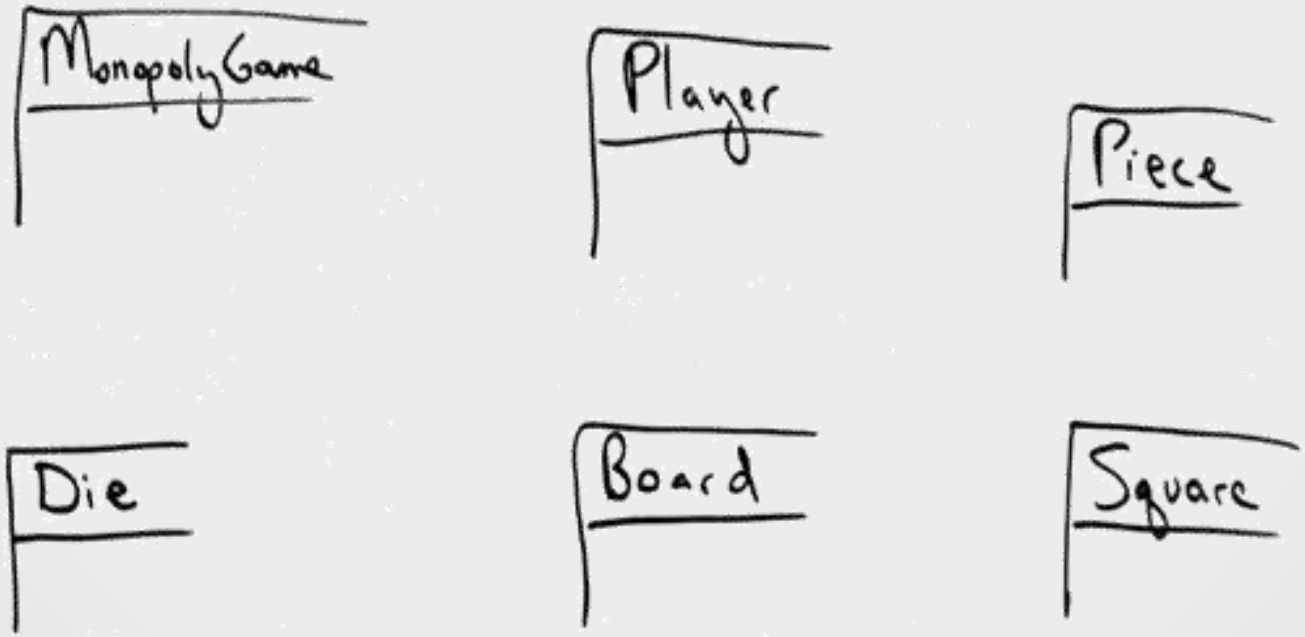


A use case or feature is often too complex to complete in one short iteration.

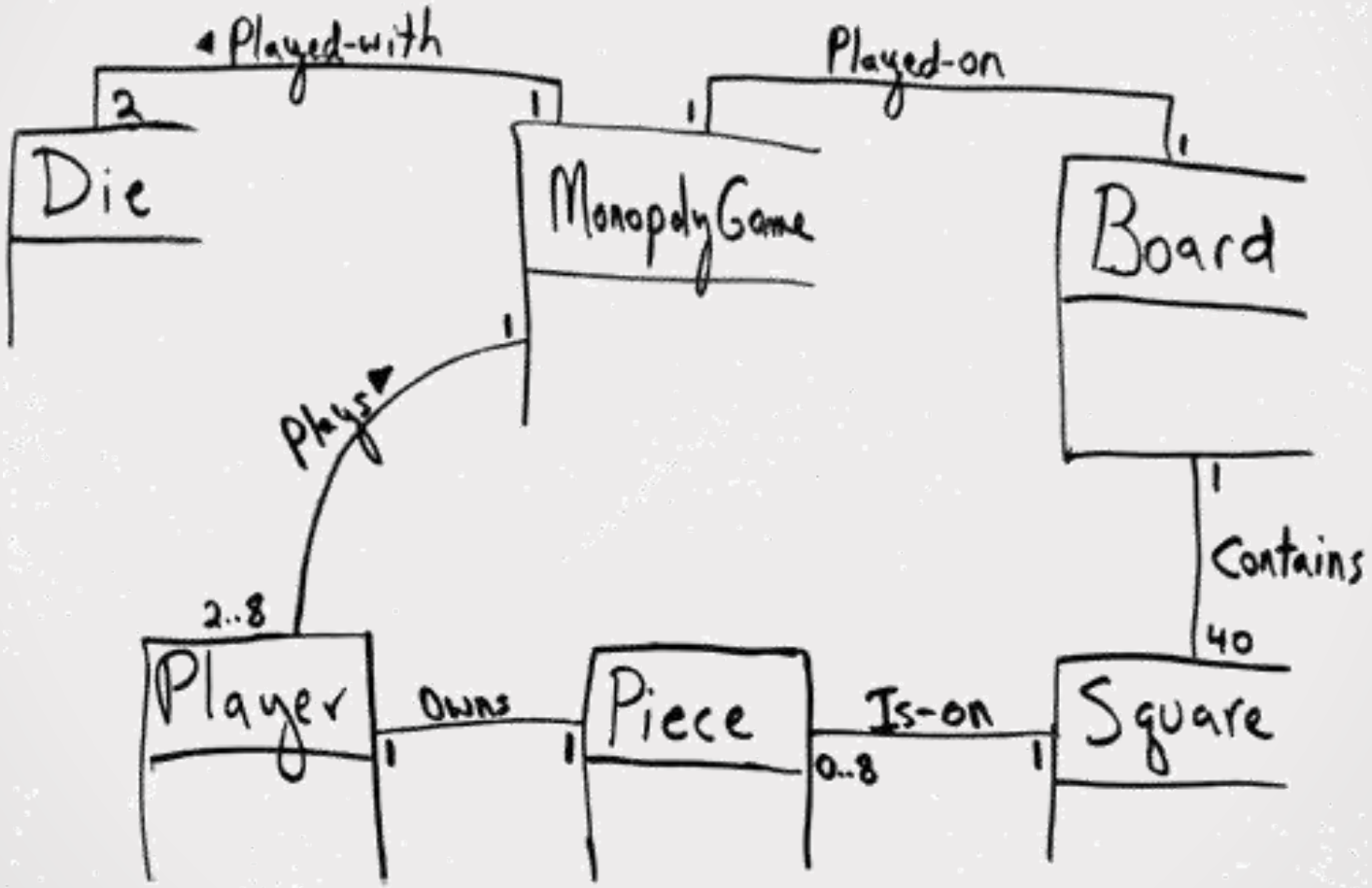
Therefore, different parts or scenarios must be allocated to different iterations.

The Domain Model

- A **game** is played as a series of rounds. During a round, each **player** takes one turn. In each turn, a player advances his **piece** clockwise around the **board** a number of **squares** equal to the sum of the number rolled on two six-sided **dice**.



Try to find associations in The Domain Model

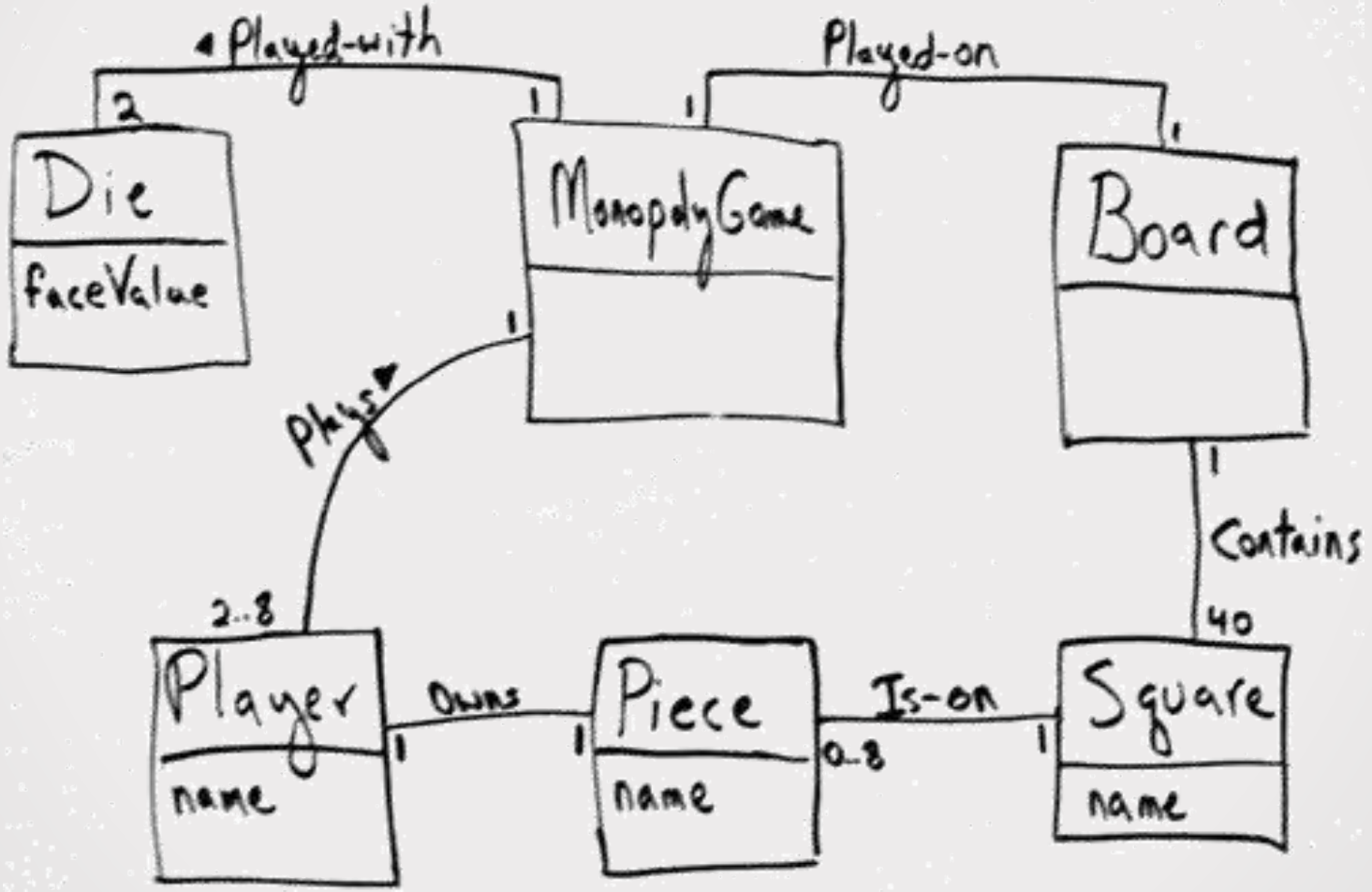


Determine the attributes of The Domain Classes

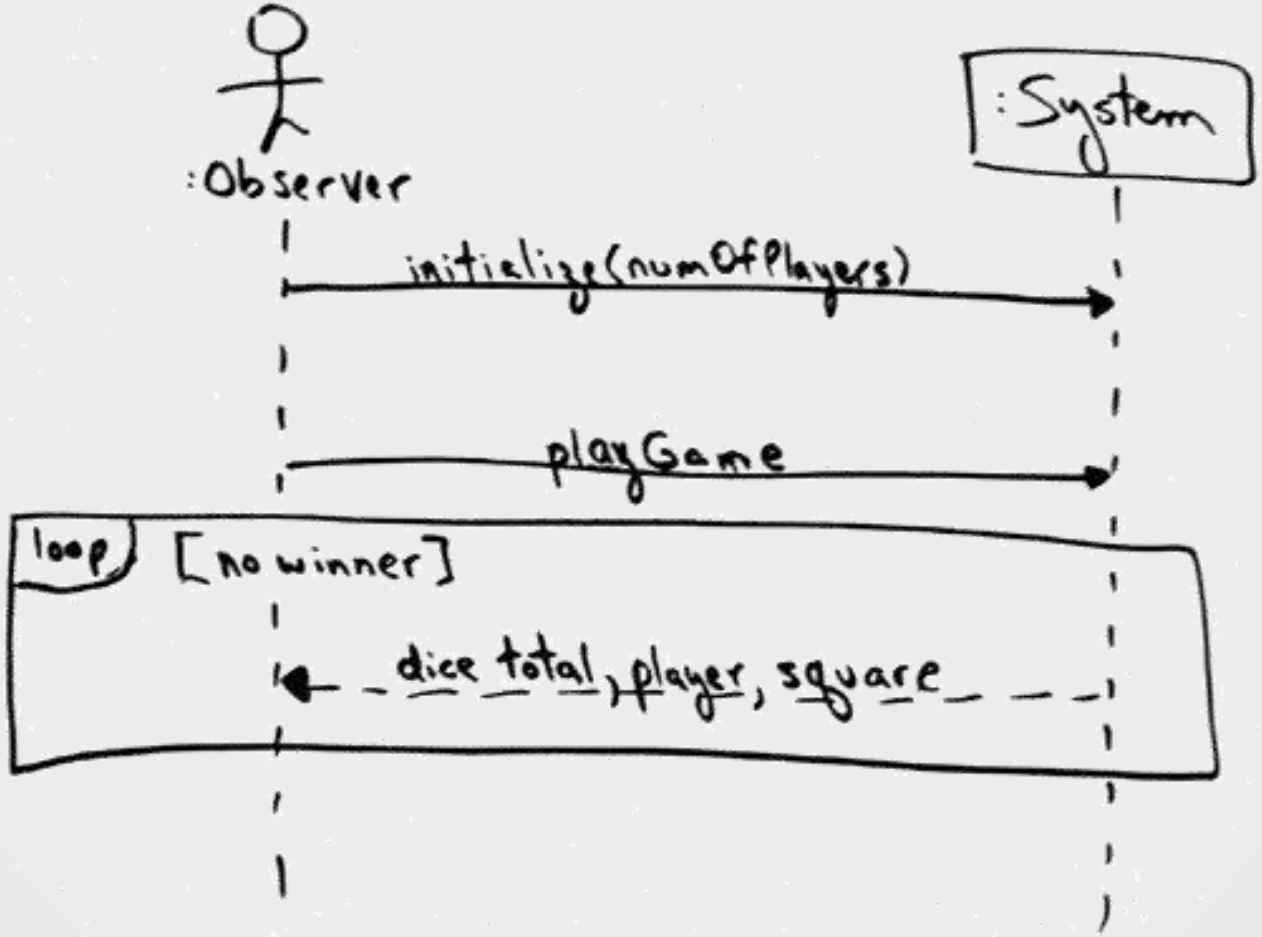
- Use case may be helpful.

- …a player advances his piece clockwise around the board a number of squares equal to the **sum of the number rolled on two six-sided dice**.
- After the dice are rolled, the **name of the player** and the roll are displayed. When the player moves and lands on a square, the name of the player and the **name of the square** that the player landed on are displayed.

Determine the attributes of The Domain Classes



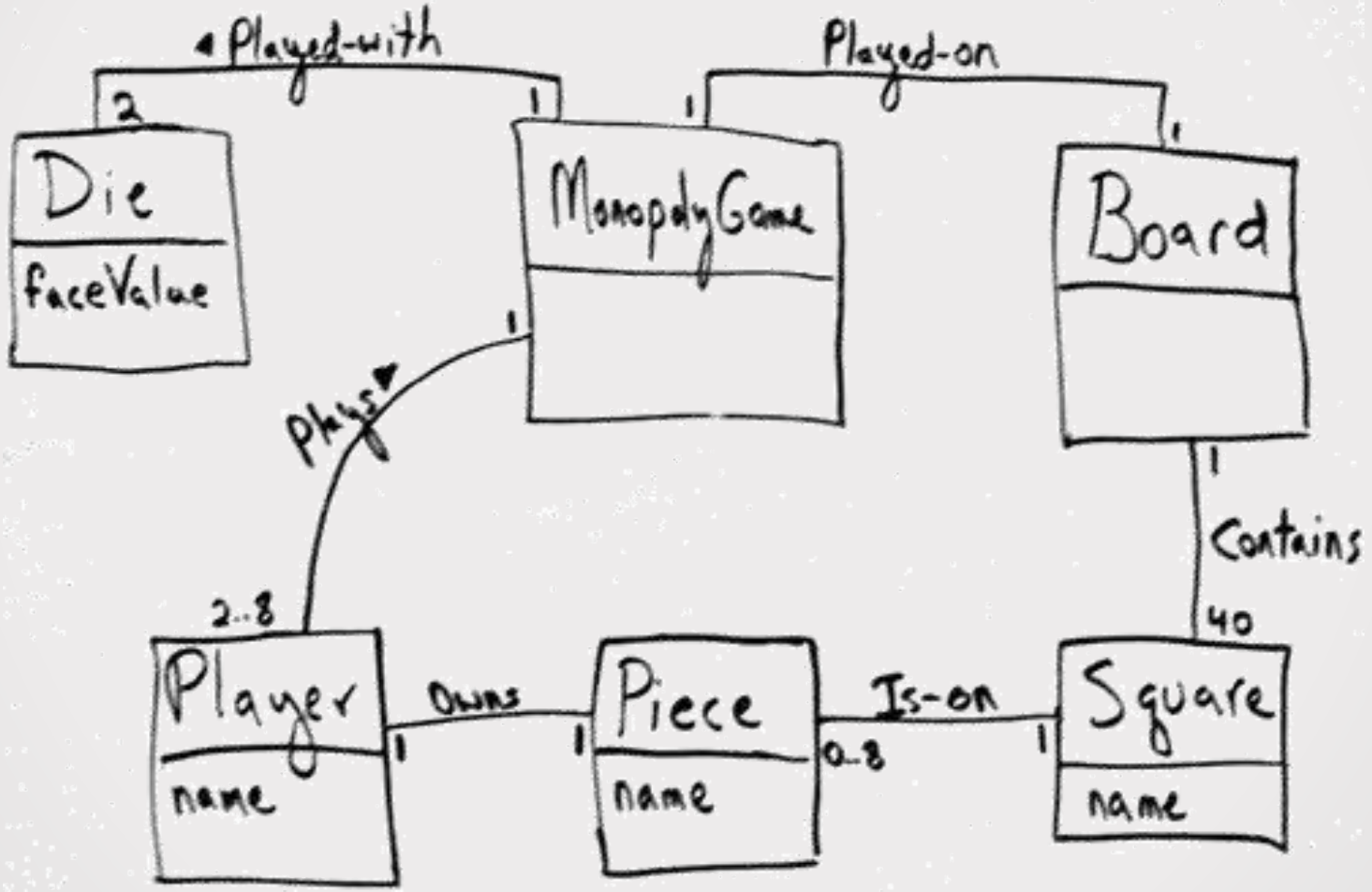
Generate a Sequence Diagram



Adding a contract

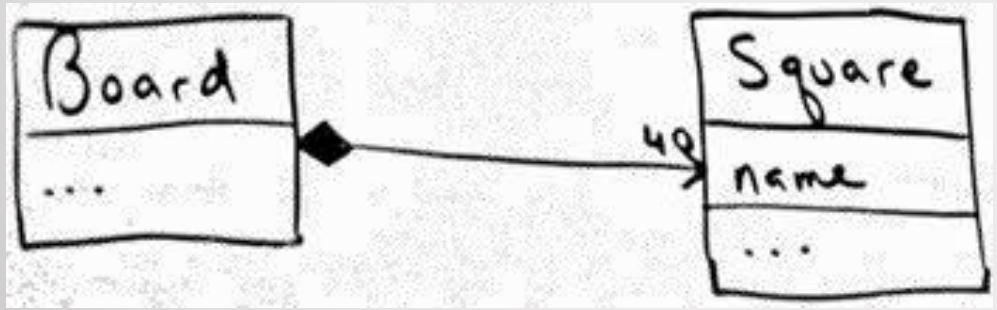
- Profit of adding a contract?

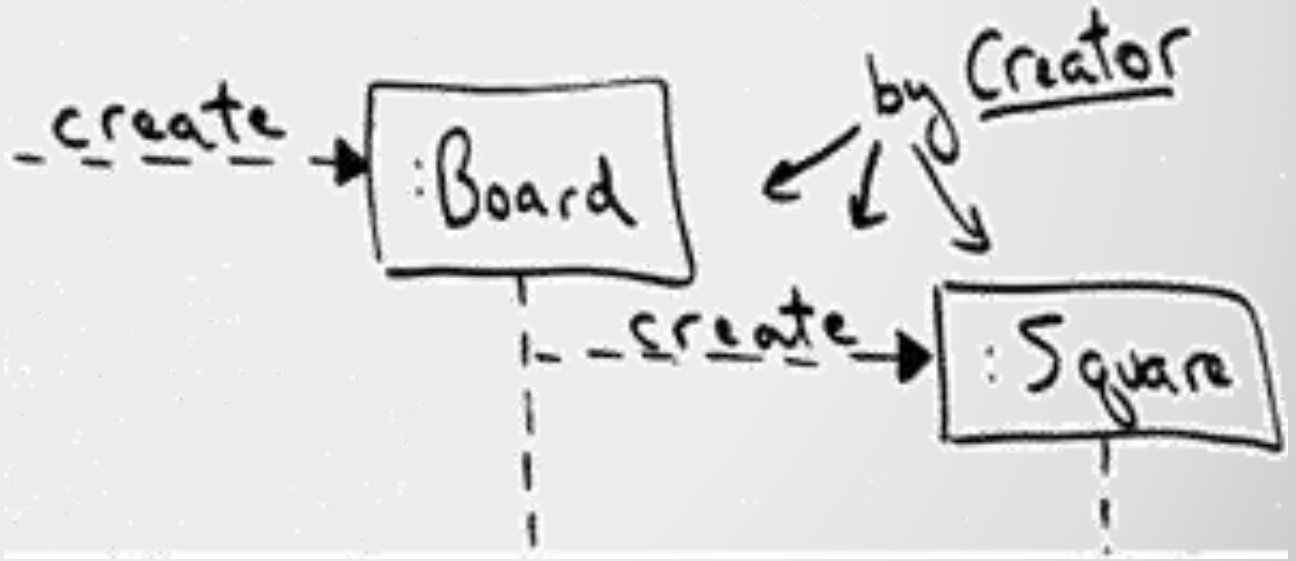
Who creates the Square?



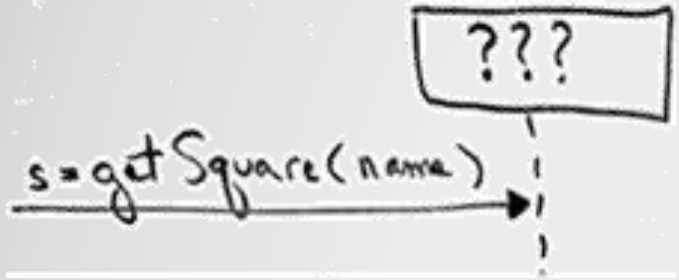
Creator pattern

- Who creates an A?
- Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):
  - B "contains" or compositely aggregates A.
  - B records A.
  - B closely uses A.
  - B has the initializing data for A.

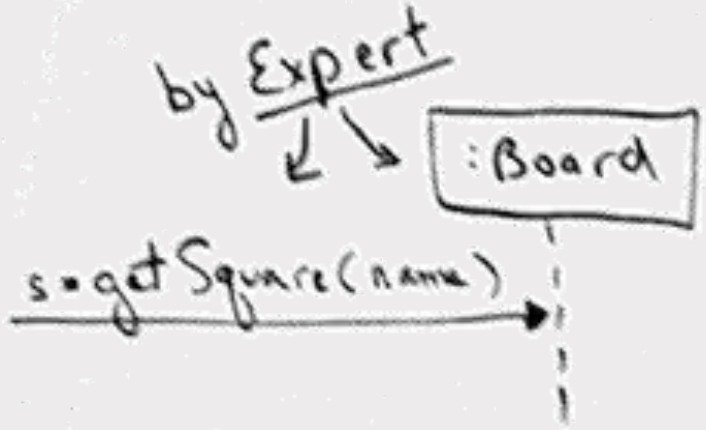


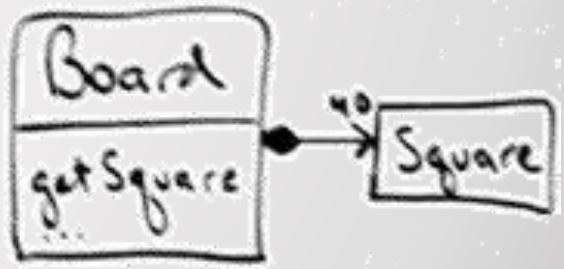


Who knows about a Square object? (Information) **Expert** pattern



- What is a basic principle by which to assign responsibilities to objects?
- Assign a responsibility to the class that has the information needed to fulfill it.

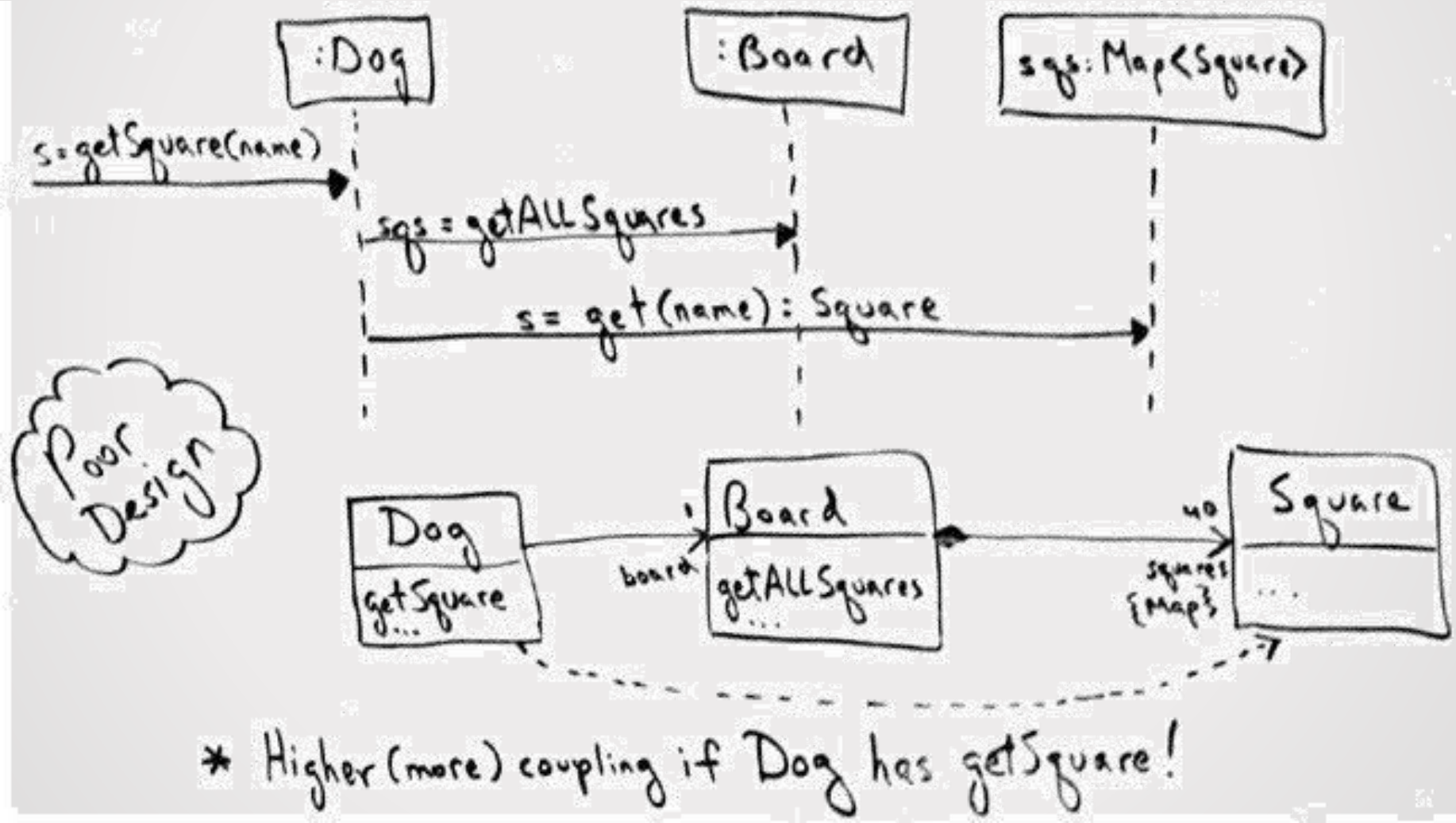




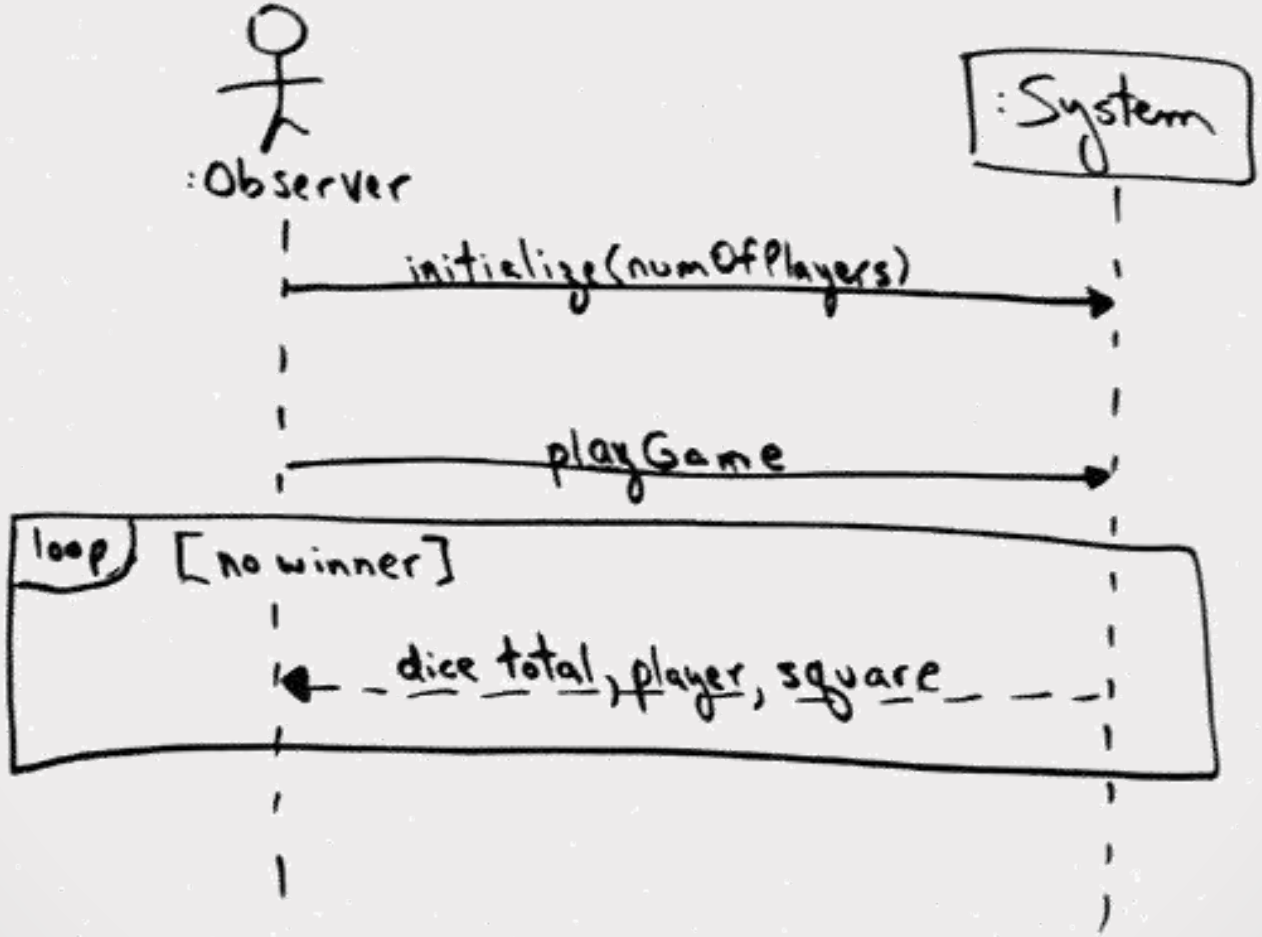
Why Board over Dog?
Low Coupling pattern

- How to reduce the impact of change?
- Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

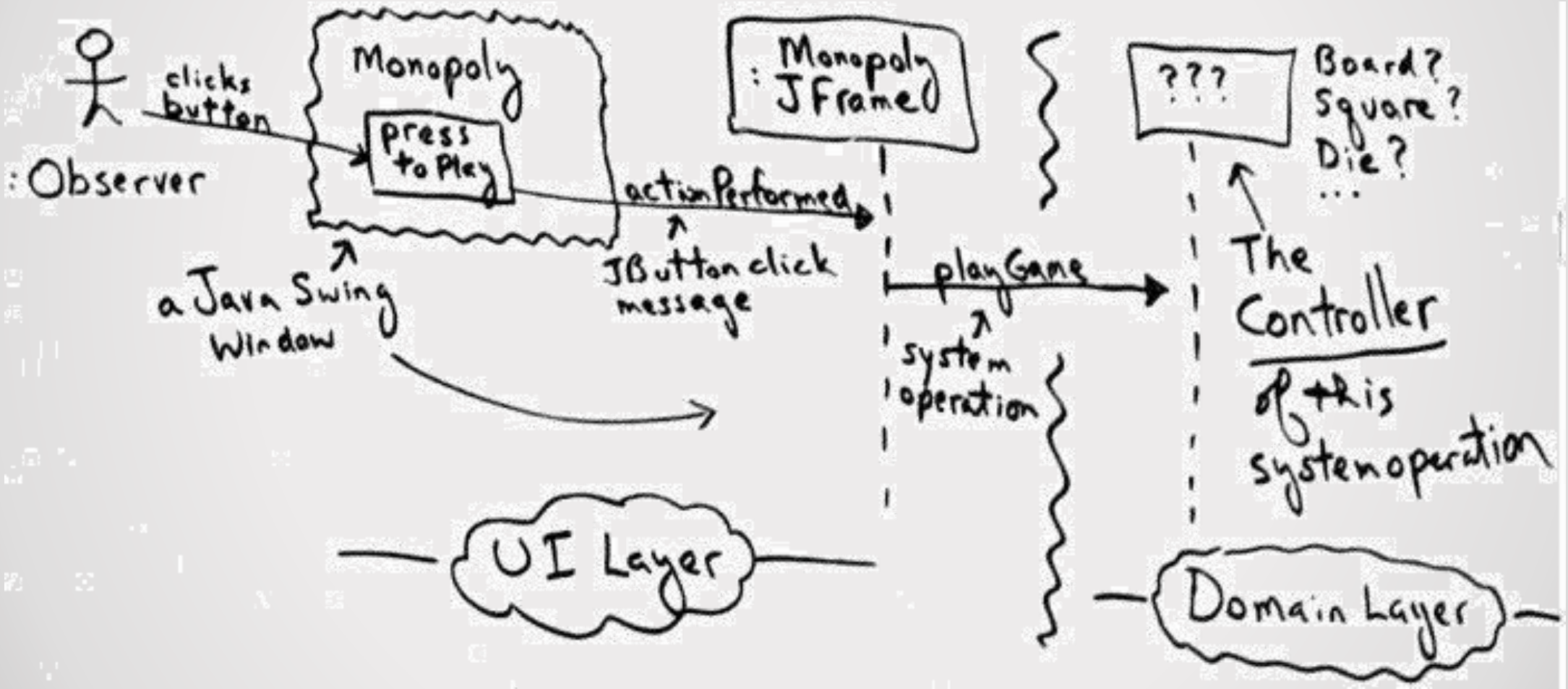
Why Board over Dog?
Low Coupling pattern



Who controls *playGame* operation? **Controller** pattern



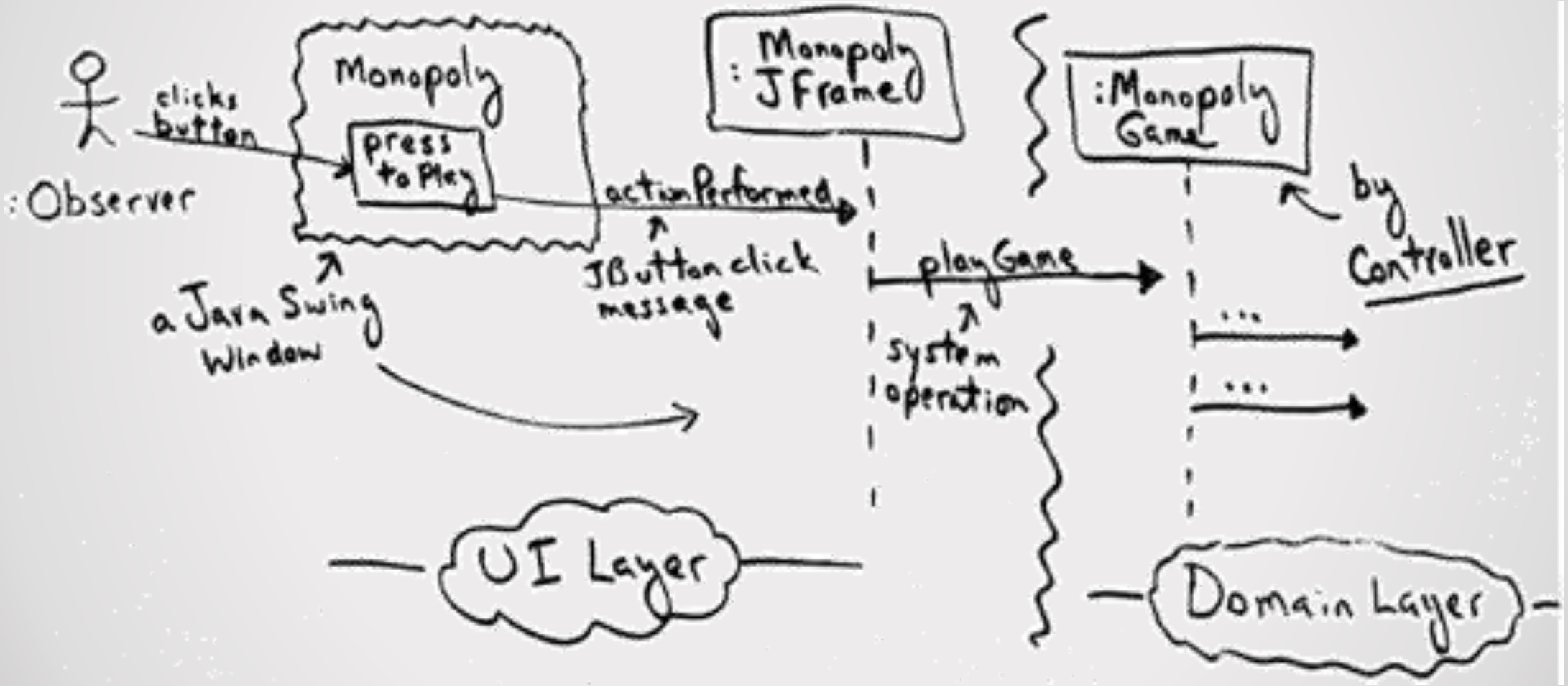
Controller pattern



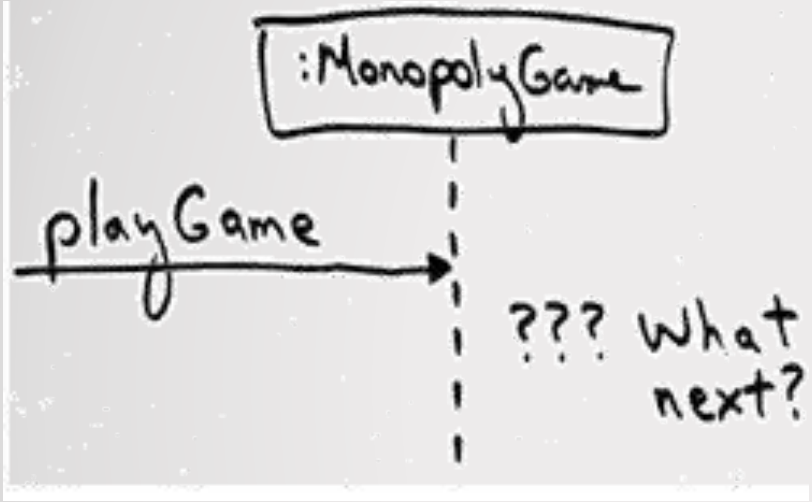
Controller pattern

- What first object beyond the UI layer receives and coordinates ("controls") a system operation?
- Assign the responsibility to an object representing one of these choices:
  - Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a facade controller).
  - Represents a use case scenario within which the system operation occurs (a use case or session controller).

Controller pattern

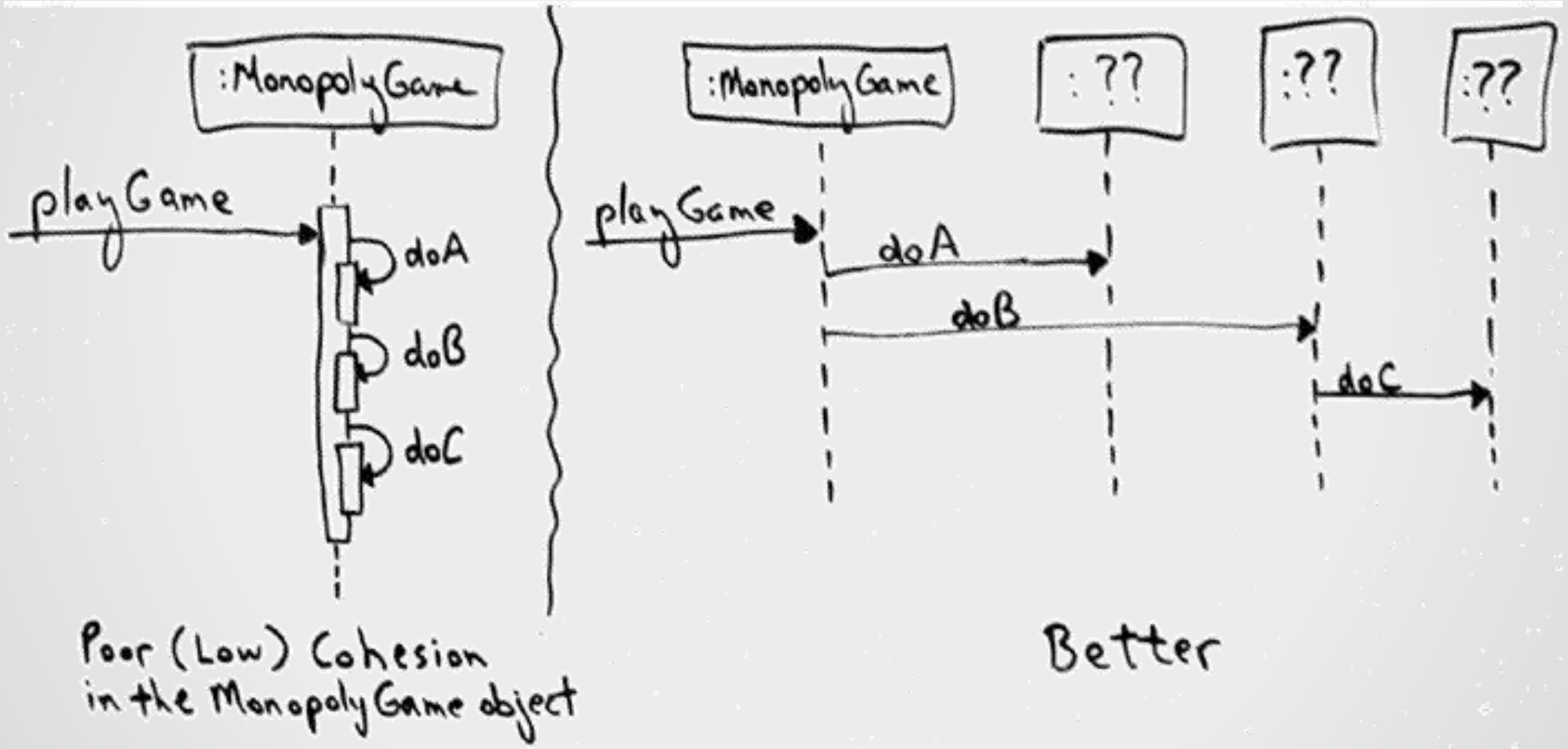


What's after the *playGame*? **High Cohesion** pattern



- How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
- Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.

High Cohesion pattern



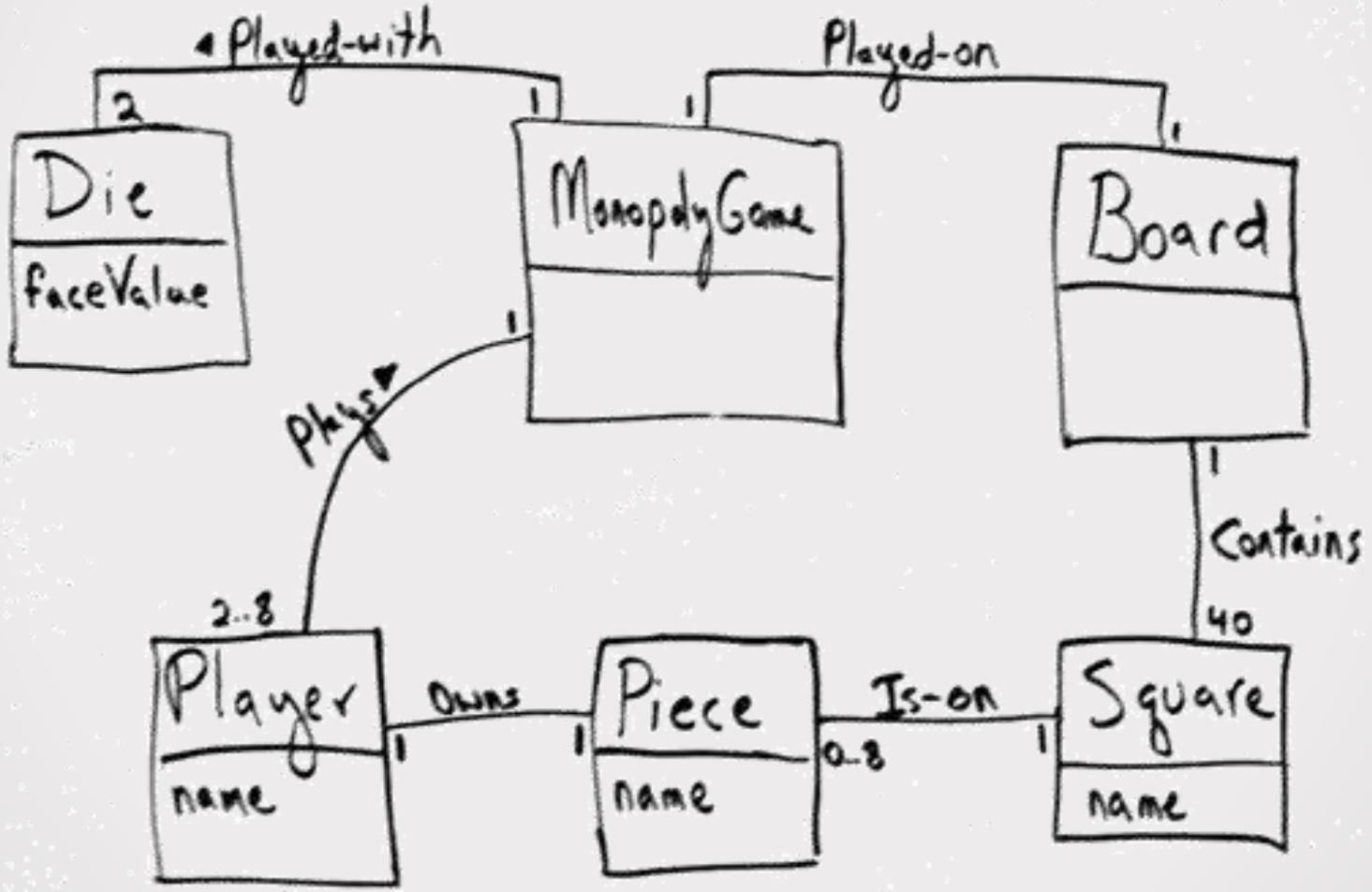
Low Coupling vs. High Cohesion

- How you decide in the real world?

Realization of the use case

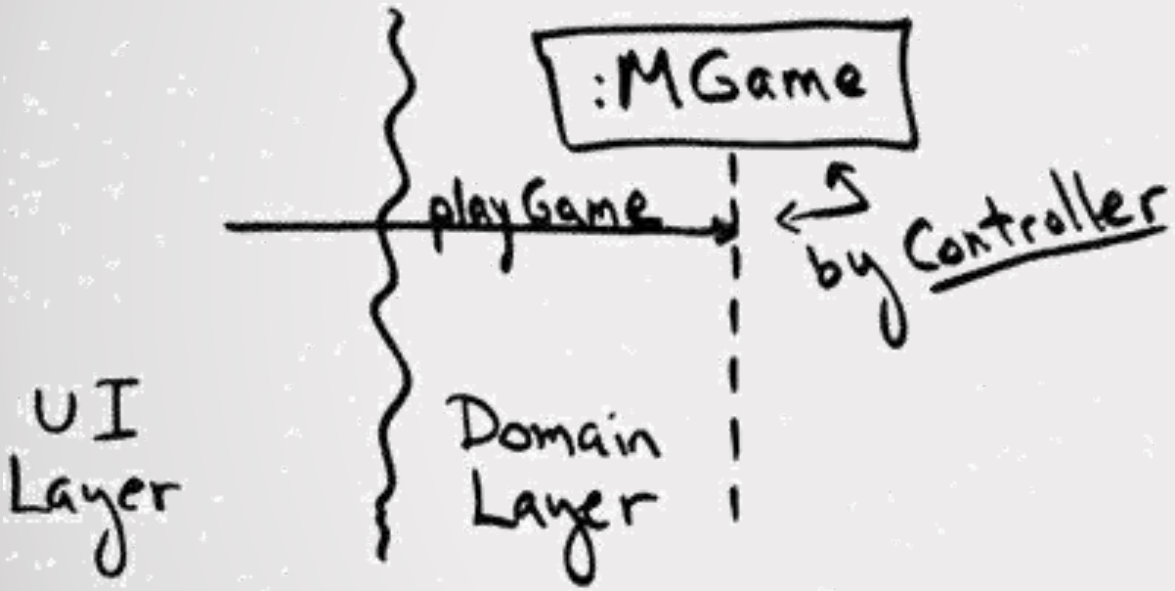
- Two to eight players can play.
- A game is played as a series of rounds. During a round, each player takes one turn. In each turn, a player advances his piece clockwise around the board a number of squares equal to the sum of the number rolled on two six-sided dice.
- Play the game for only 20 rounds.
- After the dice are rolled, the name of the player and the roll are displayed. When the player moves and lands on a square, the name of the player and the name of the square that the player landed on are displayed.
- In iteration-1 there is no money, no winner or loser, no properties to buy or rent to pay, and no special squares of any kind.
- Each square has a name. Every player begins the game with their piece located on the square named "Go." The square names will be Go, Square 1, … Square 39.
- Run the game as a simulation requiring no user input, other than the number of players.

Realization of the use case



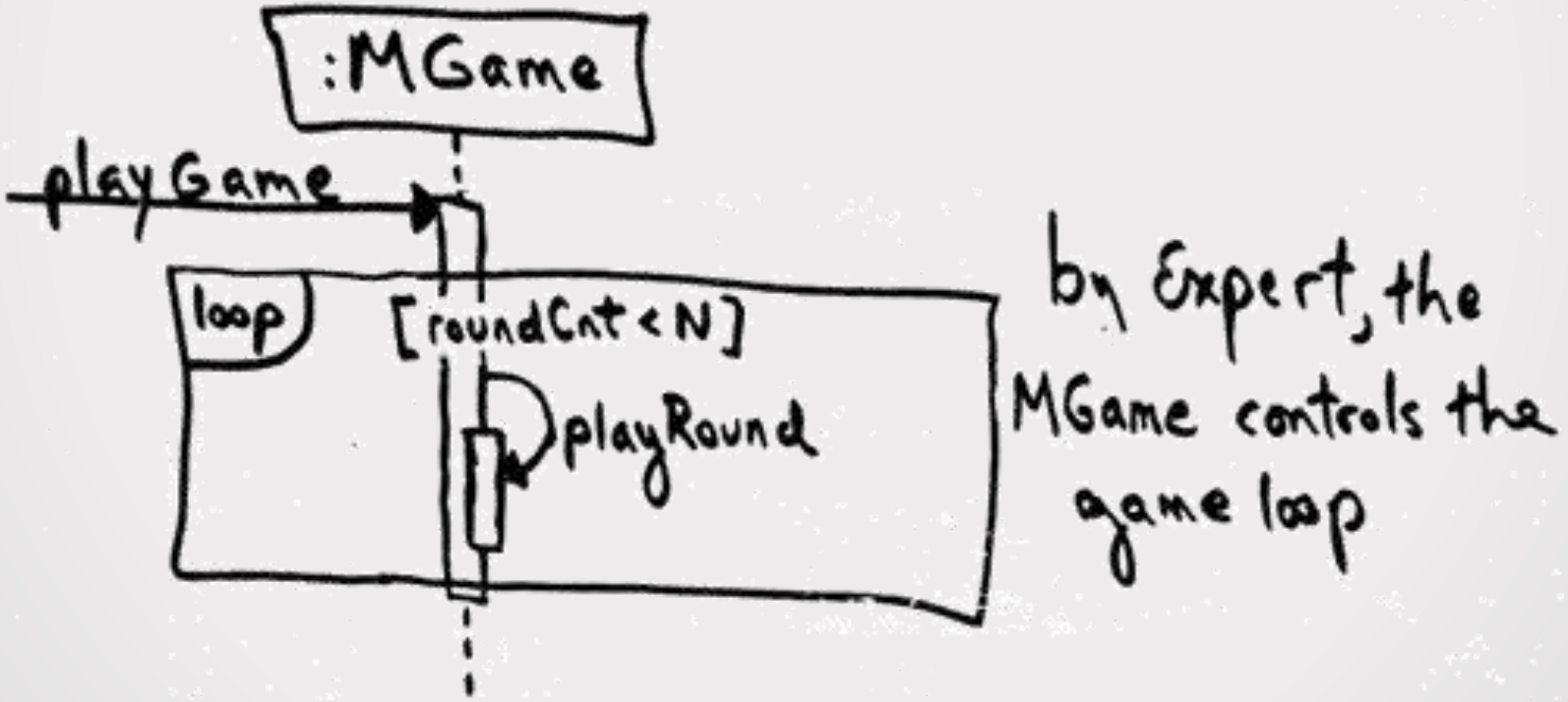
Realization of the use case

- Find the controller class in the domain layer.



Realization of the use case

- Who is Responsible for Controlling the Game Loop?
- Use **Expert** pattern.



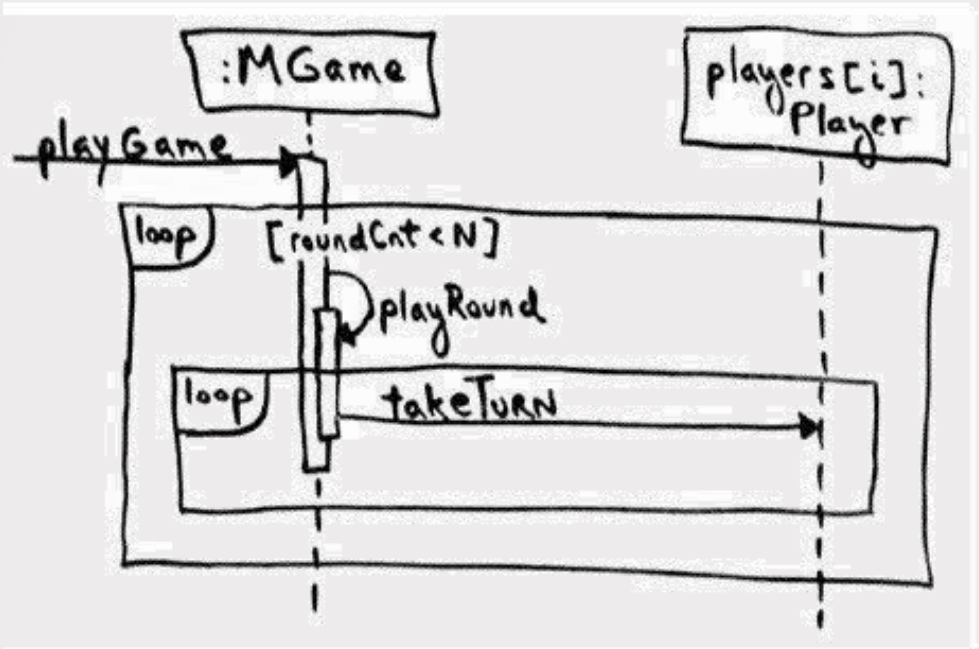
Realization of the use case Who takes a turn?

- current location of the player (to know the starting point of a move)
- a Piece knows its Square and a Player knows its Piece. Therefore, a **Player** software object could know its location by LRG.

- the two Die objects (to roll them and calculate their total)
- **MonopolyGame** is a candidate since we think of the dice as being part of the game.

- all the squares, the square organization (to be able to move to the correct new square)
- By LRG, **Board** is a good candidate.

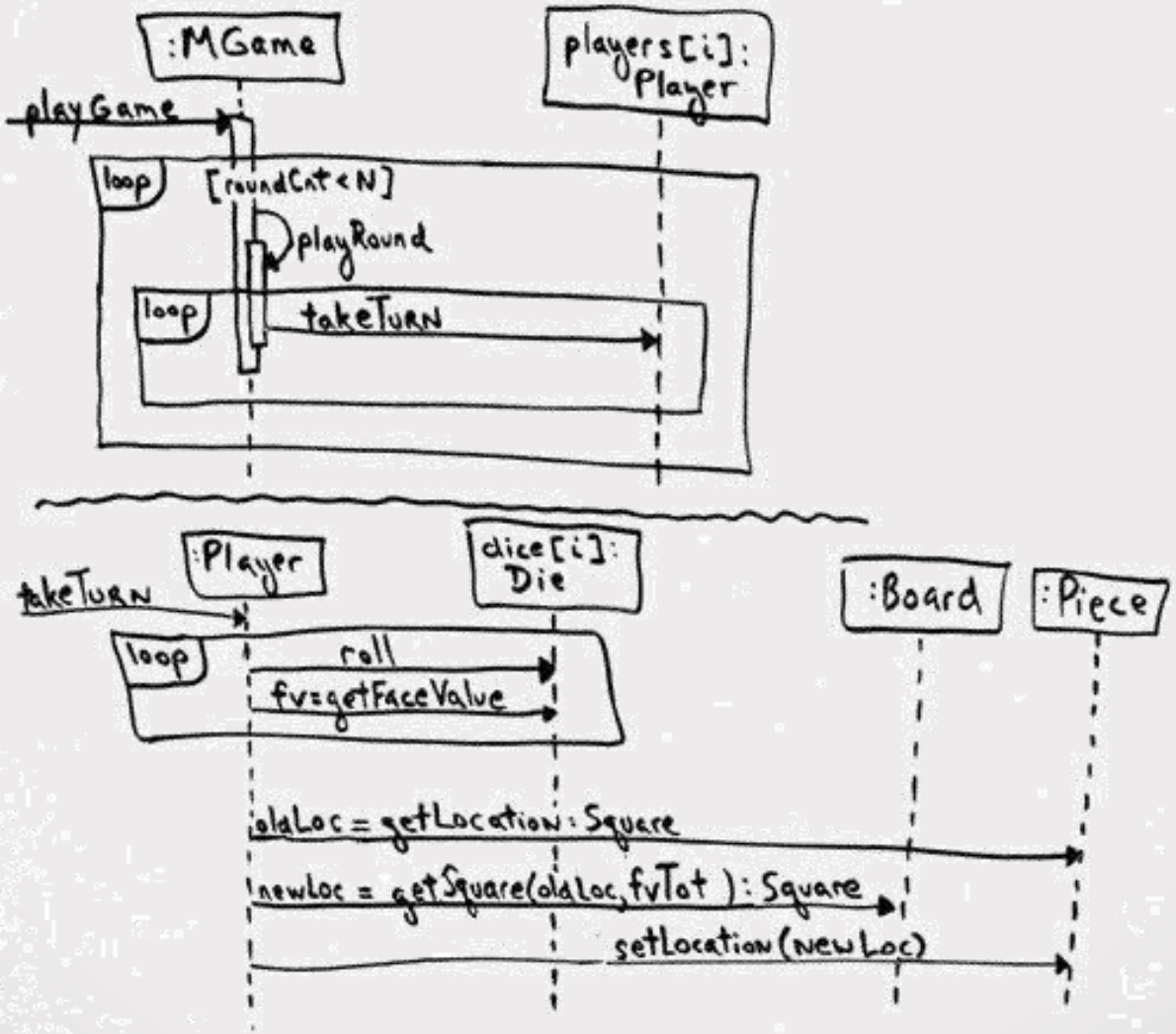
Realization of the use case Who takes a turn?



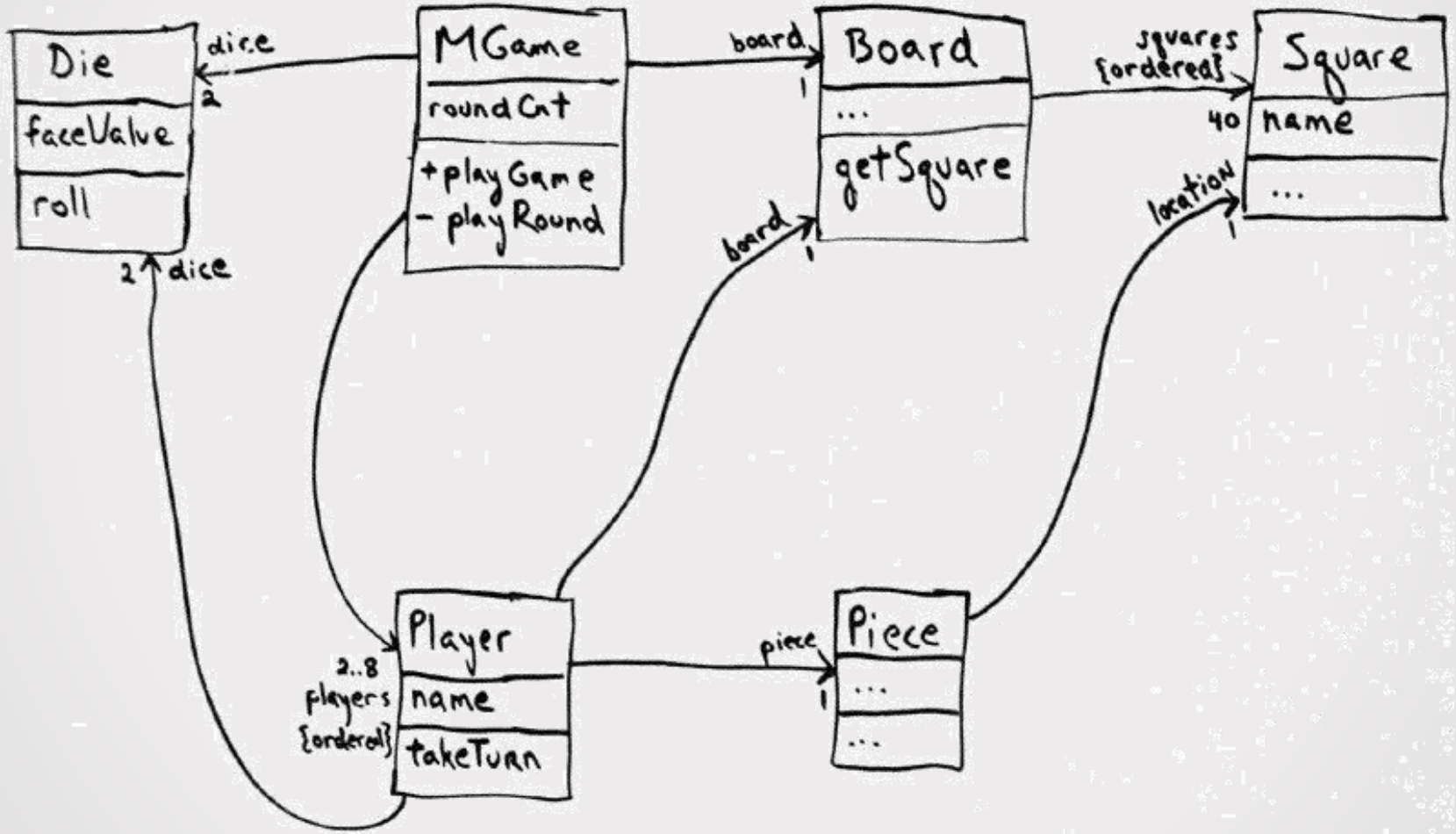
Realization of the use case How to take a turn?

- Taking a turn means:
- calculating a random number total between 2 and 12
- calculating the new square location
- moving the player's piece from an old location to a new square location

Realization of the use case How to take a turn?



Realization of the use case How to take a turn?

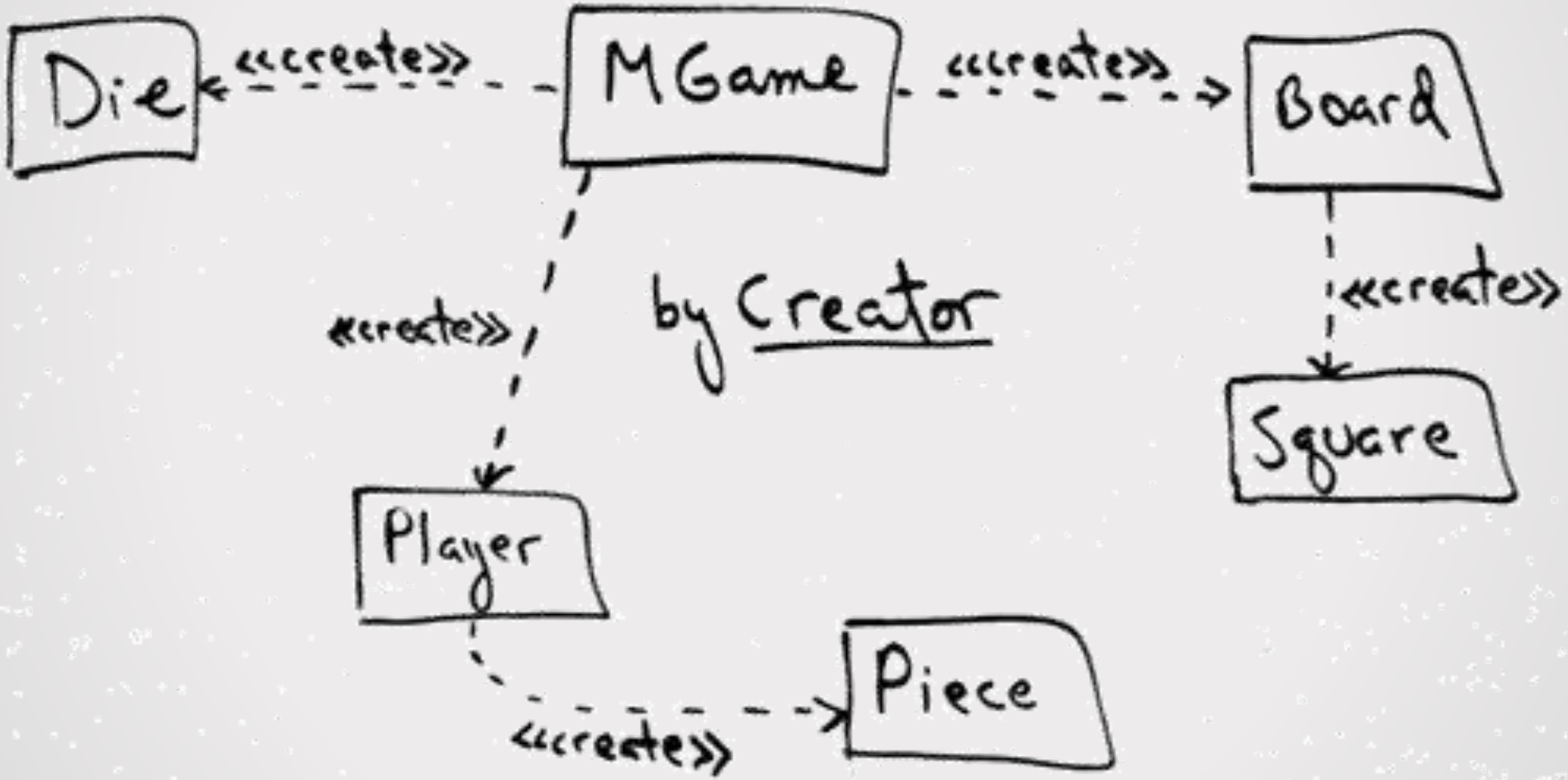


Realization of the use case Command-Query Separation

```
// style #1;
// used in the official solution

public void roll()
{
   faceValue = // random num generation
}


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

```
// style #2;
// why is this poor?

public int roll()
{
   faceValue = // random num generation
   return faceValue;
}
```

Realization of the use case Creation order



Realization of the use case Source Code, Square Class

```
public class Square
{
   private String name;
   private Square nextSquare;
   private int index;

   public Square( String name, int index )
   {
      this.name = name;
      this.index = index;
   }

   public void setNextSquare( Square s )
   {
      nextSquare = s;
   }

  public Square getNextSquare(  )
    {
       return nextSquare;
    }

    public String getName(  )
    {
       return name;
    }

  public int getIndex()
  {
    return index;
  }
}
```

Realization of the use case Source Code, Piece Class

```
public class Piece
{

 private Square location;

 public Piece(Square location)
  {
    this.location = location;
  }

 public Square getLocation()
  {
    return location;
  }

  public void setLocation(Square location)
  {
    this.location = location;
  }
}
```

Realization of the use case Source Code, Die Class

```
public class Die
{
   public static final int MAX  = 6;
   private int           faceValue;

   public Die(  )
   {
      roll(  );
   }

   public void roll(  )
   {
      faceValue=(int)((Math.random()*MAX)+1);
   }
```

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

Realization of the use case Source Code, Board Class

```
public class Board{

  private static final int SIZE    = 40;
  private List             squares = new ArrayList(SIZE);

  public Board()
  {
    buildSquares();
    linkSquares();
  }

  public Square getSquare(Square start, int distance)
  {
    int endIndex = (start.getIndex() + distance) % SIZE;
    return (Square) squares.get(endIndex);
  }
```

Realization of the use case Source Code, Board Class

```
public Square getStartSquare()
{
  return (Square) squares.get(0);
}

private void buildSquares()
{
  for (int i = 1; i <= SIZE; i++)
  {
    build(i);
  }
}
```

```
private void build(int i)
 {
   Square s = new Square("Square " + i, i - 1);
   squares.add(s);
 }
```

Realization of the use case Source Code, Board Class

```
  private void linkSquares(){
    for (int i = 0; i < (SIZE - 1); i++){
      link(i);
    }
    Square first = (Square) squares.get(0);
    Square last = (Square) squares.get(SIZE - 1);
    last.setNextSquare(first);
  }

  private void link(int i){
    Square current = (Square) squares.get(i);
    Square next = (Square) squares.get(i + 1);
    current.setNextSquare(next);
  }
}
```

Realization of the use case Source Code, Player Class

```
public class Player
{
  private String name;
  private Piece  piece;
  private Board  board;
  private Die[]  dice;

  public Player(String name, Die[] dice,
Board board)
  {
    this.name = name;
    this.dice = dice;
    this.board = board;
    piece = new
Piece(board.getStartSquare());
  }
```

```
 public void takeTurn()
  {
         // roll dice
    int rollTotal = 0;
    for (int i = 0; i < dice.length; i++)
    {
      dice[i].roll();
      rollTotal += dice[i].getFaceValue();
    }

    Square newLoc =
board.getSquare(piece.getLocation(), rollTotal);
    piece.setLocation(newLoc);

  }
```

Realization of the use case Source Code, Player Class

```
  public Square getLocation()
  {
    return piece.getLocation();
  }

  public String getName()
  {
    return name;
  }
}
```

Realization of the use case Source Code, MGame Class

```
public class MonopolyGame
{
   private static final int ROUNDS_TOTAL  = 20;
   private static final int PLAYERS_TOTAL = 2;
   private List players = new ArrayList( PLAYERS_TOTAL );
   private Board   board  = new Board(  );
   private Die[]   dice   = { new Die(), new Die() };
 public MonopolyGame(  )
   {
      Player p;
      p = new Player( "Horse", dice, board );
      players.add( p );
      p = new Player( "Car", dice, board  );
      players.add( p );
   }
```

Realization of the use case Source Code, MGame Class

```
public void playGame(  )
{
   for (int i = 0; i < ROUNDS_TOTAL; i++ )
   {
      playRound();
   }
}

public List getPlayers(  )
{
   return players;
}
```

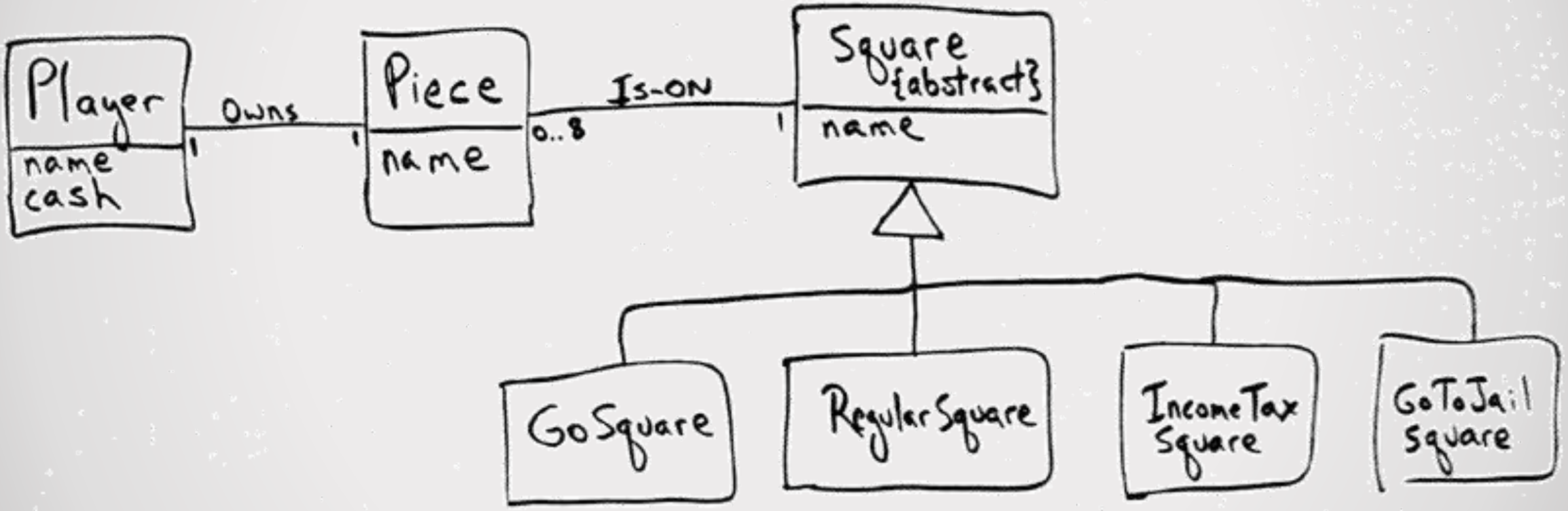
```
 private void playRound(  )
   {
      for ( Iterator iter = players.iterator(
); iter.hasNext(  ); )
      {
         Player player = (Player) iter.next();
         player.takeTurn();
      }
   }
}
```

Realization of the use case Refactor to minimize code reuse

```
public class Player
{
  // …

public void takeTurn()
{
   // the refactored helper method
   int rollTotal = rollDice();

   Square newLoc =
board.getSquare(piece.getLocation(),
rollTotal);
   piece.setLocation(newLoc);
}
```

```
private int rollDice()
{
   int rollTotal = 0;
   for (int i = 0; i < dice.length; i++)
   {
      dice[i].roll();
      rollTotal += dice[i].getFaceValue();
   }
   return rollTotal;
}

} // end of class
```

Elaboration iteration



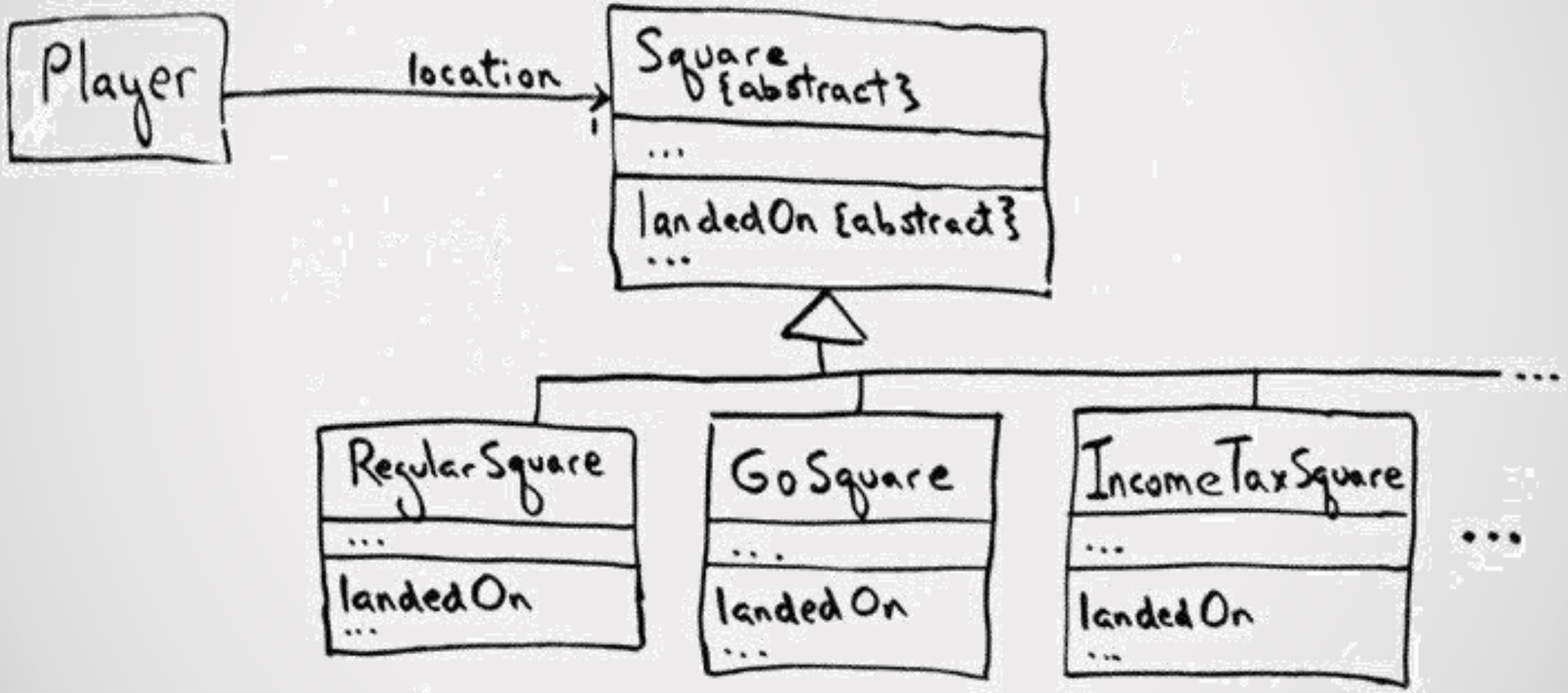
Elaboration iteration

- Each player receives $1500 at the beginning of the game. Consider the game to have an unlimited amount of money.
- When a player lands on the Go square, the player receives $200.
- When a player lands on the Go-To-Jail square, they move to the Jail square.
- However, unlike the complete rules, they get out easily. On their next turn, they simply roll and move as indicated by the roll total.
- When a player lands on the Income-Tax square, the player pays the minimum of $200 or 10% of their worth.

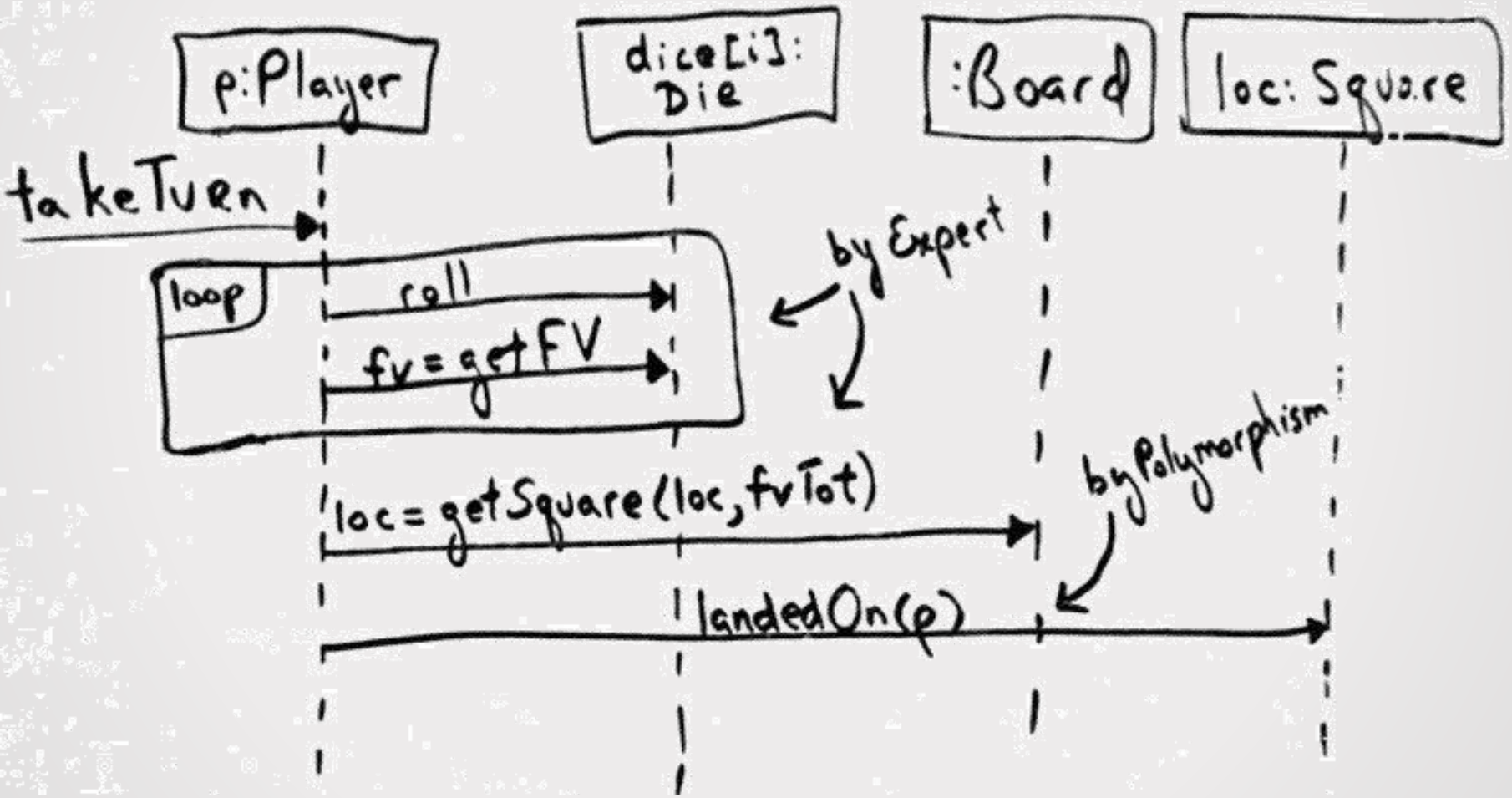
How to Design for Different Square Actions? –**Polymorphism** pattern

- How handle alternatives based on type? How to create pluggable software components?
- When related alternatives or behaviors vary by type (class), assign responsibility for the behavior using polymorphic operations to the types for which the behavior varies.

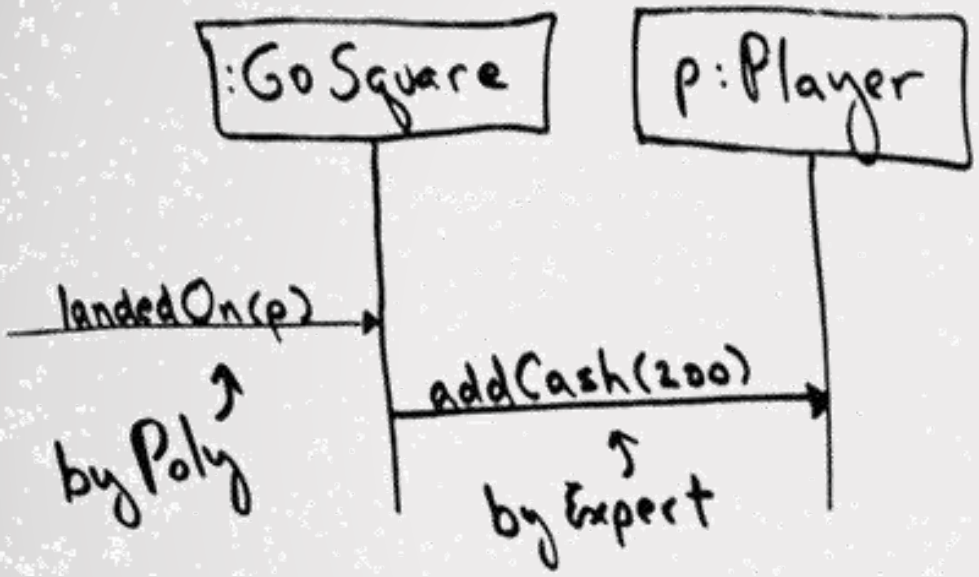
How to Design for Different Square Actions? –**Polymorphism** pattern

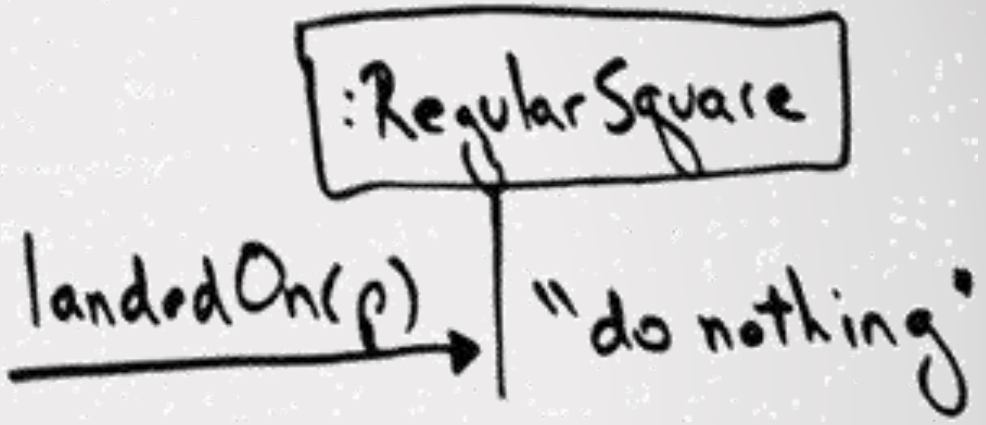


How to Design for Different Square Actions? –**Polymorphism** pattern

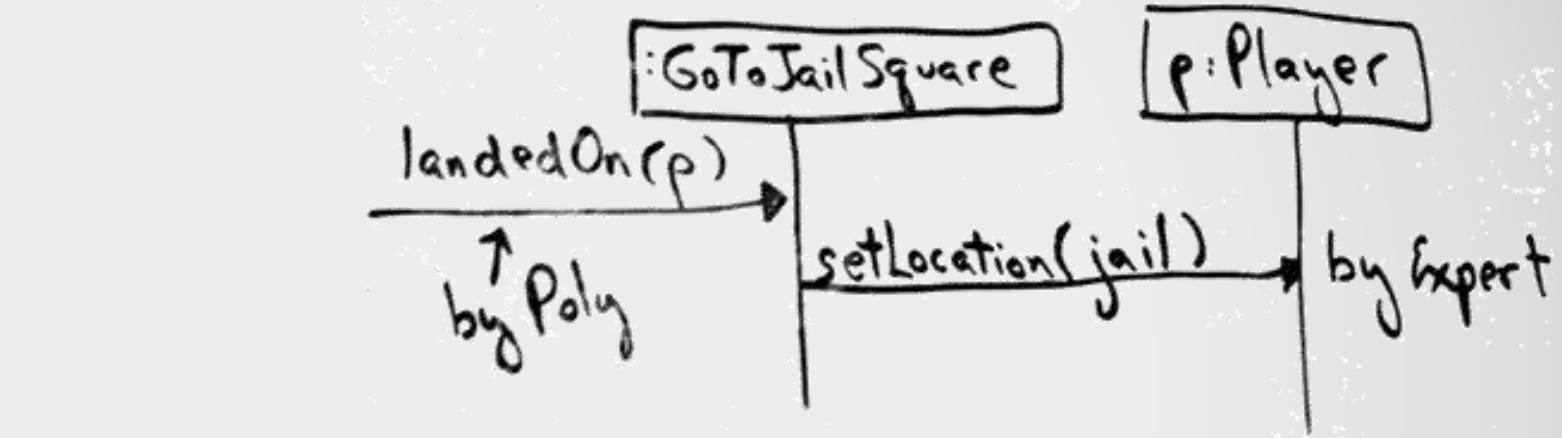


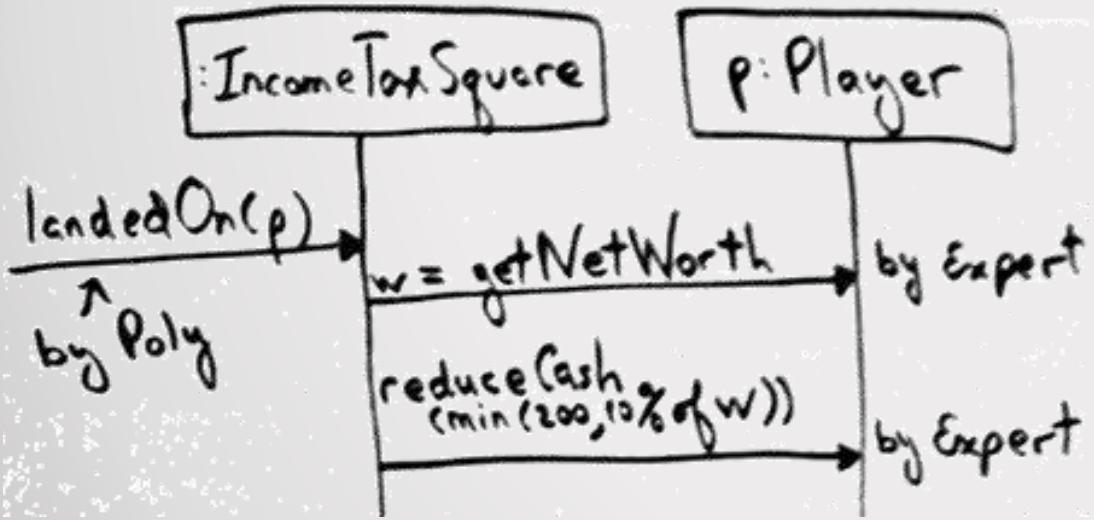
How to Design for Different Square Actions? –**Polymorphism** pattern





How to Design for Different Square Actions? –**Polymorphism** pattern





How to handle the dice better? **Pure fabrication** pattern

- What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?
- Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept something made up, to support high cohesion, low coupling, and reuse.

How to handle the dice better?
Pure fabrication pattern