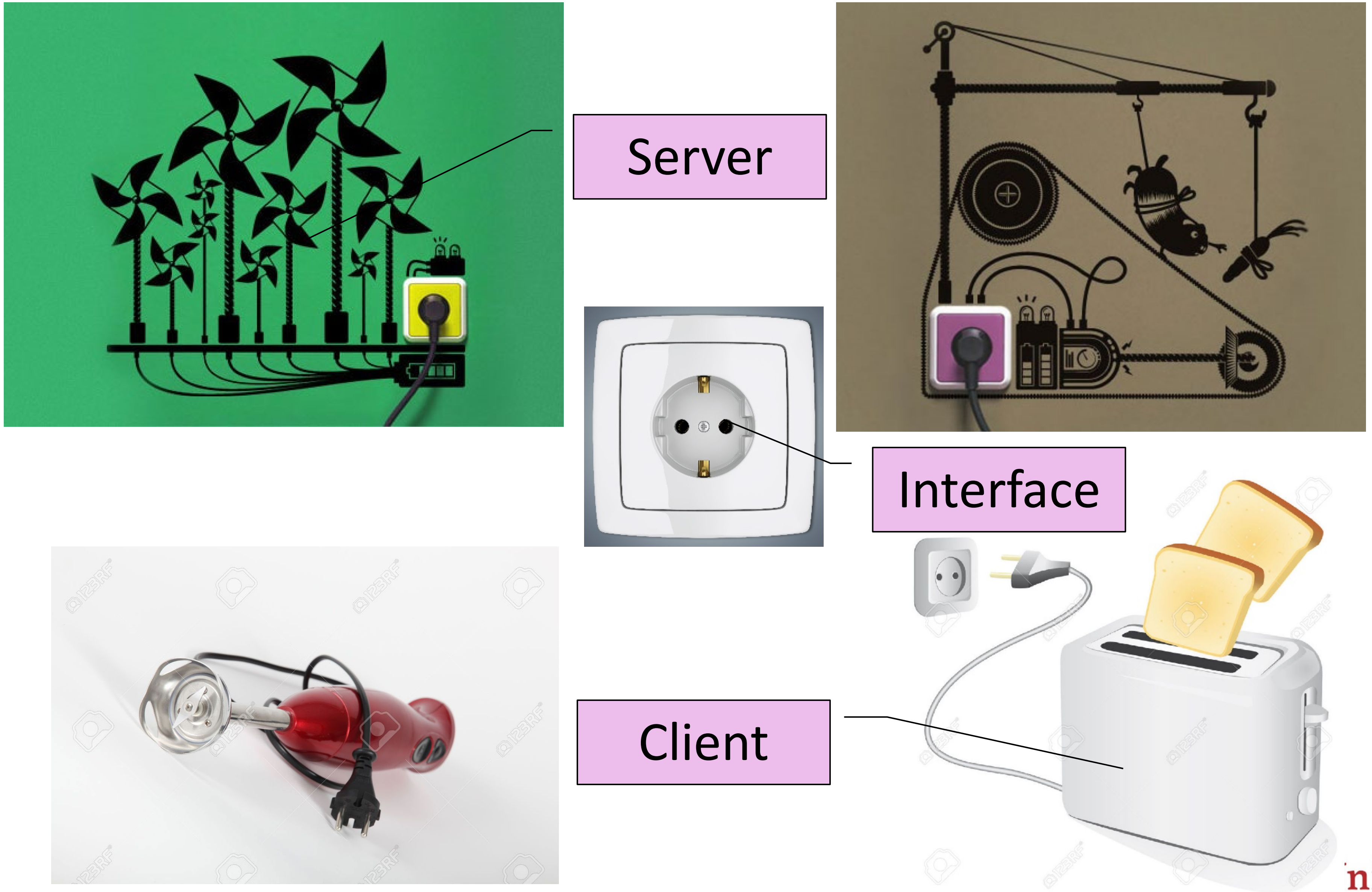
Interfaces - MVC

Lecture 3 (9 February 2021)

Interfaces



Server

Interface

Client

Interfaces: motivation

- Suppose we have two classes C1 and C2 that must be able to cooperate (i.e. calling each others methods).
- Solution: Add attributes to C1 and C2 referring to the others class.

```
public class C1 {
    private C2 myC2;

    public C1( C2 c2 ) {
        myC2 = c2;
    }
    …
}
```

```
public class C2 {
    private C1 myC1;

    public C2( C1 c1 ) {
        myC2 = c1;
    }
    …
}
```

- Disadvantage: C1 and C2 are (too) tightly coupled, entangled.
- Also has an instantiation issue.

Interfaces

- Goal: minimizing dependencies.
- Solution (to previous dependency problem): use interfaces.
- Determine for each class the maximum functionality required by the other class; put these methods in an interface

```
public interface C1usedByC2 {
    void m1C1 ();
}
```

```
public interface C2usedByC1 {
    void m1C2 ();
}
```

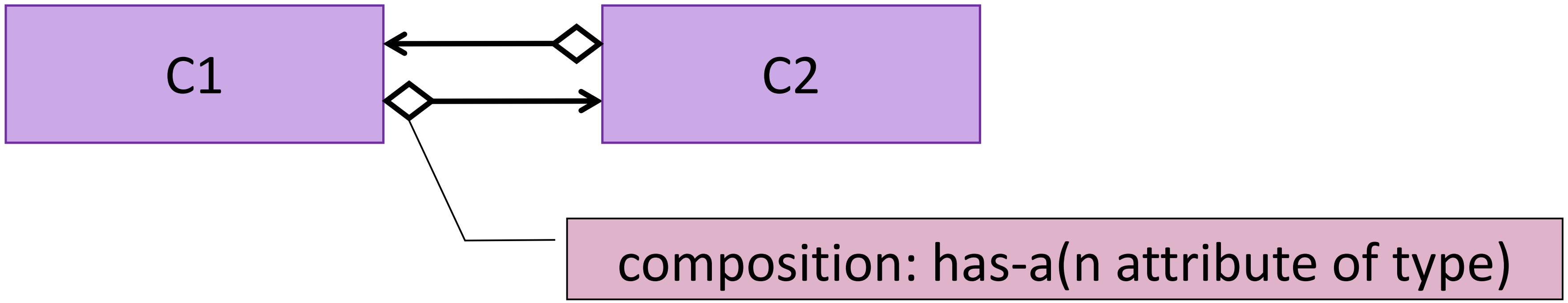
```
class C1 implements C1usedByC2 {
   private C2usedByC1 myC2;
   public C1( C2usedByC1 c2 ) {
       myC2 = c2;
   }
}
```

```
class C2 implements C2usedByC1 {
   private C1usedByC2 myC1;
   public void setC1( C1usedByC2 c1 ) {
       myC1 = c1;
   }
}
```

- One can use skeleton classes with *method stubs* during development.

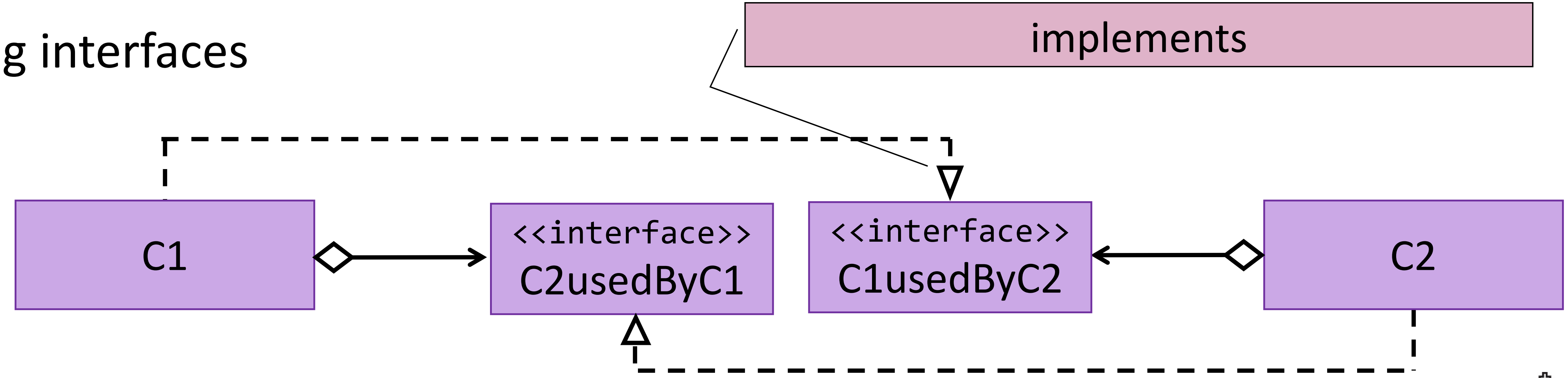
Class diagrams

- Without interfaces



composition: has-a(n attribute of type)

- Using interfaces



implements

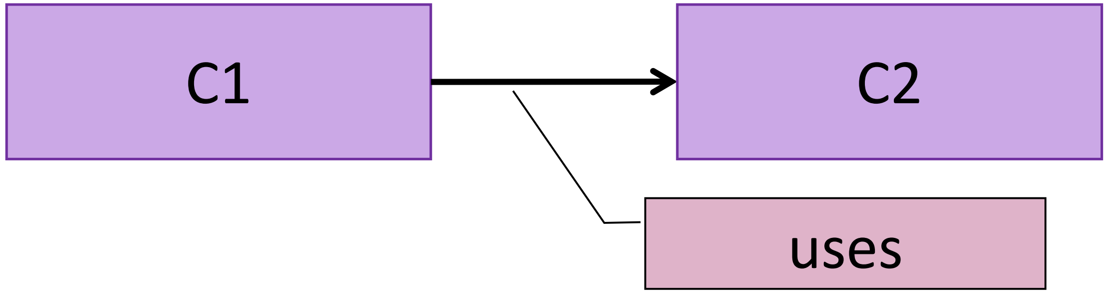
Interfaces (II)

- interfaces can also be used with simpler dependencies: C1 depends on C2 but not the other way around.

```
class C1 {
    <…>
    public void mc1( C2 c2 ) {
        c2.mc2();
    }
}
```

```
class C2 implements {
    <…>
    public void mc2() {
        some statements here;
    }
}
```

- Diagram



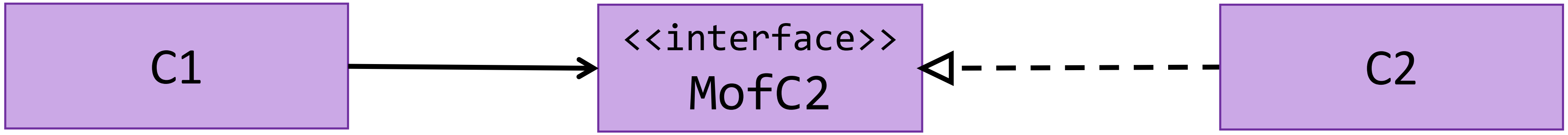
Interfaces (III)

- Solution abstract from the class C2 using an interface.

```
public interface MofC2 {
    void mc2 ();
}
```

```
class C1 {
   <…>
   public void mc1( MofC2 c2 ) {
      c2.mc2();
   }
}
```

```
class C2 implements MofC2 {
   public void mc2() {
      some statements here;
   }
}
```



Example: Shopping cart

```
public class ShoppingCart {
  private Item[] items;
  private int nrOfItems;
  private static final int MAX_NR_ITEMS = 10;

  <...>

  public void pay( CreditCard cc ) {
     cc.pay(  total() );
  }
}
```

```
class CreditCard{

  public void pay() {

    <…>

  }

}
```

- Abstracting from the paying method.

```
public class ShoppingCart {
  private Item[] items;
  private int nrOfItems;
  private static final int MAX_NR_ITEMS = 10;

  <...>

  public void pay( Payment pm ) {
     pm.pay(  total() );
  }
}
```

```
public interface Payment {

    void pay( double amount );

}
```

```
class CreditCard implements Payment {

  public void pay( double amount ) {

    <…>

  }

}
```

OO-design

Design of a System

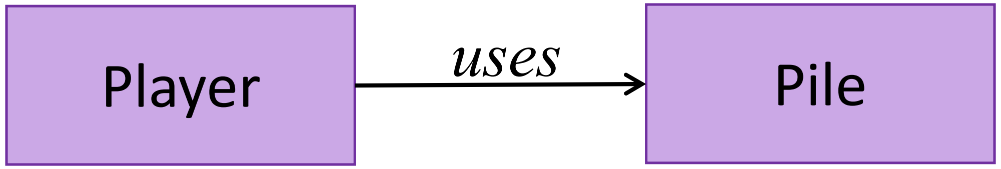
- Game of “simple nim”: there are two players and a pile of sticks. Each player, in turn, removes one, two, or three sticks from the pile. Player who removes the last stick loses.
- Initial implementation games will be played “computer vs. computer.”
- User determines whether to play another game and how many sticks to start with.

Design of Nim game model

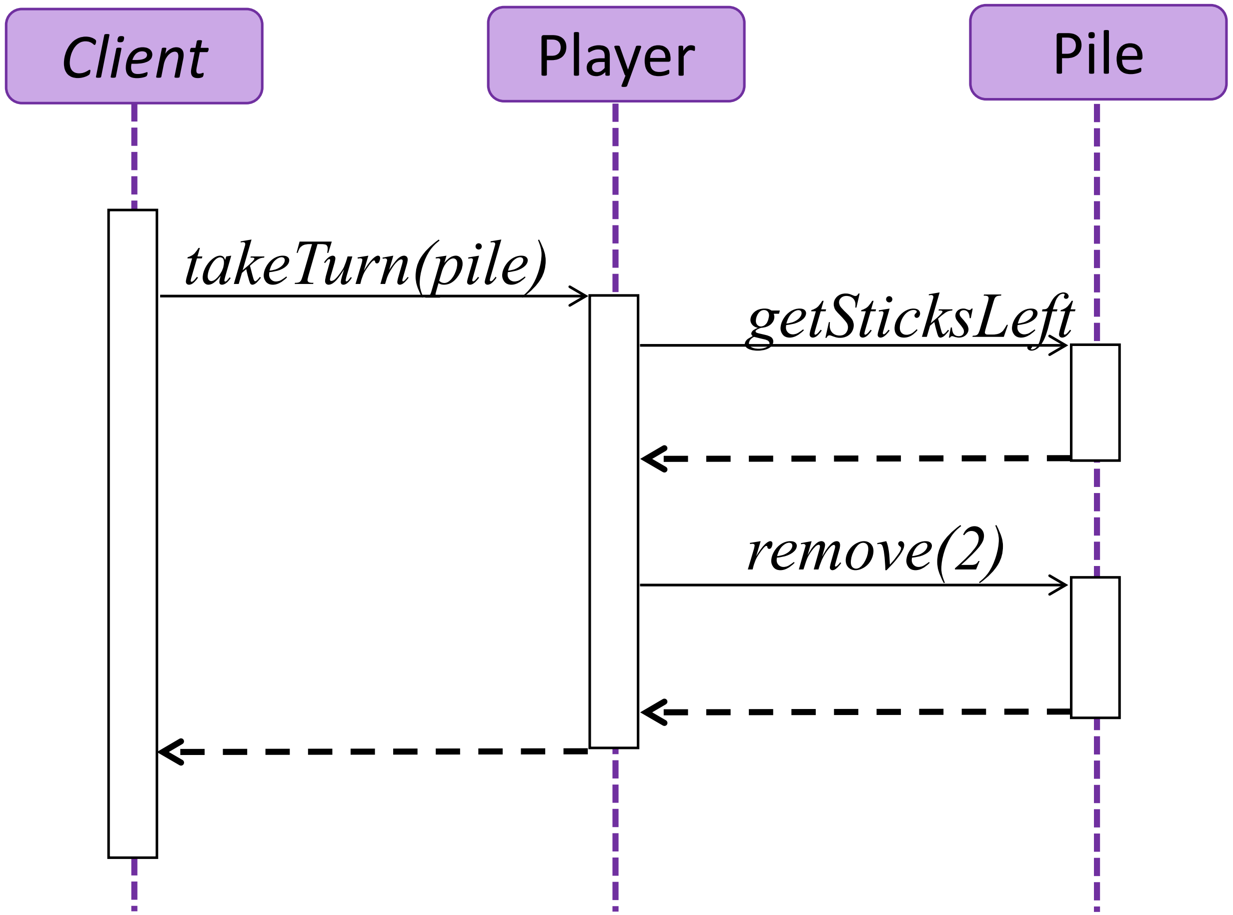
- Two objects for the picking:
  - Player
  - Pile of sticks
- Pile and Player are part of the model of the problem.
- MVC-principle: specify core aspects of the game (model) independently of how these are presented (view) to a user or how a user interacts with them (controller).

Pile and Player

- Class: *Pile*
  - *commands:*
    remove *:* reduce number of sticks by specified amount (*number*)
- Class: *Player*
  - *commands:*
    takeTurn remove 1, 2, or 3 sticks from the specified *Pile* (*pile*)



Sequence diagram: Player takes turn



Implementing the class Pile

```
pubic class Pile {
    private int sticksLeft; // sticks left in the Pile

    public Pile( int sticks ) {
        sticksLeft = sticks;
    }

    public int getSticksLeft() {
        return sticksLeft;
    }

    public void remove( int number )
        sticksLeft = sticksLeft - number;
    }
}
```

Implementing the class Player

```java
public class Player {
    private String myName;   // this Player’s name
    private int sticksTaken; // sticks taken on this
                             // Player’s most recent turn
    public Player( String name ) {
        this.myName = name;
        this.sticksTaken = 0;
    }
    public String getName() {
        return myName;
    }
    public int sticksTaken() {
        return sticksTaken;
    }
    public void takeTurn( Pile p ) { … }
}
```

Discussed later

Design of a System (2)

- What do we need more?
- Initial implementation games will be played “computer vs. computer.”
- User determines whether to play another game and how many sticks to start with.

User interface

- When the program is run, the user is offered the following menu:

```
Enter the number denoting the action to perform:
Run game...............1
Exit...................2
Enter choice:
```

- Entering 2 terminates the program.
- Entering 1 produces the following prompt:

```
Enter number of sticks (a positive integer):
```

User interface specifications

```
class NimTUI
  A simple text-based user interface for the simple nim system.
      public NimTUI ()
            Create a new user interface.
      public void start ()
            Start the interface.
```

Initializing (main) class

- The initiating class will look like this:

```
public class NimGame {
   public static void main( String[] argv ) {
      (new NimTUI()).start();
   }
}
```

System Design (Game CRC-card)

Class: Game

a manager of a simple nim game

Responsibilities:

do:

conduct a play of game,
instructing appropriate *Player* to take a turn

know:

the *Players*
the *Pile*
number of sticks that can be taken on a turn
which *Player* plays next
when the game is over
which *Player* won when game is over

Collaborators

Players, Pile

Implementing class Game

```
class Game {
        private static final int MAX_ON_A_TURN = 3;
        private Player player1;
        private Player player2;
        private Player nextPlayer;
        private Pile pile;
```

- Alternatively

```
class Game {
        private static final int MAX_ON_A_TURN = 3;
        private Player[] players;
        private int nextPlayer;
        private Pile pile;
```

class Game: constructor

```
public Game( Player player1, Player player2, int sticks ) {
        this.player1 = player1;
        this.player2 = player2;
        this.nextPlayer = player1;
        this.pile = new Pile( sticks );
}
```

Or

```
public Game( Player player1, Player player2, int sticks ){
        this.players = new Player[] { player1, player2 };
        this.nextPlayer = 0;
        this.pile = new Pile( sticks );
}
```

class Game: queries

```java
public int sticksLeft() {
      return pile.getSticksLeft();
}
public Player nextPlayer() {
      return nextPlayer;
}
public boolean gameOver() {
      return pile.getSticksLeft() == 0;
}
public Player winner() {
    if ( gameOver() ) {
         return nextPlayer
    } else {
         return null;
    }
}
public String toString() {
      return "Game with players: " + player1 + ", and " +   player2;
}
```

class Game: commands

```
public void play() {
     if ( ! gameOver() ) {
         nextPlayer.takeTurn( pile, MAX_ON_A_TURN );
         nextPlayer = otherPlayer( nextPlayer );
     }
}

private Player otherPlayer( Player player ) {
     return (player == player1) ? player2 : player1;
}
```

conditional operator: see IJPDS, 125

Modeling alternative implementations

- Strategies Player can implement when making a move (in `takeTurn`):
  - Timid strategy
  - Greedy strategy
  - Clever strategy
- We could define a separate class for each strategy:
  - **`class`** `TimidPlayer  { … }`
  - **`class`** `GreedyPlayer { … }`
  - **`class`** `CleverPlayer { … }`

🚫

Solution: Abstraction

- Player clients should be
  - independent of implementations, i.e. strategies chosen;
- Use an interface
- Like the following?

```
public interface Player {
     public String getName();
     public int sticksTaken();
     public void takeTurn( Pile pile, int maxOnATurn );
}
```

Interface implementations

```java
public class TimidPlayer implements Player {
    private String myName;
    private int sticksTaken = 0;

    public TimidPlayer( String name ) {
        this.myName = name;
    }
    public String getName() {
        return myName;
    }
    public int sticksTaken() {
        return sticksTaken;
    }
    public void takeTurn( Pile pile,
                     int maxOnATurn  ) {
        sticksTaken = 1;
        pile.remove( sticksTaken );
    }
}
```

```java
public class GreedyPlayer implements Player {
    private String myName;
    private int sticksTaken = 0;

    public GreedyPlayer( String name ) {
        this.myName = name;
    }
    public String getName() {
        return myName;
    }
    public int sticksTaken() {
        return sticksTaken;
    }
    public void takeTurn( Pile pile, int maxOnATurn  ) {
       sticksTaken=Math.min(maxOnATurn,
                            pile.getSticksLeft());
       pile.remove( sticksTaken );
    }
}
```

Interfaces and types

- An interface defines a (reference) type.
- A reference is in the interface type if it *points to an instance of a class that implements the interface.*
  - A reference of type TimidPlayer, is also of type Player.
  - A reference of type GreedyPlayer, is also of type Player.
  - A reference of type CleverPlayer is also of type Player.

Interface and types

- The types TimidPlayer, GreedyGlayer and CleverPlayer are said to be subtypes of the type Player.
- Player is a supertype of TimidPlayer, GreedyGlayer and CleverPlayer.
- A type defined by an interface can be used like any other reference type.
  - It can be the type of an attribute, local variable, or parameter.
  - It can be the return type of a method.

Types and Subtypes

- If client expects a reference of type *Player*, then a value of any *Player* **subtype** can be provided.
- Subtype rule:
  - *if type B is a subtype of type A, then*

    a B value can be provided wherever an A value is required.
- Thus for Game constructor:
  - It can be specified with parameters of type Player interface;
  - It can be invoked with arguments referencing TimidPlayers, GreedyPlayers, CleverPlayers.

```
new Game ( new TimidPlayer("..."), new CleverPlayer("..."), 17 );
```

Static types

- The Game method nextPlayer is specified as

```
public Player nextPlayer ()
// The Player whose turn is next.
```

- If game refers to a Game instance, Player is the static type (compile-time type) of expression

```
game.nextPlayer()
```

Dynamic types

- When `game.nextPlayer()` is evaluated during execution, the value returned will reference a specific object:
  - If an instance of TimidPlayer. The dynamic type (run-time type) of value returned by expression is TimidPlayer.
  - If an instance of GreedyPlayer. The dynamic type of value returned by expression is GreedyPlayer.
  - If an instance of CleverPlayer. The dynamic type of value returned by the expression is CleverPlayer.
- The dynamic type is always a subtype of (the static type) Player.

Types and Subtypes

- The following require expressions of type Player:

```
private Player nextPlayer;
private void reportPlay( Player player ) …
public Player winner() …
```

```
nextPlayer = Player expression required;

reportPlay( Player expression required );

public Player winner() {
   …
   return Player expression required;
}
```

Types and Subtypes

- Given

```
TimidPlayer timid = new TimidPlayer( "Zorro" );
```

- The following are legal:

```
nextPlayer = timid;

reportPlay( timid );

public Player winner() {
   …
   return timid;
}
```

Types and Subtypes

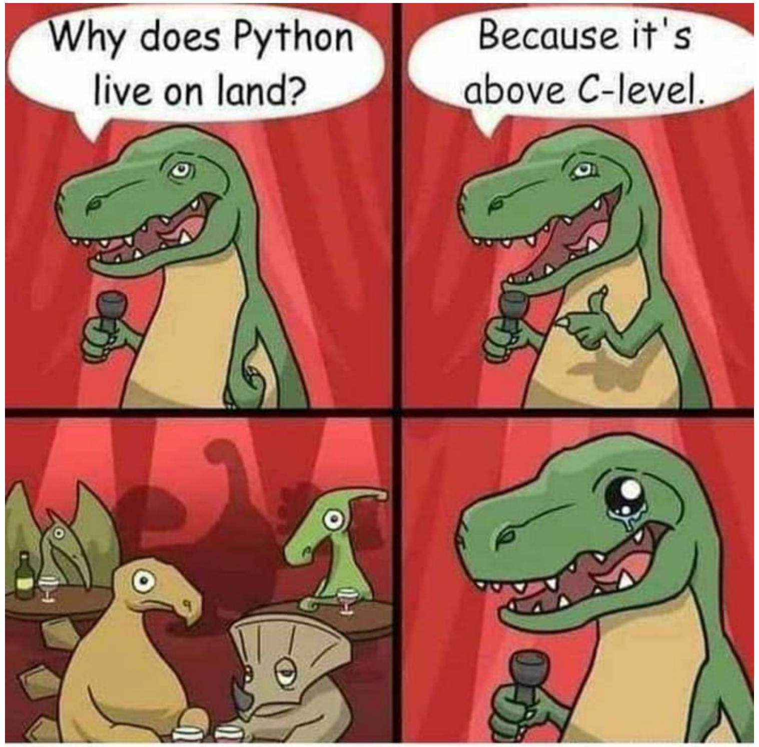
- If game is a Game instance, we cannot write the following

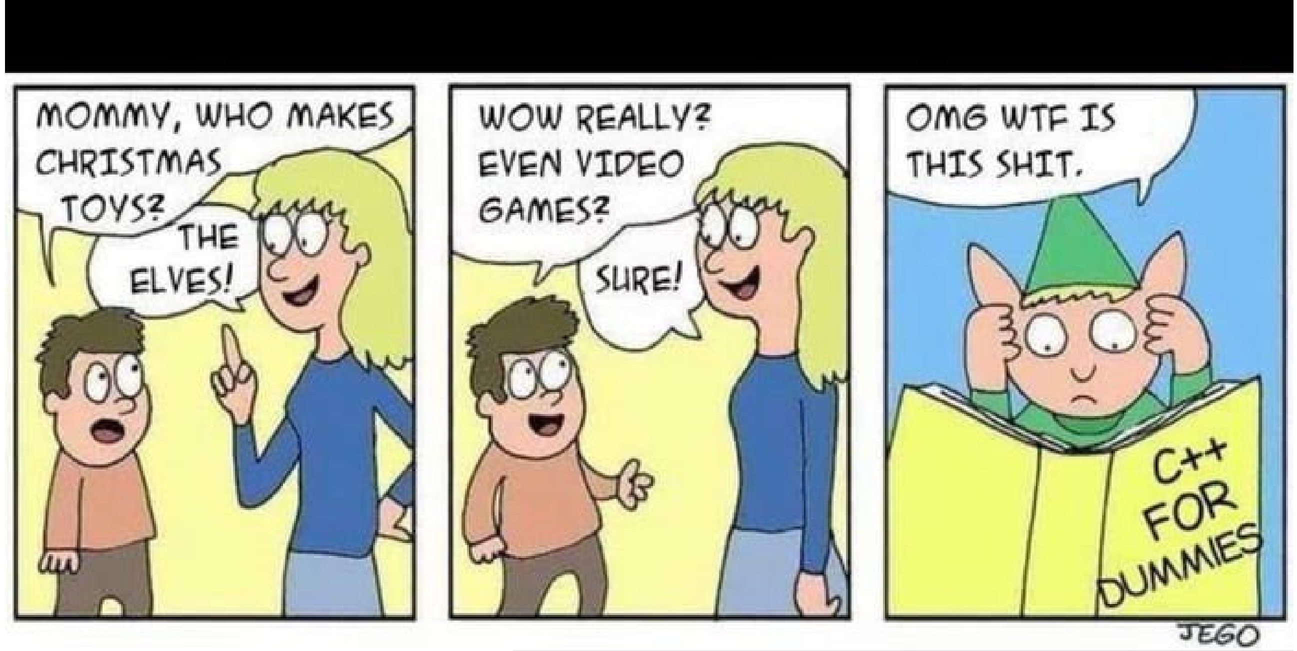
```
TimidPlayer next = game.nextPlayer();
```

🚫

- Assignment operator requires a TimidPlayer on the right.
  - game.nextPlayer() is of type Player, and Player is not a subtype of TimidPlayer.

Joke of the week




MOMMY, WHO MAKES CHRISTMAS TOYS?
THE ELVES!
WOW REALLY? EVEN VIDEO GAMES?
SURE!
OMG WTF IS THIS SHIT.
C++ FOR DUMMIES
JEGO

Programmers looking at programming memes
Ah, humor based on my pain.
Ah, ha, ha.

The strategy interface

- Looking player classes, note duplicated code.
- Only difference in TimidPlayer, GreedyPlayer, and CleverPlayer is body of takeTurn.
- Duplicate code is a prime cause of maintenance headaches, avoid when possible.
- Can reduce duplicate code in player classes.

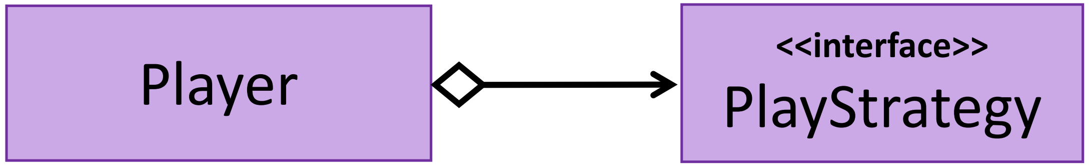
Interface implementations

```
class TimidPlayer implements Player {
    private String myName;
    private int sticksTaken = 0;

    public TimidPlayer( String name ) {
        this.myName = name;
    }
    public String getName() {
        return myName;
    }
    public int sticksTaken() {
        return sticksTaken;
    }
    public void takeTurn( Pile pile,
                          int maxOnATurn  ) {
        sticksTaken = 1;
        pile.remove( sticksTaken );
    }
}
```

```
class GreedyPlayer implements Player {
    private String myName;
    private int sticksTaken = 0;

    public GreedyPlayer( String name ) {
        this.myName = name;
    }
    public String getName() {
        return myName;
    }
    public int sticksTaken() {
        return sticksTaken;
    }
    public void takeTurn( Pile pile, int maxOnATurn  ) {
       sticksTaken=Math.min(maxOnATurn,
                                    pile.getSticksLeft());
       pile.remove( sticksTaken );
    }
}
```

The strategy pattern (2)

- Don’t change `Player` into an interface.
- Instead, give the `Player` a component that determines what move to make.



Player

<<interface>>
PlayStrategy

- The `PlayStrategy` interface is defined as follows:

```
interface PlayStrategy {
   public int numberToTake( Pile pile, int maxOnATurn );
}
```

- A Player will have an attribute referencing a `PlayStrategy`.

```
private PlayStrategy strategy;
```

Strategy pattern (3)

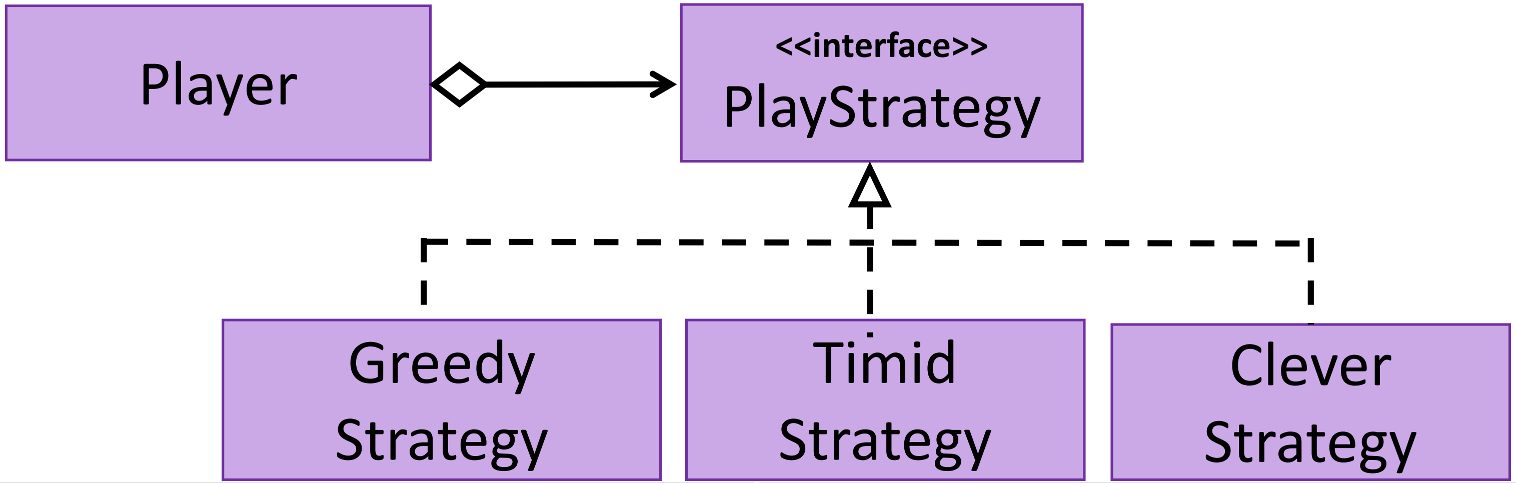
- takeTurn delegates responsibility for determining how many sticks to take to PlayStrategy:

```
public void takeTurn( Pile pile, int maxOnATurn ) {
   int sticksTaken = strategy.numberToTake( pile, maxOnATurn );
   pile.remove( sticksTaken );
}
```

method from PlayStrategy

Strategy pattern (4)

- PlayStrategy is an interface that can be implemented in various ways.



```
class TimidStrategy implements PlayStrategy {
   public int numberToTake(Pile pile,
                              int maxOnATurn ) {
      return 1;
   }
}
```

```
class GreedyStrategy implements PlayStrategy {
   public int numberToTake( Pile pile,
                              maxOnATurn ) {
      return Math.min(maxOnATurn,pile.getSticksLeft());
   }
}
```

Strategy pattern (5)

- The easiest way to equip a Player with a PlayStrategy is to provide one as a constructor argument:

```
public Player( String name, PlayStrategy strategy )
```

- Player’s strategy can be changed dynamically.
  - Include method to change Player’s strategy:

```
public void setStrategy( PlayStrategy strategy )
```

Modifying Nim: user vs. computer

- Want same simple nim game and text-based user interface
- Want user to play against “the computer” rather than just watching the game.
- Need two different kinds of players.
  - One player decides its own move;
  - the other gets its move from an external source, the user.
- Does this modification fit in with the strategy interface?

Human strategy

```
interface PlayStrategy {
   public int numberToTake( Pile pile, int maxOnATurn );
}
```

```
public class HumanStrategy implements PlayStrategy {
    private int numberToTake; // Number of sticks to be taken

    public int numberToTake( Pile pile, int maxOnATurn ){
        return numberToTake;
    }

    public void setNumberToTake( int number ) {
        this.numberToTake = number;
    }
}
```

Adding a user interface

- It creates a HumanPlayer and a ComputerPlayer
- It controls the game.

```
private Player human;
private Player computer;
private Game game;
private Scanner in;
private HumanStrategy humanStrategy;

public NimTUI() {
   this.humanStrategy = new HumanStrategy();
   this.human         = new Player( "user", humanStrategy  );
   this.computer      = new Player( "computer", new TimidStrategy() );
   this.in            = new Scanner( System.in );
}
```

Adding a user interface (2)

- Added a method for playing a game with the specified number of sticks and a parameter indicating whether user wants to play first.

```
private void playGame( int numberOfSticks,
                       boolean userPlaysFirst ) {
  if ( userPlaysFirst ) {
       game = new Game ( user, computer,numberOfSticks );
  } else {
       game = new Game ( computer,user, numberOfSticks );
  }
  while ( ! game.gameOver() ) {
        game.play();
        reportPlay( game.previousPlayer() );
  }
  reportWinner( game.winner() );
}
```

User interface – model interaction (1)

- How does user interface know when to get a play from user?
  - user interface checks whose turn it is before invoking **play**
  - Need to add a conditional to play loop

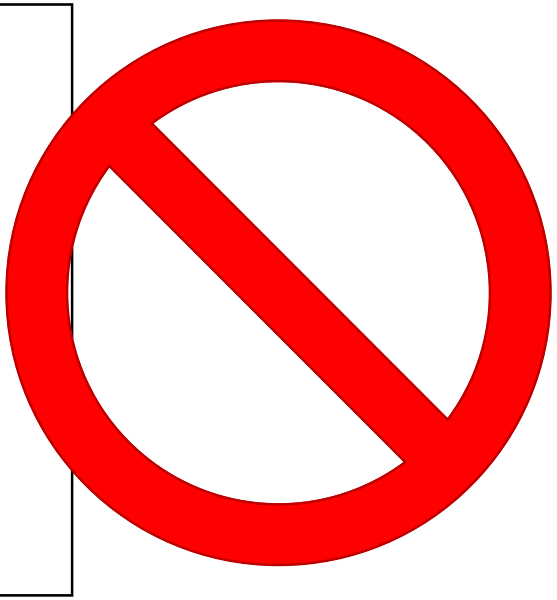
private method of NimTUI

```
while ( ! game.gameOver() ) {
       if ( game.nextPlayer().equals( user ) ) {
          int numberToTake = readNumberToTake();
          humanStrategy.setNumberToTake( numberToTake );
       }
       game.play();
       reportPlay( game.previousPlayer() );
}
```

User interface – model interaction (2)

- **Problem**: user interface is more involved in play of the game.
- Want “dumb” user interface, as isolated from model as possible.
- Role of the user interface is to manage input and output: its knowledge about how the model works should be minimized.

```
while ( ! game.gameOver() ) {
        if ( game.nextPlayer().equals( user ) ) {
            int numberToTake = readNumberToTake();
            humanStrategy.setNumberToTake( numberToTake );
        }
        game.play();
        reportPlay( game.previousPlayer() );
}
```



User interface – model interaction (3)

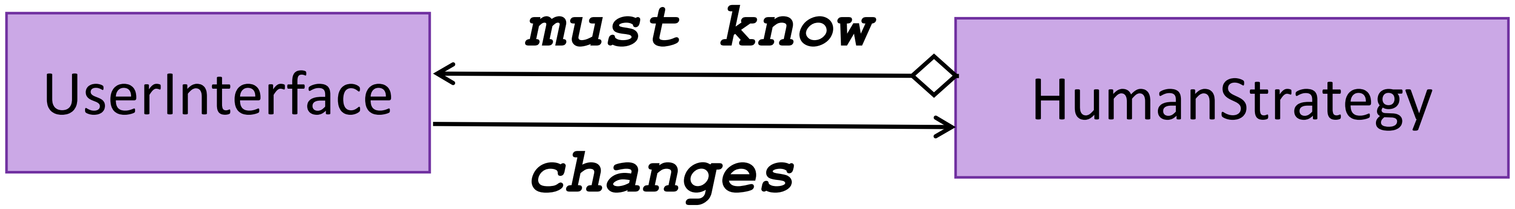
- HumanStrategy tells user interface it needs a move.
- This alternative requires the model to be **client** of user interface.
  - In general, we don’t want a model to dependent on user interface.
- Common: User interface (client) needs to know that the model (server) has reached a state in which it needs input from the user.
- HumanStrategy
  - must know the user interface.
  - needs to notify it when it is about to make a move.

User interface – model interaction (4)

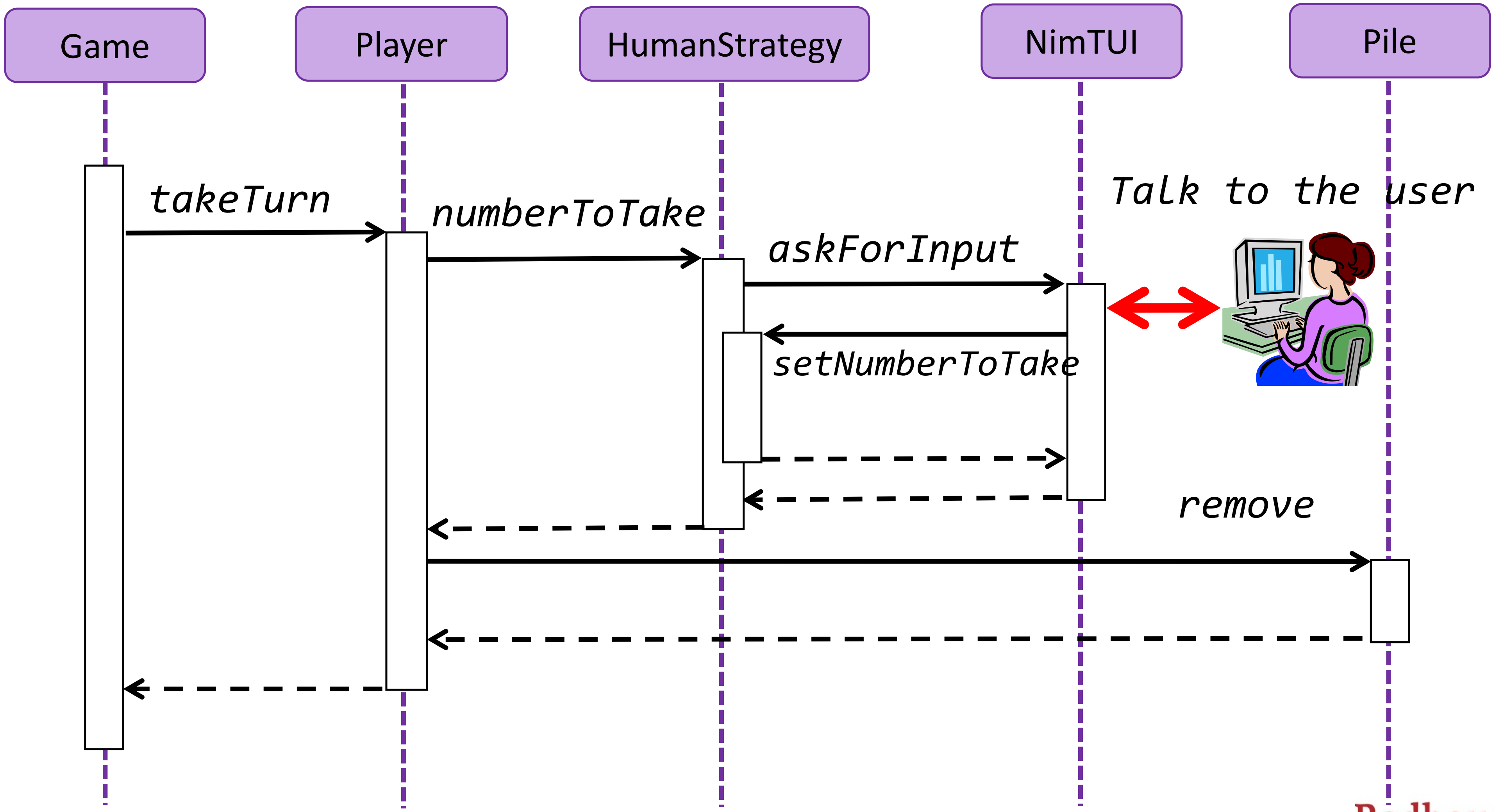
- What do we need to do?
  - Extend user interface with a method askForInput.

```
public void askForInput( HumanStrategy player, int max )
```

  - To invoke askForInput the human player should have access to the user interface
  - Add an attribute to the HumanStrategy class that refers to the user interface.



Sequence diagram: Human player takes turn


Game

Player

HumanStrategy

NimTUI

Pile

takeTurn

numberToTake

askForInput

setNumberToTake

Talk to the user

remove

A UI-abstraction

- Can we avoid the NimTui attribute in the HumanStrategy?
- Yes, by using an interface!

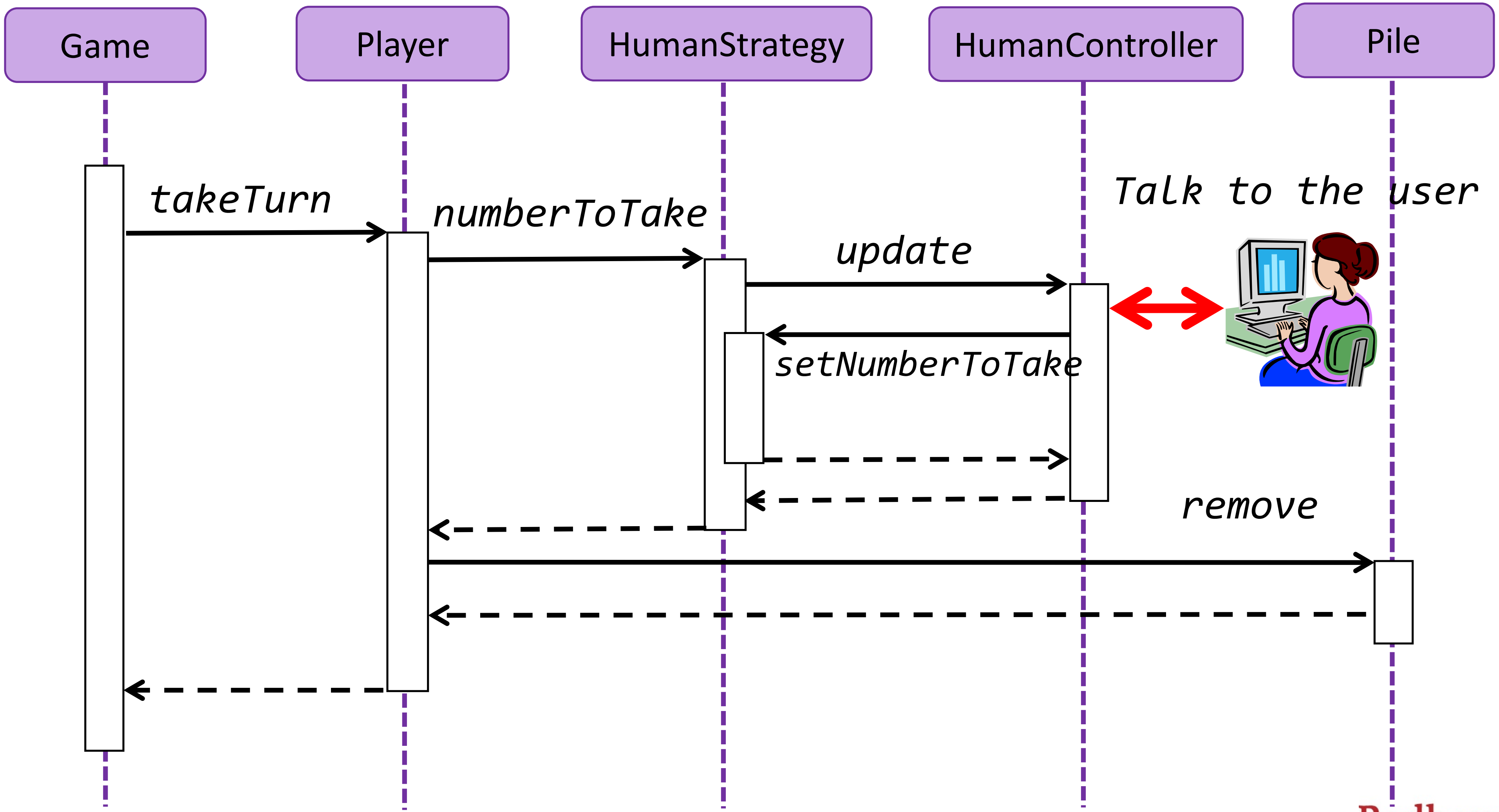
```
interface HumanObserver
   Models an object that needs to be informed when a
   human player is about to make a play.
    public void update( HumanStrategy player, int MaxOnTurn )
      The specified HumanStrategy is making a play.
```

- We remove the code implementing the dialogue with the user from NimTui and put it into a separate class HumanController
- HumanController implements the HumanObserver interface

Class HumanController

```
class HumanController implements HumanObserver {
    private Scanner in;

    public HumanController( Scanner in ) {
        this.in = in;
    }

    public void update( HumanStrategy human, int maxOnATurn ) {
        int numberToTake = readNumberToTake( maxOnATurn );
        human.setNumberToTake(numberToTake);
    }

    private int readNumberToTake( int max ) {
       ...
    }
}
```

Sequence diagram: Human player takes turn



Changing HumanStrategy

- In order to get the input from the user the HumanStrategy informs the controller to start the interaction

```java
public class HumanStrategy implements PlayStrategy {
    private int myNumberToTake;
    private HumanObserver myController;

    @Override
    public int numberToTake( Pile pile, int maxOnATurn ) {
        myController.update( this, min( maxOnATurn, pile.getSticksLeft() ) );
        return myNumberToTake;
    }
    public void setNumberToTake( int number ) {
        myNumberToTake = number;
    }
    public void register( HumanObserver controller ) {
        myController = controller;
    }
```

this strategy informs the observer

To set the observer this strategy reports to.

To summarize

- What has changed:

  - `HumanObserver` interface added
  - `HumanController` class added:
    - implements `HumanObserver`
    - `registers` itself to the human player
  - `HumanStrategy`:
    - extended with an attribute of type `HumanObserver` that can be set with the method `register`.
    - asks the Observer to provide the number of sticks that are going to be removed (by invoking `update`).

The MVC principle

- The Model determines the main behavior of the system.
- The View (or a View) is a way of looking at or displaying the model
- The Controller provides for user input and translates user actions into model modifications

- These three components are usually implemented as separate classes

Finally

- questions?

NEXT WEEK

Lecture 4: Inheritance