Jframe: Title ”Tic-tac-toe”
JPanel: knappar? Jtable? Ej i scroll; Vilken resolution? Hämta fullscreen
Dialogfönster? Button-addbutton for adding players
Input med spelarnamn? TextField? Spelarklass def
Jdialog, använd ImageIcon
Interfaces?

Action-listeners; Klick? Keypress med index? Olika game-modes där?

Bäst av-funktion (Du får ut vinnaren av ett visst antal spel)
Du klickar på en och den switchar 2 andra lite Rubiks kub-vibe, mer komplicerade rörelsemönster

Gör UML? → Skriv ut klass-specifikation ordentligt

Player → Datorspelare och HumanPlayer: Arvsstruktur (ÄR)*

```
Player {
String name;
int[][] activeCoordinates;
Player(inGrejer){
Randomisera om den är x eller o, typ greja en char[2] xorO = {'X', 'O'}; och sen
//while(!seriesOfGamesIsOver){
//Så alltså innan inledning av spel förldelas x/o genom
int index = randomGen.nextInt(0,2);
player1.setxorO(xorO[index]);
//if index = 0 then 1-0 = 1, if index = 1 then 1-1 = 0.
player2.setxorO(xorO[1-index]);


while(!gameIsOver){









}

}

makeMove(PlayingField){
Kommer behöva någon modulus-shit

}
```

Ett spelfield har spelare
Spelare kommer behöva metoder för att göra  drag
Layoutmanager (borderlayout)

håll den i centern pga vi vill ge den extra space

Vi kommer behöva en spelplan. Den ska kolla av om den sista aktionen ledde till vinst eller inte → Bara kolla den → Posta också?

Logik: Vi kan bara kolla om det finns någon på var sin sida om, vi kan lägga en boolean på varje plats → Koppla till while.-sats → Vi hade kunnat göra en for-sats som kollar för varje

AI-tankar: Kolla 2 random, möjliga drag. Schackdator-metoden. Evaluera bästa.
(II) gör **näst-bästa** draget?? Speltesta. Kolla faktorer för hur stark ett ”save-state”
Annars måste vi börja tänka på hur spelet fungerar, tänk på det som matris. Vad kan vi göra för regler?

Datastrukturellt?
Vi hade kunnat ha en snapshot genom att ha all historik i en array (för varje runda så hade det motsvarat en index i array du vet enligt tidigare)

Abstrakt klass:
En klass som ryggradsdjur(Föräldraklass), varje ryggradsdjur **är** en typ av djur som instantierar som t.ex Noshörning etc (barnklasser)
((Var ska vi komma åt det, vad är det (orden abstrakt och klass) och namn på denna. Först vad sen namn)

Protected: Kan kommas åt i barnklasser och i egna klassen men inte public

```
public abstract class Ship {
    protected final String[] allShipTypes =
{"Submarine","Torpedoship","Cruiser","Battle-cruiser"};
    protected Point[] shipCoordinates;
    protected String shipType;
/** Is responsible for putting all ships in a collection (fleet), helps initiate
ships and gather common operations
 * for the ship-subclasses.
 */
    /** Creates a ship, sets shipCoordinatesIn as shipCoordinates and accounts
for ship-type*/
    protected Ship(Point[] shipCoordinatesIn){
        shipCoordinates = shipCoordinatesIn;
        switch(shipCoordinates.length){
            case 1: shipType = allShipTypes[0];break;
            case 2: shipType = allShipTypes[1]; break;
            case 3: shipType = allShipTypes[2]; break;
            case 4: shipType = allShipTypes[3];break;
        }
    }
    /**This operation returns a point-array containing the coordinates for the
ship.
     *
     *
     *
     *
     * @return
     */
```

```
    protected Point[] getShipCoordinates(){
        Point[] pReturn = shipCoordinates;
        return pReturn;
    }
    /**This operation will return the type of the ship (Submarine,
cruiser...etc)
     * will be useful when we register a hit -> The type of ship must be
mentioned
     *
     * @return String shipType
     */
    protected String getShipType(){
        String s = shipType;
        return s;
    }
    /**This operation will allow players to set a position for their ship in the
first round, specified
     in each of the subclasses */
    public abstract void setPosition(Point[] shipCoordinatesIn);
}
```

Adam:
Skapa en abstrakt player-klass, ha med properties som delas av alla spelartyper (metoder som båda gör och t.ex då propertien namn som båda **har,** glöm inte access-modifier som är speciell vad det gäller föräldraklasser).

/**This operation will allow players to set a position for their ship in the first round, specified in each of the subclasses/* public abstract void setPosition(Point[] shipCoordinatesIn); }

1. Okej denna var väldigt viktig och verkade som den inte riktigt kom med i clipboarden, detta är då ett exempel på en abstrakt metod - Dvs en metod vilken aldrig används utav denna klassen, till skillnad från då t.ex getShipType() och de andra metoderna som kom med vilka då faktiskt kan kallas av barnklasserna med hjälp av "super.getShipType();" vilket pga att den är protected får användas av barnen

2. Annat smidigt med denna typen av struktur är att eftersom föräldraklassen delas av barnklasserna så hade man kunnat för oss, även om HumanPlayer och ComputerPlayer är två olika klasser kan lägga in dem i samma array -> 1 HumanPlayer[] humanArray = new HumanPlayer[humanElement1, element 2, .... lastElement]; ComputerPlayer comp1 = new ComputerPlayer(inGrejer); humanArray[0] = comp1; 1 Exempel 1 funkar ej för de är inte samma klass. Basically så säger vi ju när vi skapar en array av människor att den ska skapa plats för så många humanPlayers i minnet. Där kan man ju inte stoppa in en datorSpelare som har helt andra properties osv. Finns grejer att säga om hur det funkar i minnet vi får ta snabbt imorgon pga så hjälpsamt för programmering! Men ja det vi kan göra 2 HumanPlayer[] humanArray = new HumanPlayer[humanElement1, element 2, .... lastElement]; ComputerPlayer comp1 = new ComputerPlayer(inGrejer); Player[] playerArray = {humanArray[0],humanArray[1].... , comp1} 2 Detta funkar för att båda är av klassen Player och då gör att man kan samla totalt antal spelare tillsammans

TLDR; /*This operation will allow players to set a position for their ship in the first round, specified in each of the subclasses/ public abstract void setPosition(Point[] shipCoordinatesIn); } Okej denna var väldigt viktig och verkade som den inte riktigt kom med i clipboarden, detta är då ett exempel på en abstrakt metod - Dvs en metod vilken aldrig används utav denna klassen, till skillnad från då t.ex getShipType() och de andra metoderna som kom med vilka då

faktiskt kan kallas av barnklasserna med hjälp av "super.getShipType();" vilket pga att den är protected får användas av barnen

3. Denna metod kollas alltid av mot barnklassens definition av metoden, den föregås av @override

4. @Overrides

5. *

6. i barnklassens definition då

7.

Right! I konstruktorn snackade vi ju lite om vad vi ville att de aktiva-koordinaterna skulle sättas till

1. (int[3][3] activeCoordinates, när vi skapar en player bör ha något startvärde)

2.