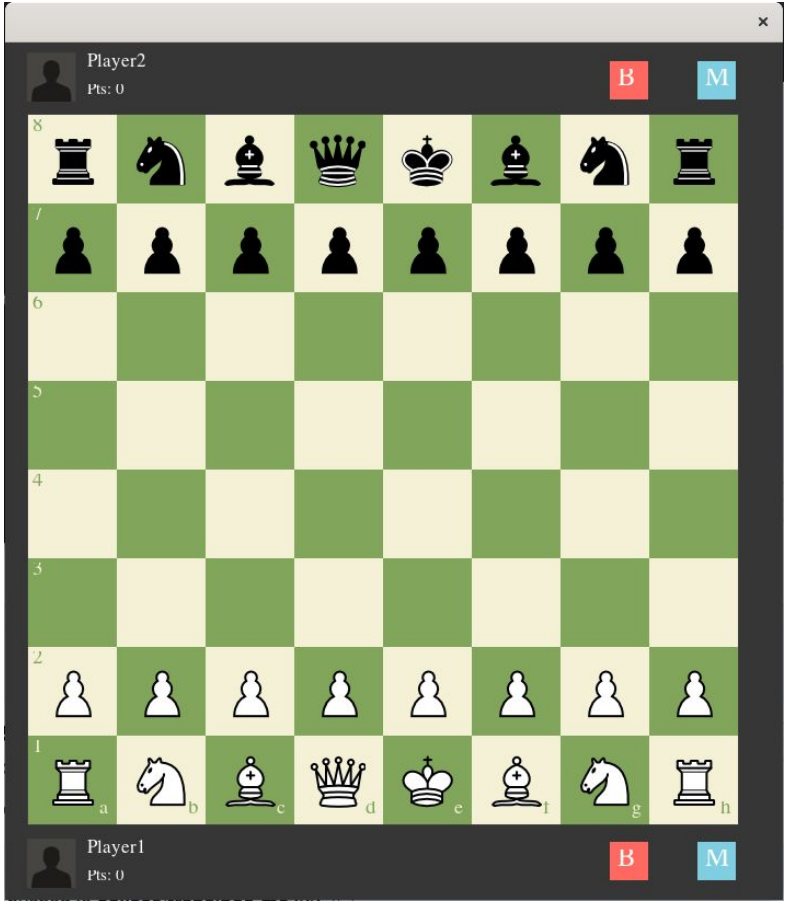
# *War of Nations*

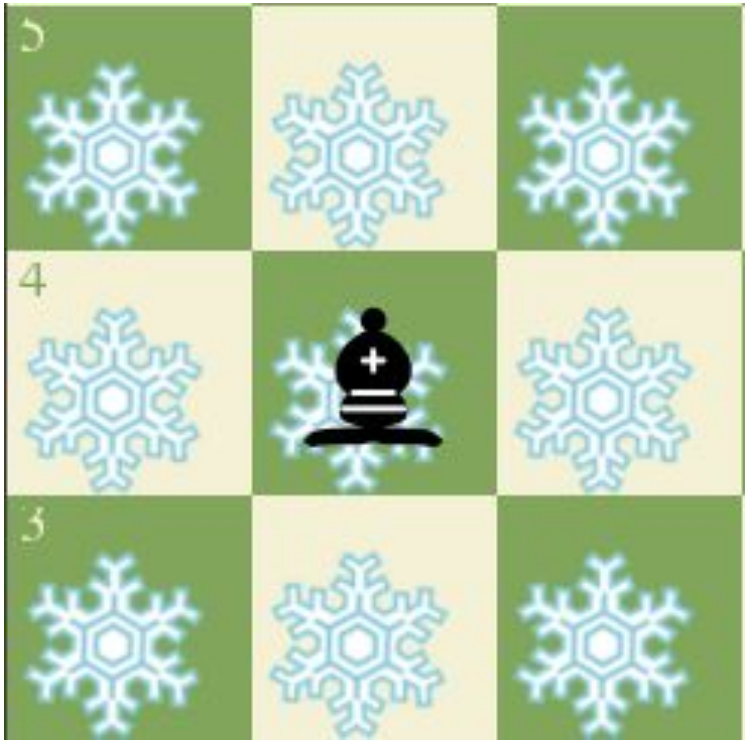
## Sobre o jogo

- É um jogo de xadrez, mas…
- Cada Jogador pertence a uma Nação
- Cada Nação possui duas habilidades: *Habilidade Básica* (3 pt) e *Habilidade  Principal* (9pt)
- Nações que implementamos: Nação de Gelo (Brancas) Nação de Pedra (Pretas)



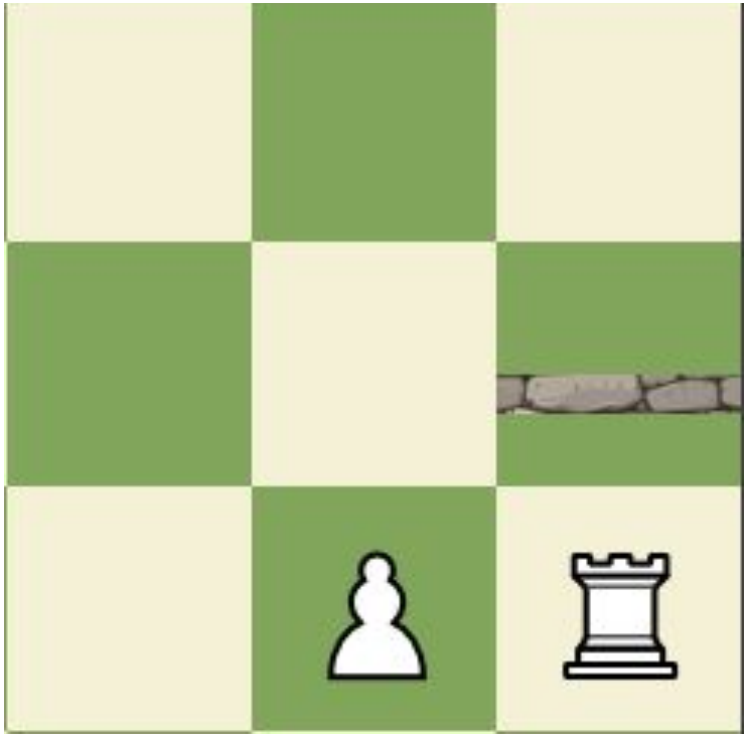
# Nação de Gelo

Habilidade Básica

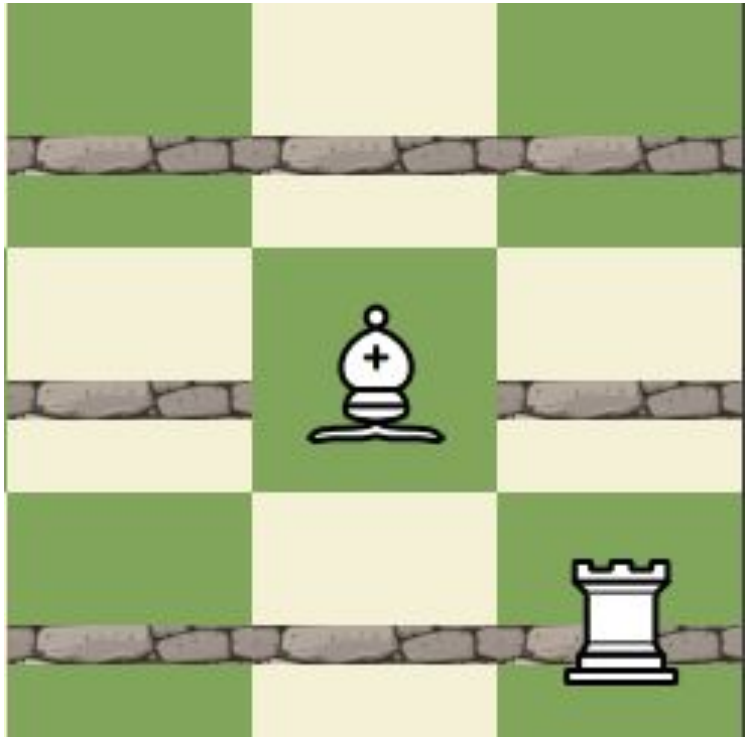



Habilidade Principal

# Nação de Pedra

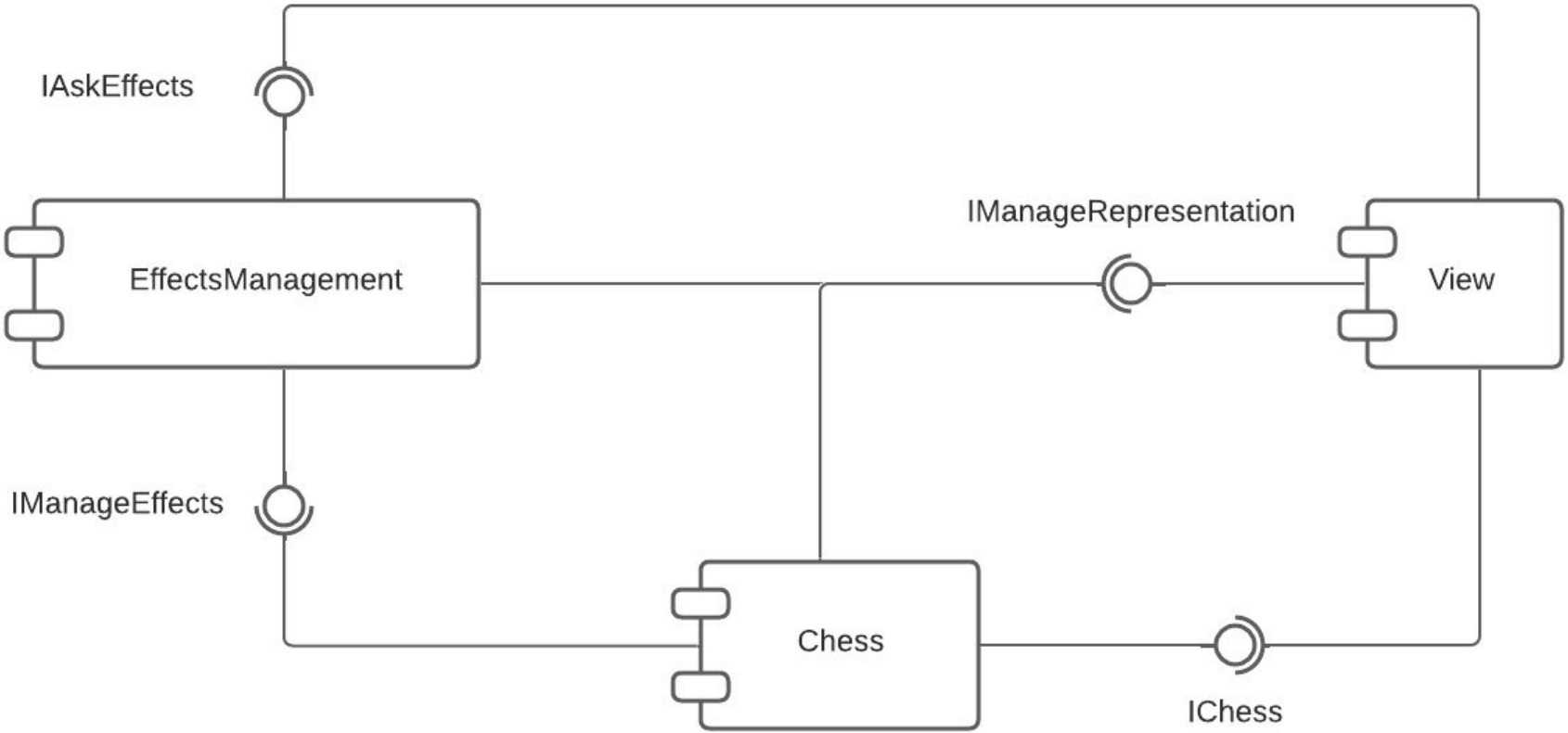


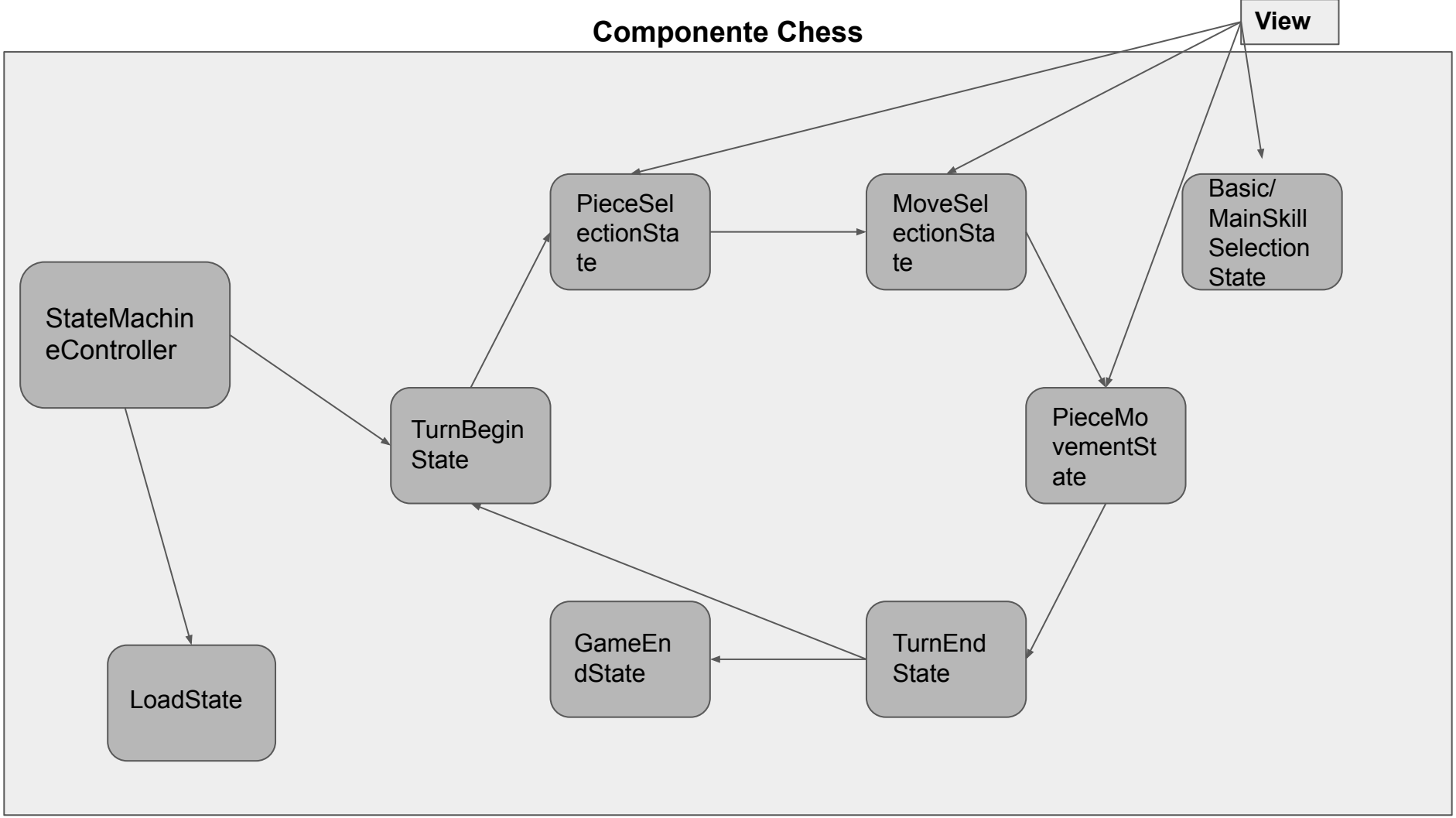
Habilidade Básica



Habilidade Principal

Diagrama Geral




Componente Chess
View
PieceSelectionState
MoveSelectionState
Basic/MainSkillSelectionState
StateMachineController
TurnBeginState
PieceMovementState
LoadState
GameEndState
TurnEndState

```java
public abstract class State {

    public abstract void enter();

    public void exit() {
        ;
    }
}
```

```java
// Muda o jogo do estado atual para um outro estado. Antes de iniciar o novo estado, executa a saída do estado anterior.
public void changeTo(State state) {
    if (this.currentState != state) {

        if (this.currentState != null) {
            this.currentState.exit();
        }

        this.currentState = state;
        if (this.currentState != null) {
            this.currentState.enter();
        }
    }
}

// Inicia o jogo.
public void startGame() {
    changeTo(new TurnBeginState());
}
```

```java
public class StateMachineController implements IChess{   // A máquina de estados controla o fluxo do jogo alternando entre os estados.

    public static StateMachineController instance;  // Instancia estática para acessar o Controller através do Board e do View.

    private Player player1;
    private Player player2;
    private Player currentPlayer;
    private State currentState;
    private Piece selectedPiece;    // Peça selecionada no estado PieceSelectionState
    private int[] selectedHighlight;  // Highlight selecionado no estado MoveSelectionState
```

```java
public class PieceMovementState extends State{   // Movimenta a peça e inicia o estado de fim de turno.

    public void enter(){
        System.out.println("PieceMovementState:");

        Square highlightedSquare = Board.instance.getSquare(StateMachineController.instance.getSelectedHighlight()[0], StateMachineController.i
        MoveType moveType = highlightedSquare.getMoveType();

        // Analisa qual será o tipo de movimento e executa o movimento necessário.
        if (moveType == MoveType.CastlingMovement)
            SpecialsMovements.castlingMovement();

        else if (moveType == MoveType.PawnPromotionMovement)
            SpecialsMovements.pawnPromotion();

        else if (moveType == MoveType.PawnDoubleMovement)
            SpecialsMovements.pawnDoubleMovement();

        else if (moveType == MoveType.EnPassantMovement && StateMachineController.instance.getSelectedPiece() instanceof Pawn)
            SpecialsMovements.enPassantMovement();

        else
            SpecialsMovements.normalMovement();


        // Volta o MoveType do square selecionado para NormalMovement
        highlightedSquare.setMoveType(MoveType.NormalMovement);

        // Após 1 rodada, transforma os moveType EnPassantMovement do inimigo em NormalMovement.
        int iPos = (StateMachineController.instance.getCurrentPlayer().getTeam() == "WhiteTeam") ? 2 : 5;
        clearEnemyEnPassants(iPos);

        StateMachineController.instance.changeTo(new TurnEndState());
    }
```

```java
public enum MoveType {     // Após obter os movimentos válidos para uma dada peça com as classes que herdam Movement, todos os squares
                           // desses movimentos tem seu atributo moveType atualizado para NormalMovement. Após isso, caso haja algum
    CastlingMovement,      // movimento especial para algum desses squares, o atributo moveType é alterado para este movimento especial.

    PawnPromotionMovement,

    PawnDoubleMovement,

    EnPassantMovement,

    NormalMovement;
}
```

```java
public class Square {

    private int position[];

    private Piece piece;

    private boolean isHighlighted; // Indica se o square está marcado com movimento de paça.

    private MoveType moveType;  // Indica se um movimento de uma peça para este Square será normal ou algum movimento especial
```

```java
public abstract class Movement {

    // Retorna uma lista com as posições de todos os movimentos possíveis para uma dada peça. O parametro safeMovements, caso tenha o valor
    // true, faz com que o método retorne apenas os movimentos que evitem que o Rei do jogador fique em cheque.
    public abstract ArrayList<int[]> getValidMoves(boolean safeMovements, Piece piece);

    // Analisa se uma peça é inimiga do jogador passado como argumento
    protected boolean isEnemy(Square square, Player player){ ···

    // Retorna uma lista com as posições de todos os movimentos possíveis em uma dada direção, para uma dada peça. O parametro "includeBlocked",
    //  sendo false, indica que ao encontrar uma peça inimiga deve-se interromper o movimento sem poder comer a peça (caso do peão).
    protected ArrayList<int[]> untilBlockedPath(Piece piece, int yDirection, int xDirection, boolean includeBlocked, int limit){ ···


    // Analisa se um dado square está sob ataque de alguma peça do jogador passado como argumento
    public static boolean isSquareAttacked(Player player, Square square){ ···

    // Analisa quais dos movimentos de uma lista de movimentos são seguros, excluindo os demais.
    protected void getSafeMovements(ArrayList<int[]> moves, Piece piece){ ···


    // Analisa se um movimento é seguro
    boolean isSafeMovement(Piece piece, int[] movement, Player enemyPlayer){ ···
}
```

```java
public class QueenMovement extends Movement {

    public ArrayList<int[]> getValidMoves(boolean safeMovements, Piece piece){

        ArrayList<int[]> moves = new ArrayList<>();

        moves.addAll(untilBlockedPath(piece, 1, 0, true, 8));
        moves.addAll(untilBlockedPath(piece, -1, 0, true, 8));
        moves.addAll(untilBlockedPath(piece, 0, 1, true, 8));
        moves.addAll(untilBlockedPath(piece, 0, -1, true, 8));
        moves.addAll(untilBlockedPath(piece, 1, 1, true, 8));
        moves.addAll(untilBlockedPath(piece, -1, 1, true, 8));
        moves.addAll(untilBlockedPath(piece, 1, -1, true, 8));
        moves.addAll(untilBlockedPath(piece, -1, -1, true, 8));

        if (safeMovements){
            getSafeMovements(moves, piece);
        }

        return moves;
    }
```

```java
public abstract class Piece {

    protected Player player;

    protected Movement movement;  // Movement são os movimentos que determinada peça pode realizar.

    protected Square square;

    protected boolean wasMoved;

    protected String name;
```

```java
public class Queen extends Piece {

    public Queen(Player player, Square square){
        super(player, square);
        this.movement = new QueenMovement();
        name = player.getTeam().substring(0, 1)+"Queen";
    }
```