卒業研究報告書

題目

Java を用いたどうぶつしょうぎアプリの開発

指導教員

石水 隆　講師

報告者

14–1–037–0130

竹内 天斗

近畿大学理工学部情報学科

令和 2 年 2 月 3 日提出

概要

どうぶつしょうぎは 3x4 の盤面で「ひよこ」「ぞう」「キリン」「ライオン」の 4 種の駒を用いた，駒の動きを簡略化した将棋類の一種である．この将棋は低年齢層への将棋普及のために女流棋士である藤田麻衣子氏によって開発され，初心者用将棋盤駒として動物をモチーフにした将棋の盤駒がデザインされた．

どうぶつしょうぎのルールは通常の将棋より簡略化されており，また，字だけが書かれた本将棋の駒と異なり，どうぶつしょうぎの駒はかわいい動物のイラストが用いられている．このため，どうぶつしょうぎは低年齢層への知育玩具としても用いられている．

初心者が気軽にどうぶつしょうぎを遊ぶためには，1 人でも簡単に遊べるアプリがあることが望ましい．しかし，現在確認できているどうぶつしょうぎのアプリは，ユーザインタフェースは使いやすく仕上がっているものの，盤面の巻き戻しなどには対応していない [4]．そこで，本研究では知育玩具としてどうぶつつしょうぎ AI の CPU 及び対人での対戦が出来，なおかつユーザインタフェースを強化したアプリケーションを開発する．

目次

1　序論

1.1　背景

どうぶつしょうぎは，歩=ひよこと，それが成ったと金=にわとり，斜めに 1 マス移動出来るぞう，前後左右に 1 マス移動できるきりん，王将=ライオンの 4 種類の駒と 3*4 の盤面からなる将棋類の一種である．どうぶつしょうぎは 2008 年に女流棋士の北尾まどか氏によって考案されたボードゲームである．将棋に類似しているが，将棋と比べて非常に簡潔なルールになっている．どうぶつしょうぎは将棋の入門として幅広く普及しており，スマートフォンのアプリからニンテンドー Switch まで，様々な機器で遊ぶことが出来る．

1.2　二人零和有限確定完全情報ゲーム

どうぶつしょうぎは，二人零和有限確定完全情報ゲームに分類される．二人零和有限確定完全情報ゲームとは以下の条件を満たすゲームである．

- プレイヤーの数が二人
- プレイヤーの利害が対立
- ゲームの手番は有限
- ランダム要素は存在しない
- すべての情報が全プレイヤーに公開される

二人零和有限確定完全情報ゲームに分類されるボードゲームは，双方最善手を打った場合，先手勝ち，後手勝ち，引き分けのどれになるかはゲーム開始時点で決定しており，理論上，全ての可能な局面を解析することができれば最善の手を打つことができる．

しかし多くのボードゲームでは，可能な局面の総数が極めて大きいため，完全解析を行うことは不可能である．例を挙げれば，可能な局面数はリバーシが 10^{28} 通り，チェスが 10^{50} 通り，将棋が 10^{69} 通り，囲碁が 10^{170} 通り程度あるとされており，現在の計算機の性能を越えている．一方，可能な局面数が少ないゲームでは完全解析されているものもある．連珠は双方最善手を打った場合，47 手で先手が勝つ [7]．チェッカーは双方最善手を指すと引き分けとなる [8]．

局面数が大きいゲームについては，ゲーム盤をより小さいサイズに限定した場合の解析も行われている．サイズ 6x6 のリバーシでは，双方最善手を打つと 16 対 20 で後手勝ちとなる [10]．また，サイズ 4x4 の囲碁は双方最善手を打つと持碁 (引き分け)[11]，5x5 の囲碁は黒の 25 目勝ちとなる [12]．

将棋については，盤面のサイズおよび駒の種類を減らしたどうぶつしょうぎやアンパンマンはじめてしょうぎでは完全解析されており，どうぶつしょうぎは双方最善手を指すと 78 手で後手勝ち [1]，アンパンマンはじめてしょうぎは双方最善手を指すと引き分けとなる [14]．

1.3　どうぶつしょうぎに関する既知の結果

前節で述べた通り，どうぶつしょうぎは完全解析されており，双方が最善手を指すと，78 手で後手勝ちとなる [1]．よって最善手順を完全に覚えることができれば後手は必ず勝てるが，詰みまで 78 手と長く，途中の変化も多いためそれは現実的ではない．したがって，人間同士が対戦する分には，どうぶつしょうぎはまだま

だ有用な対戦ゲームであると言える．

どうぶつしょうぎはいくつかのアプリが作られている [4][5][6]．しかし既存のアプリには詰め将棋などの対局を行えるものは存在するものの，局面を自由に作ったり，盤面の巻き戻しに対応していたりするものは無く，必ずしも初心者に向いたものであるとは言えない．

1.4　本研究の目的

前節で述べた通り，現在確認できているどうぶつしょうぎのアプリ [4][5][6] は，ユーザインタフェースは使いやすく仕上がっているものの，盤面の巻き戻しなどには対応していない．そこで本研究ではその点も強化したアプリケーションを作成する．

1.5　本報告書の構成

本報告書の構成は以下の通りである．2 章で本研究の対象であるどうぶつしょうぎについて説明する．続く 3 章では，本研究で作成したどうぶつしょうぎのプログラムについて述べる．4 章から 8 章まででプログラムのクラス図を示し，9 章で CPU の戦略を述べ，10 章ではこのアプリの使い方について述べる．11 章にて考察を述べ，最後に 12 章で結論と今後の課題を述べる．

2　どうぶつしょうぎ とは

本章では，本研究の対象であるどうぶつしょうぎについて説明する．

2.1　どうぶつしょうぎとは

どうぶつしょうぎは女流棋士の北尾まどかがルールを，同じく女流棋士の藤田麻衣子がデザインをして，2008 年にその二人によって発表された．

どうぶつしょうぎは 3*4 の盤面で「ひよこ」「ぞう」「きりん」「ライオン」の 4 種の駒を用いて，玉将に相当するライオンを取るか，自分のライオンを相手の一陣目に移動させることで勝利となる．

2.2　どうぶつしょうぎのルール

本節では，どうぶつしょうぎのルールについて述べる．図 1 にどうぶつしょうぎの初期配置を示し，以下にどうぶつしょうぎのルールを示す．

キ	ニ	ヒ
	ワ	
	ひ	
ぞ	ラ	キ

図 1　どうぶつしょうぎ 初期配置

- 上段中央にひよこ，下段に左から順にぞう，ライオン，キリンの順に並べる．
- ひよこは相手の一陣目へ進むことで，にわとりに成ることが出来る．
- 駒の動かし方は，ひよこ，にわとり，ライオンはそれぞれ歩，と金，ライオン将と同じ．
- ぞうは斜めに 1 マス，キリンは上下左右に 1 マス移動出来る．
- 先手から交互に駒を動かし，相手のライオンを取るか，相手の一陣目にライオンを移動させれば勝ち．
- 将棋と同じく，自分の駒のない空いたマスに進むことが出来，相手の駒のある場所に進むと相手の駒を取ることが出来る．取った駒は持ち駒に出来る．
- 持ち駒は，自分の手番にお互いの駒がない場所に置くことが出来る．
- 打ち歩詰めや二歩などの反則はなく，千日手は引き分けとなる．

3　どうぶつしょうぎプログラム

本章では本研究で作成したどうぶつしょうぎのアプリケーションについて説明する．

本研究では，[2] の将棋プログラムをベースに，Java を用いてどうぶつしょうぎプログラムを作成した．付録に本研究で作成したプログラムのソースコードを示す．

本研究で作成したどうぶつしょうぎプログラムは，既存の CPU と対戦，および対人戦を行えるようになっている．

以下に本研究で作成したどうぶつしょうぎの各クラスについて説明する．

4　board パッケージ

4.1　Ban クラス

Ban クラスは盤を作成して駒を配置するクラスである．図 2 に Ban クラスのクラス図を示す．

Ban	# 盤を定義するクラス
- banarray : Koma[][]	# 盤面
+ Ban()	# コンストラクタ
+ Ban (banarray:Koma[][])	# コンストラクタ
+ clone() : Ban	# クローン生成
+ equal (ban:Ban) : boolean	# 同値判定
+ saveFile (filewriter : FileWriter) : void	# ファイルへの書き出し
+ loadFile (br:BufferdReader) : boolean	# ファイルからの読み込み
+ outputBan() : void	# 盤面出力
+ setFirstBan() : void	# 初期局面生成
+ setBanarray (a:int, b:int, koma:Koma) : void	# 指定した座標に駒をセット
+ getBanarray (a:int, b:int) : Koma	# 指定した座標の駒を得る

図 2　Ban クラスのクラス図

4.2 Board クラス

Board クラスは画面に関係するクラスである．図 3 に Board クラスのクラス図を示す．

4.3 CalcKyokumenFoul クラス

CalcKyokumenFoul クラスは千日手かどうかを調べるクラスである．図 4 に CalcKyokumenFoul クラスのクラス図を示す．

4.4 CalcMoveHuman クラス

CalcMoveHuman クラスは人間の入力に関するクラスである．図 5 に CalcMoveHuman クラスのクラス図を示す．

4.5 FreeKyokumen クラス

FreeKyokumen クラスは局面を自由に編集できるクラスである．図 6 に FreeKyokumen クラスのクラス図を示す．

4.6 Komadai クラス

Komadai クラスは持ち駒を管理する駒台のクラスである．図 7 に Komadai クラスのクラス図を示す．

4.7 Kyokumen クラス

Kyokumen クラスは局面に関するクラスである．図 8 に Kyokumen クラスのクラス図を示す．

4.8 MakeBoardList クラス

MakeBoardList クラスは Board のリストを作成するクラスである．図 9 に MakeBoardList クラスのクラス図を示す．

4.9 Move クラス

Move クラスは駒の移動に関するクラスである．図 10 に Move クラスのクラス図を示す．

4.10 ValueComparatorSente クラス

ValueComparatorSente クラスは先手の情報を比べるクラスである．図 11 に ValueComparatorSente クラスのクラス図を示す．

Board	# 盤面を定義するクラス
- kyokumen : Kyokumen	# 局面
- player : Player	# プレイヤー
- beforeMove : Move	# 直前の動き
- boardList : ArrayList <Board >	# 全ての手の候補のリスト
- value : Value	# 評価値
- maxDepth : int	# 読む手数の数
- kyokumenList : ArrayList <Kyokumen >	# 千日手チェックに必要
- kihu : Kihu	# 棋譜を保存する
+ Board(kyokumen:Kyokumen, player:Player, kyokumenList:ArrayList <Kyokumen >)	# コンストラクタ
+ Board ()	# コンストラクタ
+ Board (kyokumen:Kyokumen)	# コンストラクタ
+ readKihu() : void	# 棋譜読み込み
+ readKyokumen(loadFile:String) : void	# 局面読み込み
+ checkSennitite(kyokumenLis:ArrayList <Kyokumen >) : int	# 千日手か確認
+ checkTumi() : boolean	# 詰んでいるか確認
+ decideAllKomaMovePlace() : void	# 全ての駒の移動場所を決める
+ outputKyokumen() : void	# 局面を出力
+ moveBackKyokumen(move:Move) : void	# Move を得て戻る
+ geMoveNextBoard(move:Move) : Board	# move から次の局面の board を返す
+ moveNextKyokumen(move:Move) : void	# move を得て局面を進める
+ goNextBoard() : Board	# 次の局面に行く
+ goNextBoardFromSwing() : int	# 次の局面がなければ 0 を返す
+ sortBoardList() : void	# 評価値を並べ換える
+ decideNextBoardNoParent(depth:int) : Board	# boardList 作り, 次の局面を返す
+ decideNextBoardHaveParent(depth:int, boardParent:Board) : Board	# boardList 作り, 次の局面を返す
+ compareBoard(b1:Board,b2:Board) : Boolean	# b1 の評価値が良ければ true
+ selectBoardList() : void	# 勝っている board を選択
+ makeBoardList() : void	# 全ての手のリストを作る
+ setValueThis() : void	# 評価値を読む
+ outputValueTest() : void	# 評価値を出力
+ getTebans() : String	# 手番を返す
+ getTebanOpposites() : String	# 相手の手番を返す
+ getBeforeMove() : Move	# 前回の動きを返す
+ getValue() : Value	# 評価値を返す
+ setBeforeMove(move:Move) : void	# 前回の動きを読む
+ getKyokumen() : Kyokumen	# 局面を返す
+ getBoardList() : ArrayList <Kyokumen >	# boardList を返す
+ getTeban() : int	# 手番を返す
+ setPlayerSente(sente:boolean) : void	# 先手のプレイヤーを読む
+ setPlayerGote(gote:boolean) : void	# 後手のプレイヤーを読む
+ getPlayerBoth() : String	# 両方のプレイヤーを返す
+ getTebanPlayer() : void	# 手番のプレイヤーを返す
+ getKyokumenList() : ArrayList <Kyokumen >	# 局面のリストを返す
+ getKihu() : Kihu	# 棋譜を返す
+ saveKihu(fileName:String) : void	# 棋譜を読み込む
+ deleteLastKyokumenList() : void	# 最後に保存した局面リストを削除する
+ containCp() : boolean	# プレイヤーに CP がいれば true
+ setKyokumen(kyokumen:Kyokumen) : void	# 局面を読み込む
+ setKihu(kihu:Kihu) : void	# 棋譜を読み込む

図 3　Board クラスの クラス図

CalcKyokumenFoul	# 千日手か調べるクラス
+ checkSennitite(kyokumenList:ArrayList <Kyokumen >,kyokumen:Kyokumen) : int	# 千日手か調べる
+ checkLionte(kyokumen:Kyokumen,teban:int) : boolean	# ライオン手であれば true

図 4　CalcKyokumenFoul クラスのクラス図

CalcMoveHuman	# 人間の入力のクラス
- kyokumen : Kyokumen	# 局面
+ CalcMoveHuman(kyokumen : Kyokumen)	# コンストラクタ
+ getMove() : Move	# 正しい入力の Move を返す
+ checkGoNextKyokumen(move:move) : boolean	# 次の局面に行けるか確認．行けるなら true
+ checkGoBeforePlace(beforeA:int, beforeB:int) : boolean	# beforeAB に駒があるか確認
+ makeMove() : Move	# Move クラスを作る
+ getTeban() : void	# 手番を返す

図 5　CalcMoveHuman クラスのクラス図

FreeKyokumen	# 局面を編集するクラス
+ FreeKyokumen(kyokumen : Kyokumen)	# コンストラクタ
+ getKyokumen(teban:int) : Kyokumen	# 局面を返す
+ move(beforePlace:int, afterPlace:int) : Koma	# 駒を動かす．駒台に駒を置く

図 6　FreeKyokumen クラスのクラス図

Komadai	# 駒台を定義するクラス
- komadai : int[]	# 駒台の駒のリスト
- teban : int	# 手番
+ Komadai(teban:int)	# コンストラクタ
+ clone() : Komadai	# クローン生成
+ equal (komadaiN:Komadai) : boolean	# 同値判定
+ saveFile (filewriter : FileWriter) : void	# ファイルへの書き出し
+ loadFile (br:BufferdReader) : boolean	# ファイルからの読み込み
+ getAllPointTeban() : int	# 駒台にある駒の合計ポイントを返す
+ getKomadaiAllKoma() : ArrayList <Koma >	# 駒台の全ての駒のリストを返す
+ outputKomadi() : void	# 駒台の駒を出力
+ decreaseKomadai(koma:Koma) : void	# 駒を打って減らす
+ decreaseKomadaiKomaNumber(komaNumber:int) : void	# 駒を打って数を減らす
+ setKomadai(koma:Koma) : void	# 駒を駒台に入れる
+ getKomadai(place:int) : Koma	# place に駒台があれば返す
+ getKomaNumber(place:int) : int	# 駒の数を返す
+ setTeban(teban:int) : void	# 手番を読み込む
+ getTeban() : void	# 手番を返す
+ getTebanS() : String	# String で手番を返す
+ changeNS(n:int) : String	# n をそれぞれの駒に変える

図 7　Komadai クラスのクラス図

Kyokumen	# 局面を定義するクラス
- senteKomadai : Komadai	# 先手の駒台
- goteKomadai : Komadai	# 後手の駒台
- ban : Ban	# 盤
- teban : int	# 手番 先手=1，後手=2
- komaArray : ArrayList <Koma >	# 全ての駒のリスト
+ Kyokumen(ban:Ban,senteKomadai:Komadai, goteKomadai:Komadai,teban:int)	# コンストラクタ
+ Kyokumen()	# コンストラクタ
+ clone() : Kyokumen	# クローン生成
+ equal (kyokumen:Kyokumen) : boolean	# 同値判定
+ cloneAllKoma() : void	# 全ての駒のクローン生成
+ getTebanKomaListBan(teban:int) : ArrayList <Koma >	# 自分の盤の駒のリストを返す
+ getTebanKomaListAll(teban:int) : ArrayList <Koma >	# 自分の全ての駒のリストを返す
+ getPlaceLion(teban:int) : int	# 手番のライオンの場所を返す
+ decreaseMotigoma(koma:Koma) : void	# 打った駒を受け取り駒台の駒を減らす
+ moveBackKyokumen(move:Move) : void	# move をもらって前の局面へ
+ moveNextKyokumen(move:Move) : void	# move をもらって次の局面へ
+ changeTeban() : void	# 手番を変える
+ decideAllKomaMovePlace() : void	# 全ての駒の動く場所を決める
+ addKomaArray(koma:Koma) : void	# 引数の駒を ArrayList に追加する
+ removeKomaArray(koma:Koma) : void	# 引数の駒を ArrayList から削除する
+ setKomaArray() : void	# 駒の ArrayList を作成
+ outputKyokumen() : void	# 局面を出力する
+ saveKyokumenFromFileWriter(fileWriter:FileWriter) : void	# fileWriter を受け取り局面をファイルに保存する
+ saveKyokumen(saveFile:String) : void	# 局面をファイルに保存する
+ readKyokumenFromFirstKyokumen(br:BufferedReader) : boolean	# 局面を読み込む
+ loadKyokumen(loadFile:String) : boolean	# 局面を読み込む
+ getKomaFromPlace(placeA:int,placeB:int) : Koma	# 入力の場所の駒を返す
+ setKomaFromPlace(placeA:int,placeB:int) : void	# 入力の場所に駒をセットする
+ getSenteKomadaiKoma(placeB:int) : Koma	# 先手の駒台の駒を返す
+ getGoteKomadaiKoma(placeB:int) : Koma	# 後手の駒台の駒を返す
+ getSenteKomadai() : Komadai	# 先手の駒台を返す
+ getGoteKomadai() : Komamadai	# 後手の駒台を返す
+ getKomadai(teban:int) : Komadai	# 駒台を返す
+ getBanarray (a:int, b:int) : Koma	# 指定した座標の駒を得る
+ getBanarray (place:int) : Koma	# 指定した座標の駒を得る
+ getTeban() : int	# 手番を返す
+ getKomadArray() : ArrayList <Koma >	# 駒のリストを返す
+ getBan() : Ban	# 盤を返す
+ getTebanString() : String	# 手番を String で返す
+ setBanKomadai(ban:Ban,sente:Komadai,gote:Komadai) : void	# 盤と駒台をセットする

図 8　Kyokumen クラスのクラス図

MakeBoardList	# 盤を定義するクラス
+ getNextMoveListOnlyLionte(kyokumen:Kyokumen) : ArrayList <Move >	# ライオン手の局面を作る
+ getNextMoveListNoFoul(kyokumen:Kyokumen) : ArrayList <Move >	# 次に行ける Move クラスのリストを返す
+ getNextBoardList(board:Board) : ArrayList <Board >	# 次の手の Board のリストを返す
+ makeMoveList(kyokumen:Kyokumen) : ArrayList <Move >	# kyokumen から moveList を作る

図 9　MakeBoardList クラスのクラス図

Move	# 駒の動きを定義するクラス
- beforePlaceA : int	# 2 三の 2, 駒台なら 10, 20
- beforePlaceB : int	# 2 三の三
- AfterPlaceA : int	# 移動後の座標 A
- AfterPlaceB : int	# 移動後の座標 B
- naru : boolean	# true で成る
- getKoma : int	# 取った駒．取っていなければ 0
+ Move()	# コンストラクタ
+ Move(move:Move)	# コンストラクタ
+ Move(beforeA:int, beforeB:int, afterA:int, afterB:int)	# コンストラクタ
+ clone() : Move	# クローン生成
+ equalsMove(move:Move) : boolean	# 同値判定
+ checkNaru(koma:Koma) : boolean	# 駒が成れるか判断
+ ableNaru(teban:int) : boolean	# 駒が成れるか判断
+ outputMoveTest() : void	# move の出力のテスト
+ outputMoveKihu() : void	# 棋譜を保存するときのアウトプット
+ getMoveKihu() : String	# 棋譜を保存するときの改行を含む文字列を返す
+ inputMove() : void	# move の入力
+ getInputMove(str:String) : Move	# 新しく move クラス作って返す
+ matchAfterPlace(a:int,b:int) : boolean	# 入力と同じ場所が afterPlace か確認
+ matchAfterPlace(place:int) : boolean	# 入力と同じ場所が afterPlace か確認
+ getBeforePlaceA() : int	# BeforePlaceA を返す
+ getBeforePlaceB() : int	# BeforePlaceB を返す
+ getAfterPlaceA() : int	# AfterPlaceA を返す
+ getAfterPlaceA() : int	# AfterPlaceB を返す
+ setBeforePlaceA(beforeA:int) : void	# 引数を beforePlaceA にセットする
+ setBeforePlaceB(beforeB:int) : void	# 引数を beforePlaceA にセットする
+ setPlace(beforeA:int, beforeB:int, after:int) : void	# 引数をそれぞれの place にセットする
+ setAfterPlaceA(afterA:int) : void	# 引数を afterPlaceA にセットする
+ setAfterPlaceB(afterB:int) : void	# 引数を afterPlaceB にセットする
+ setAfterPlace(afterPlace:int) : void	# 引数をそれぞれ afterPlace にセットする
+ getNaru() : boolean	# 成りか不成かの状態を boolean で返す．成っていれば true
+ getGetKoma() : int	# getKoma を返す
+ setNaru(naru:boolean) : void	# 成りか不成かの状態をセットする．成っていれば true
+ setGetKoma(koma:int) : void	# getKoma に引数の駒をセットする
+ getAfterPlaceMix() : int	# afterPlace を合わせたものを返す

図 10　Move クラスのクラス図

ValueComparatorSente	# 先手の評価値のクラス
+ compare(b1:Board,b2:Board) : int	# b1 と b2 の評価値の比較を行う
+ compareBoardSente(b1:Board,b2:Board) : boolean	# b1 と b2 で評価値の比較を行い, b1 が大きければ true

図 11　ValueComparatorSente クラスのクラス図

4.11　ValueComparatorGote クラス

ValueComparatorGote クラスは後手の情報を比べるクラスである．図 12 に ValueComparatorGote クラスのクラス図を示す．

ValueComparatorGpte	# 後手の評価値のクラス
+ compare(b1:Board,b2:Board) : int	# b1 と b2 の評価値の比較を行う
+ compareBoardGote(b1:Board,b2:Board) : boolean	# b1 と b2 で評価値の比較を行い, b1 が小さければ true

図 12　ValueComparatorGote クラスのクラス図

5　calcValue パッケージ

5.1　LikelyToBeTokenValue クラス

LikelyToBeTokenValue クラスは自分の駒の動けるところに相手の駒があるか確認し，その評価値を決めるクラスである．図 13 に LikelyToBeTokenValue クラスのクラス図を示す．

LikelyToBeTokenValue	# 駒の動ける所に相手の駒があるか確認し，評価値を決めるクラス
+ LikelyToBeTokenValue(kyokumen:Kyokumen)	# コンストラクタ
+ calcValueL() : void	# 評価値を決める

図 13　LikelyToBeTokenValue クラスのクラス図

5.2　LossKomaValue クラス

LossKomaValue クラスは駒を失う評価値を決めるクラスである．図 14 に LossKomaValue クラスのクラス図を示す．

LossKomaValue	# 駒を失う評価値を決めるクラス
+ LossKomaValue(kyokumen:Kyokumen)	# コンストラクタ
+ calcValue() : void	# 評価値を決める

図 14　LossKomaValue クラスのクラス図

5.3 MovePlaceValue クラス

MovePlaceValue クラスは移動する際の評価値のクラスである．図 15 に MovePlaceValue クラスのクラス図を示す．

MovePlaceValue	# 駒を動かす際の評価値を決めるクラス
+ MovePlaceValue(kyokumen:Kyokumen)	# コンストラクタ
+ calcValue() : void	# 評価値を決める

図 15　MovePlaceValue クラスのクラス図

5.4 ParentValue クラス

ParentValue クラスは Kyokumen クラスの評価値を決める親クラスである．図 16 に ParentValue クラスのクラス図を示す．

ParentValue	# Kyokumen クラスの評価値を決める親クラス
- kyokumen : Kyokumen	# 局面
- valueBoolean : bolean	# 勝敗が決まれば true
- valueInt : int	# 評価値の数値
+ ParentValue(kyokumen : Kyokumen)	# コンストラクタ
+ calcValue() : void	# 評価値を決める
+ getValueBoolean() : boolean	# valueBoolean を返す
+ getValueInt() : int	# int 型で評価値を返す
+ getKyokumen() : Kyokumen	# 局面を返す
+ getTeban() : Int	# 手番を返す
+ setValueInt(int valueInt) : void	# 評価値を valueInt にセットする
+ setValueBoolean(valueBoolean:bolean) : void	# valueBoolean に引数をセットする

図 16　ParentValue クラスのクラス図

5.5 SafetyLionValue クラス

SafetyLionValue クラスはライオンの安全度を評価するクラスである．図 17 に SafetyLionValue クラスのクラス図を示す．

MovePlaceValue	# ライオンの安全度を評価するクラス
+ SafetyLionValue(kyokumen:Kyokumen)	# コンストラクタ
+ calcValue() : void	# 評価値を決める
+ makeValue(teban:int) : int	# 手番のライオンの危険度を返す
- checkDiff(diff:int) : boolean	# 周囲に敵駒がいるか確認．1 マス以内にいれば true

図 17　SafetyLionValue クラスのクラス図

5.6 TadaValue クラス

TadaValue クラスはただで取れる駒の評価値を確認するクラスである．図 18 に TadaValue クラスのクラス図を示す．

TadaValue	# ただで取れる駒の評価値を確認するクラス
+ TadaValue(kyokumen:Kyokumen)	# コンストラクタ
+ checkEnemyKomaMove(move:Move) : boolean	# 相手の駒が move の移動先に効いているか確認する．効いていれば true

図 18　TadaValue クラスのクラス図

5.7 TumiCheck クラス

TumiCheck クラスは詰んでいるかを確認するクラスである．図 19 に TumiCheck クラスのクラス図を示す．

TumiCheck	# 詰んでいるか確認するクラス
+ checkTumiTeban(kyokumen:Kyokumen) : boolean	# 手番のライオンがその時詰んでいるかを確認する．0 手読み
+ checkTumiNTe(kyokumenKyokumen,n:int) : Move	# 局面で n 手以下の詰みがあるか調べる
+ checkTumiForLion(kyokumen:Kyokumen,n:int) : boolean	# ライオン手されている局面で n 手以下で詰まされるか調べる

図 19　TumiCheck クラスのクラス図

5.8 Value クラス

Value クラスは評価値を計算するクラスである．図 20 に Value クラスのクラス図を示す．

6 koma パッケージ

6.1 CalcKoma クラス

CalcKoma クラスは駒を作るクラスである．図 21 に CalcKoma クラスのクラス図を示す．

6.2 Hiyoko クラス

Hiyoko クラスはひよこの駒のクラスである．図 22 に Hiyoko クラスのクラス図を示す．

6.3 Kirin クラス

Kirin クラスはキリンの駒のクラスである．図 23 に Kirin クラスのクラス図を示す．

Value	# 評価値を計算するクラス
- value : int	# 評価値
- determe : bolean	# 勝負が決まっている時 true
+ Value()	# コンストラクタ
+ calcValuePresent(kyokumen:Kyokumen,kyokumenList: ArrayList <Kyokumen >kyokumenList, beforeMove:Move) : void	# 評価値を計算する
+ addValue(add:int) : void	# 評価値を追加する
+ getValue() : int	# 評価値を返す
+ getDeterme() : boolean	# determe を返す
+ setValue(value:int) : void	# 引数の評価値をセットする
+ setDeterme(determe:bolean) : void	# determe をセットする
+ setValue(value:Value) : void	# value と determe をセットする
+ setWin(teban:int) : void	# 勝ちが決まっているときに設定する
+ setLose(teban:int) : void	# 負けが決まっているときに設定する
+ getWin(teban:int) : boolean	# 勝ちが決まっていれば true

図 20　Value クラスのクラス図

CalcKoma	# 駒を作るクラス
+ KomaMakeLoadKoma(loadStr:String) : Koma	# ファイルから局面を読み込んだ時の読み込み
+ changeTebanIntFromString(str:String) : int	# String で手番を変える
+ makeKoma(int n,int teban) : Koma	# 番号から駒を作る
+ isBan(a:int,b:int) : boolean	# a*10+b が盤内にあるか確認
+ isBan(place:int) : boolean	# a*10+b が盤内にあるか確認

図 21　CalcKoma クラスのクラス図

Hiyoko	# ひよこのクラス
+ Hiyoko(tebanN:int)	# コンストラクタ
+ ableToMove() : boolean	# 動けるかを確認
+ ableToNaru() : boolean	# 成れるかを確認
+ getKomaNumber() : int	# 駒の番号を返す．ひよこは 1
+ getKomaName() : String	# 駒の名前を 1 文字で返す
+ decideMovePlace(kyokumen:Kyokumen) : void	# 動ける場所を決める

図 22　Hiyoko クラスのクラス図

Kirin	# キリンのクラス
+ Kirin(tebanN:int)	# コンストラクタ
+ getKomaNumber() : int	# 駒の番号を返す．キリンは 3
+ getKomaName() : String	# 駒の名前を 1 文字で返す
+ decideMovePlace(kyokumen:Kyokumen) : void	# 動ける場所を決める

図 23　Kirin クラスのクラス図

6.4 Koma クラス

Koma クラスは駒を定義するクラスである．図 24 に Koma クラスのクラス図を示す．

6.5 Lion クラス

Lion クラスはライオンのの駒のクラスである．図 25 に Lion クラスのクラス図を示す．

6.6 Niwatori クラス

Niwatori クラスは鶏の駒のクラスである．図 26 に Niwatori クラスのクラス図を示す．

6.7 Zou クラス

Zou クラスは象の駒のクラスである．図 27 に Zou クラスのクラス図を示す．

7 main パッケージ

7.1 Calc クラス

Calc クラスは文字列が数字か確認するクラスである．図 28 に Calc クラスのクラス図を示す．

7.2 Kihu クラス

Kihu クラスは棋譜を保存するクラスである．図 29 に Kihu クラスのクラス図を示す．

MainSwing クラスは Main クラスから画面を呼び出す実行クラスである．図 30 のような画面が表示される．

7.3 Main クラス

Main クラスは対局を管理する実行クラスである．

7.4 Player クラス

Player クラスは先手後手が人間か CP かを確認するクラスである．図 31 に Player クラスのクラス図を示す．

8 swing パッケージ

8.1 ActionKihuListener クラス

ActionKihuListener クラスはアクションイベントの棋譜に関するクラスである．図 32 に ActionKihuListener クラスのクラス図を示す．

Koma	# 駒を定義するクラス
- teban : int	# 手番
- placeA : int	# 1 三の 1
- placeB : int	# 1 三の三
- point : int	# 駒の価値を表すポイント
- pointSpecial : int	# 特殊なポイント
- movePlace : ArrayList <Integer >	# 動ける場所
- motigoma : boolean	# 持ち駒かどうか．違うなら false
+ Koma(tebanN:int)	# コンストラクタ
+ clone() : Komadai	# クローン生成
+ equal (koma:Koma) : boolean	# 同値判定
+ saveFile (filewriter : FileWriter) : void	# ファイルへの書き出し
+ ableToMove() : boolean	# 動けるかを確認
+ getNarazukoma() : Koma	# 不成の駒を作る
+ getNarikoma() : Koma	# 成り駒を作る
+ ableToNaru() : boolean	# 成れるかを確認
+ getKomaName() : String	# 駒の名前を 1 文字で返す
+ getTebanS() : String	# 全角 2 文字で駒の種類と先手後手を表す
+ outputKoma() : void	# 全角 2, 半角 1, ↑ ↓, 半角スペース 1 で出力する
+ outputTest() : void	# 出力のテスト
+ movePlaceClear() : void	# movePlace を一つ削除する
+ decideMovePlace(kyokumen:Kyokumen) : void	# 動ける場所を決める
+ containMovePlace(place:int) : boolean	# その場所に動けるか
+ outputMovePlace() : void	# 自分のいる場所、駒の名前、動ける場所を出力する
+ setTeban(tebanN:int) : void	# 手番を読み込む
+ getTeban() : void	# 手番を返す
+ setPlace(a:int,b:int) : void	# 引数の位置をセットする
+ setPoint(pointN:int) : void	# 引数のポイントをセットする
+ getPoint() : int	# 引数のポイントを返す
+ setPointSpecial(pointSpecialN:int) : void	# 引数の特殊なポイントをセットする
+ getPoint() : int	# 引数の特殊なポイントを返す
+ setMovePlace(movePlace:ArrayList <Integer >) : void	# 移動可能な場所をセットする
+ getMovePlace() : ArrayList <Integer >	# 移動可能な場所を返す
+ getMovePlaceCloneDeep() : ArrayList <Integer >	# MovePlace のクローンを作る
+ addMovePlace(place:int) : void	# 移動可能な場所を追加する
+ getPlaceA() : int	# placeA を返す
+ getPlaceB() : int	# placeB を返す
+ getKomaNumber() : int	# 駒の番号を返す
+ getMotigoma() : boolean	# 持ち駒かどうかを確認する．持ち駒なら true
+ getPlace() : int	# placeA と placeB を合わせたものを返す
+ getPointTeban() : int	# 手番の point を返す
+ changeteban() : void	# 手番を変える

図 24　Koma クラスのクラス図

Lion	# ライオンのクラス
+ Lion(tebanN:int)	# コンストラクタ
+ getKomaNumber() : int	# 駒の番号を返す．ライオンは 4
+ getKomaName() : String	# 駒の名前を 1 文字で返す
+ decideMovePlace(kyokumen:Kyokumen) : void	# 動ける場所を決める

図 25　Lion クラスのクラス図

Niwatori	# 鶏のクラス
+ Niwatori(tebanN:int)	# コンストラクタ
+ getKomaNumber() : int	# 駒の番号を返す．鶏は 11
+ getKomaName() : String	# 駒の名前を 1 文字で返す
+ decideMovePlace(kyokumen:Kyokumen) : void	# 動ける場所を決める

図 26　Niwatori クラスのクラス図

Zou	# 象のクラス
+ Zou(tebanN:int)	# コンストラクタ
+ getKomaNumber() : int	# 駒の番号を返す．象は 3
+ getKomaName() : String	# 駒の名前を 1 文字で返す
+ decideMovePlace(kyokumen:Kyokumen) : void	# 動ける場所を決める

図 27　Zou クラスのクラス図

Calc	# 文字列が数字か確認するクラス
+ isNumber(str:String) : boolean	# 文字列が数字かどうか判断する
+ changeTeban(teban:int) : int	# 手番を変える

図 28　Calc クラスのクラス図

8.2　ActionListenerMenu クラス

ActionListenerMenu クラスはアクションイベントのメニューのクラスである．図 33 に ActionListener-Menu クラスのクラス図を示す．

8.3　ActionListener クラス

ActionListener クラスはアクションイベントを受け取るためのクラスである．図 34 に ActionListener クラスのクラス図を示す．

8.4　BanPanel クラス

BanPanel クラスは盤のパネルを管理するクラスである．図 35 に BanPanel クラスのクラス図を示す．

Kihu	# 棋譜を保存するクラス
- kihuList : ArrayList <Move >	# 棋譜のリスト
- firstKyokumen : Kyokumen	# 初期局面
- lastKyokumen : Kyokumen	# 対局終了時の局面
+ Kihu()	# コンストラクタ
+ Kihu(kyokumen:Kyokumen)	# コンストラクタ
+ cloneKihu() : Kihu	# クローン生成
+ makeLastKyokumen() : void	# 対局終了時の局面を作る
+ readKihuFromFirstKyokumen(inputFile:String) : void	# 初期局面から棋譜を読み込む
+ inputKihu(inputFile:String) : void	# 棋譜を読み込む
+ saveKihuFromFirstKyokumen(saveFile:String) : void	# 棋譜ファイルを最初の局面から保存する
+ saveKihuFile(saveFile:String) : void	# 棋譜をファイルに保存する
+ outputKihu(inputFile:String) : void	# 棋譜を出力
+ addMove(move:Move) : void	# kihuList に駒の移動を追加
+ getKihuList() : ArrayList <Move >	# kihuList を返す
+ getLastMove() : Move	# 最後の駒の移動を返す
+ removeLastMove() : void	# 最後の駒の移動を削除
+ getFirstKyokumen() : Kyokumen	# 初期局面を返す
+ getLastKyokumen() : Kyokumen	# 最後の局面を返す

図 29　Kihu クラスのクラス図

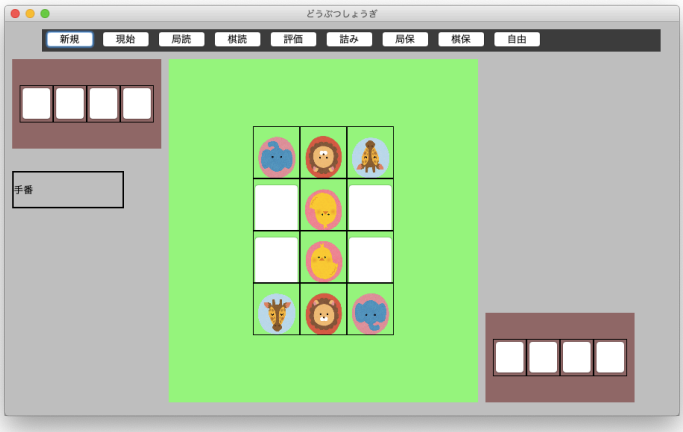

図 30　どうぶつしょうぎ 初期画面

8.5　Frames クラス

Frames クラスは盤や駒などのパネルの設定を行うクラスである．図 36 に Frames クラスのクラス図を示す．

Player	# 先手後手が人間か CP かを確認する
- playerSente : boolean	# 先手が人間か CP か確認する．CP なら true
- playerGote : boolean	# 後手が人間か CP か確認する．CP なら true
+ Player()	# コンストラクタ
+ decidePlayer() : void	# 先手後手のプレイヤーを決める
+ outputPlayer() : void	# プレイヤーを出力する
+ getPlayerBoth() : String	# 両プレイヤーの種類を返す
+ getPlayerSenteString() : String	# 先手のプレイヤーの種類を返す
+ getPlayerGoteString() : String	# 後手のプレイヤーの種類を返す
+ getPlayerSente() : boolean	# 先手のプレイヤーの種類を返す．CP なら true
+ getPlayerGote() : boolean	# 後手のプレイヤーの種類を返す．CP なら true
+ getPlayer(teban:int) : boolean	# 引数の手番のプレイヤーの種類を返す．CP なら true
+ setPlayerSente(playerSente:boolean) : void	# 先手のプレイヤーの種類をセットする
+ setPlayerGote(playerGote:boolean) : void	# 後手のプレイヤーの種類をセットする
+ containCp() : boolean	# 両プレイヤーのどちらかにに cp が含まれていれば true

図 31　Player クラスのクラス図

ActionKihuListener	# アクションイベントの棋譜に関するクラス
- board : Board	# 盤面
- frame : FrameS	# 設定に関するパネル
- state : StateGame	# ゲームの状態．1 なら対局中
+ ActionKihuListener(board:Board, frame:FrameS, state:StateGame)	# コンストラクタ
+ actionPerformed(e:ActionEvent) : void	# アクションイベントを起こす

図 32　ActionKihuListener クラスのクラス図

ActionListenerMenu	# アクションイベントのメニューに関するクラス
- board : Board	# 盤面
- frame : FrameS	# 設定に関するパネル
- state : StateGame	# ゲームの状態．1 なら対局中
- saveKyokumenName : String	# 局面を保存するときのファイルの名前
- readKyokumenName : String	# 局面を読み込むときのファイルの名前
- saveKihuName : String	# 棋譜を保存するときのファイルの名前
- readKihuName : String	# 棋譜を読み込むときのファイルの名前
+ ActionListenerMenu(board:Board, frame:FrameS, state:StateGame)	# コンストラクタ
+ actionPerformed(e:ActionEvent) : void	# アクションイベントを起こす
- ask(question:String, title:String): int	# 質問して yes なら 1，no なら 2
- matta() : void	# 局面を一手前に戻す
- setThisKyokumen() : void	# 保存していた局面をセットする
- setThisKihu() : void	# 保存していた棋譜をセットする
- setFirstKyokumen () : void	# 初期局面をセットする
- setPlayer() : boolean	# 先後を決める．決まらなければ false

図 33　ActionListenerMenu クラスのクラス図

ActionListener	# アクションイベントを受け取るためのクラス
- board : Board	# 盤面
- frame : FrameS	# 設定に関するパネル
- state : StateGame	# ゲームの状態．1 なら対局中
- freeKyokumen : FreeKyokumen	# 自由に局面を変える
+ ActionListener(board:Board, frame:FrameS, state:StateGame)	# コンストラクタ
+ actionPerformed(e:ActionEvent) : void	# アクションイベントを起こす
- checkFreeAfter(placeS:String): boolean	# 駒を移動させるとき，移動後の場所が正しいか確認する
- checkFreeBefore(placeS:String): boolean	# 駒を移動させるとき，移動後の場所が正しいか確認する
- makeMove(beforePlace:int, afterPlace:int) : Move	# move クラスを作る
- checkAfterPlace(placeS:String): boolean	# 駒を選択した後，動かせる場所か確認
- checkBeforeKoma(placeS:String): boolean	# 駒を選択する前に，動かせる駒か確認
- setFreeKyokumen() : void	# 自由に作った盤をセットする
- getFreeKyokumen() : FreeKyokumen	# 自由に作った盤を返す

図 34　ActionListener クラスのクラス図

BanPanel	# 盤のパネルを管理するクラス
- gridBagLayout : GridBagLayout	# レイアウト
- panelArray : JPanel[][]	# パネルのリスト
- him : window	# ウインドウ
+ BanPanel(her:Window)	# コンストラクタ
+ changePanel(a:int, b:int, kyokumen:Kyokumen, actionListener:ActionListener) : void	# パネルの変更
+ changeColor(a:int, b:int) : void	# 色の変更
+ setBanPanel() : void	# 盤面を呼び込む
- makeMasu(gbl:GridBagLayout) : void	# 駒を置くマスを作成する
+ makeMasuFromBan(ban:Ban, actionListener:ActionListener) : void	# 駒を置くマスを作成する

図 35　BanPanel クラスのクラス図

8.6　ImageIcon2 クラス

ImageIcon2 クラスはアイコンの画像を管理するクラスである．図 37 に ImageIcon2 クラスのクラス図を示す．

8.7　KihuPanel クラス

KihuPanel クラスは棋譜のパネルを管理するクラスである．図 38 に KihuPanel クラスのクラス図を示す．

8.8　KomadaiPanel クラス

KomadaiPanel クラスは駒台のパネルを管理するクラスである．図 39 に KomadaiPanel クラスのクラス図を示す．

Frames	# 盤や駒などのパネルの設定を行うクラス
- panelMenu : MenuPanel	# メニューパネル
- panelBan : BanPanel	# 盤のパネル
- panelSente : KomadaiPanel	# 先手の駒台パネル
- panelGote : KomadaiPanel	# 後手の駒台パネル
- labelTeban : JLavel	# 手番を表示
- actionListener:ActionListener	# アクションイベント
- actionListenerMenu:ActionListenerMenu	# メニューのアクションイベント
- actionKihuListener:ActionKihuListener	# 棋譜のアクションイベント
+ Frames(actionListener:ActionListener, actionListenerMenu:ActionListenerMenu, actionKihuListener:ActionKihuListener)	# コンストラクタ
+ setNestTe(teban:String) : void	# 局面が変わる際に手番のプレイヤーをセットする
+ changeLabelTeban(string:String) : void	# 手番のラベルを変更する
+ checkNextOrExit() : boolean	# cp 同士の対局の際に，次の手に行くか確認
+ outputMessage(mes:String,title:String) : void	# メッセージを出力する
+ outputTouryou(teban:String) : void	# 投了メッセージを出力する
+ changePanel(a:int, b:int, kyokumen:Kyokumen) : void	# パネルの変更
+ changeBottonColor(place:int) : void	# パネルの変更
+ startGame() : void	# 対局開始の距離
+ endGame() : void	# 対局終了の距離
+ setFreeBefore() : void	# 自由に作った局面をセット
+ setReadingKihu() : void	# 呼び込んだ棋譜をセット
+ swing2() : void	# どうぶつしょうぎの画面を表示
+ makePanel(panel:JPanel) : void	# パネルを作成する
+ setKyokumen(kyokumen:Kyokumen) : void	# 入力の局面を表示する
+ setKihuPanelVisible() : void	# 棋譜パネルを表示する
+ setKihuPanelnotVisible() : void	# 棋譜パネルを消す

図 36　Frames クラスのクラス図

ImageIcon2	# アイコンの画像を管理するクラス
+ ImageIcon2(f:String, own:Window)	# コンストラクタ

図 37　ImageIcon2 クラスのクラス図

KihuPanel	# 棋譜のパネルを管理するクラス
- buttonNext : JBotton	# 棋譜を 1 つ進めるボタン
- buttonBack : JBotton	# 棋譜を 1 つ戻すボタン
- buttonFirst: JBotton	# 最初の棋譜に戻すボタン
- buttonEnd : JBotton	# 最後の棋譜まで進めるボタン
+ setKihuPanel(actionKihuListener:ActionKihuListener) : void	# 棋譜パネルを表示する
+ setKihuPanelVisible() : void	# 棋譜パネルを表示する
+ setKihuPanelnotVisible() : void	# 棋譜パネルを消す

図 38　KihuPanel クラスのクラス図

BanPanel	# 盤のパネルを管理するクラス
- gridBagLayout : GridBagLayout	# レイアウト
- panelArray : JPanel[]	# パネルのリスト
- him : window	# ウインドウ
+ KomadaiPanel(her:Window)	# コンストラクタ
+ changePanel(b:int, komadai:Komadai, actionListener:ActionListener) : void	# パネルの変更
+ setKomadaiPanel(pW:int, pH:int) : void	# 駒台を呼び込む
- makeKomadai(gbl:GridBagLayout) : void	# 駒台を作成する
+ makeKomadaiPanel(ban:Ban, actionListener:ActionListener) : void	# 駒台のパネルを作成する

図 39　KomadaiPanel クラスのクラス図

8.9　MakeKomaImage クラス

MakeKomaImage クラスは駒の画像を読み込むクラスである．図 40 に MakeKomaImage クラスのクラス図を示す．

MakeKomaImage	# 駒の画像を読み込むクラス
+ makeKomaImage(komaNumber:int, teban:int, him:Window) : ImageIcon	# 駒の画像を読み込む

図 40　MakeKomaImage クラスのクラス図

8.10　MenuPanel クラス

MenuPanel クラスは画面上部のメニューボタンを表示させるクラスである．図 41 に MenuPanel クラスのクラス図を示す．

8.11　StateGame クラス

StateGame クラスは対局の状態を確認するクラスである．図 42 に StateGame クラスのクラス図を示す．

9　CPU の戦略

本章では，本研究で作成したどうぶつしょうぎアプリケーションが用いている戦略について述べる．

9.1　局面の評価値

本研究では，局面の評価値を算出するパラメタとして，以下の用をを用いる．

- 各駒に割り当てた価値
- ライオンの危険度
- 着手可能手数
- タダで取れる駒の評価

KihuPanel	# 画面上部のメニューボタンを表示させるクラス
- buttonNew : JBotton	# 新規対局ボタン
- buttonStart : JBotton	# 現在の局面から対局開始するボタン
- buttonTouryou : JBotton	# 投了するボタン
- buttonReadKyokumen : JBotton	# 局面を読み込むボタン
- buttonReadKihu : JBotton	# 棋譜を読み込むボタン
- buttonCalc : JBotton	# 評価値を計算するボタン
- buttonTumi : JBotton	# 詰んでいるか確認するボタン
- buttonSaveKyokumen : JBotton	# 局面を保存するボタン
- buttonSaveKihu : JBotton	# 棋譜を保存するボタン
- buttonMatta : JBotton	# 局面を巻き戻すボタン
- buttonFree : JBotton	# 自由に局面を作成するボタン
+ setGaming() : void	# ボタンを対局中の状態に変える
+ setNotGaming() : void	# ボタンを対局前の状態に変える
+ setFreeBefore() : void	# ボタンを局面作成中の状態に変える
+ setMenuPanel(actionListenerMenu:ActionListenerMenu) : void	# メニューパネルを作成する

図 41　MenuPanel クラスのクラス図

駒の価値は，駒の動ける場所からそれぞれひよこ，ぞう，キリン，ライオンの順で評価値をあげている．

ライオンの危険度は，ライオンの周辺 1 マスに駒が存在するか，自陣に相手の駒が入っている際に危険と判断する．

着手可能手の数は，盤面から駒を移動できる方向が多いほど，評価値をあげている．

タダで取れる駒の評価は，自分が駒を取れる場合に，その場所に移動できる駒がない場合に評価する．

9.2　αβ法

本研究で作成したアプリケーションの CPU は，一定手数先の局面を先読みし，その局面の評価値を求め，αβ法を用いて最も評価値が高くなる局面になる手を選択する．

αβ法とは探索アルゴリズムの一つである．αβ法では探索の際に，αカットとβカットという手法を用いる．これは，探索する必要がない枝を探索しないための工夫である．「α」「β」の範囲は，普通は「-∞」「＋∞」から始めるが，この幅を縮めることで高速に探索が行われる．ただし，その場合には，必ずしも最善手順及び最善手順での評価値が得られるとは限らなくなるが，返してくる値は，α以下であれはば得られる評価値の上限を示す値，β以上であれば，得られる評価値の加減を示す値になる．αβ法で，理論上の最高速度を得るには，探索の順番が完全に良い評価を返す順にソートされている必要がある [3]．図 43 にαβ法における，αカット，βカットの例を示す．D が 4 だとわかった時点で，C の値は 4 以下となり，B の値を超えないことが分かるためそれ以降を探索しない．また J が 8 だとわかった時点で，I の値は 8 以上になり，H の値を下回らないことが分かるため，それ以降を探索しない．しかしこのαβ法で最高速度を得るには，探索の順番が完全に良い評価を返す順にソートされている必要がある．そこで本研究では，まず各着手可能手に対してその手を 1 手指した局面を生成し，その評価値を求める．次に求めた評価値をソートし，評価値の高い手から順にαβ法で探索を行う．

StateGame	# 対局の状態を確認するクラス
- state : int	# 状態を保存
- beforePlace : int	# 初期の駒の位置
- kihu : Kihu	# 読み込んだ棋譜を保存する場所
- countTesuu : int	# 棋譜を読んだ際に，何手目か数える場所
- board : Board	# 盤面
- frame : FrameS	# 設定に関するパネル
+ StateGame(board:Board, frame:FrameS)	# コンストラクタ
+ setNotGaming() : void	# 画面を対局前の状態に変える
+ setBeforePlace(beforePlace:int) : void	# beforePlace を設定して手番が人間で駒を選択した状態にする
+ checkGamingPlayer() : boolean	# 対局中のプレイヤーを確認．人間が手番の時 true
+ checkChoseAfterPlace() : boolean	# 対局中，人間の手番かつ駒を選択していれば true
+ checkNotGaming() : boolean	# 対局中か確認．していなければ true
+ checkNothin() : boolean	# 対局も何もしていないか確認．していなければ true
+ checkChoseBeforeKoma() : boolean	# 対局中，人間の手番かつ何もしていなければ true
+ checkReadingKihu() : boolean	# 棋譜を読み込んでいる状態なら true
+ checkFreeBefore() : boolean	# 駒を自由に動かせる，かつ駒を選択していないなら true
+ checkFreeAfter() : boolean	# 駒を自由に動かせる，かつ駒を選択しているなら true
+ getBeforePlace() : int	# 前の状態の場所に移動する
+ getState() : void	# 現在の状態を返す
+ setGaming(player:boolean) : void	# 対局開始の状態にする
+ setFreeBefore() : void	# 駒を自由に動かせる，かつ駒を選択していない状態にする
+ setFreeAfter() : void	# 駒を自由に動かせる，かつ駒を選択している状態にする
+ setReadingKihu(tesuu:int) : void	# 棋譜を読み込んでいる状態にする
+ setTesuuAdd() : void	# 手数を追加する
+ setTesuuDecrease() : void	# 手数を減らす
+ setTesuuFirst() : void	# 最初の手数にする
+ setTesuuEnd() : void	# 最後の手数にする
+ getKihu() : Kihu	# 棋譜を返す
+ getCountTesuu() : int	# 手数を返す
+ getMoveNext() : Move	# 棋譜を次の手数に進める
+ getMoveBack() : Move	# 棋譜を前の手数に戻す
+ getBoard() : Board	# 盤面を返す

図 42　StateGame クラスのクラス図

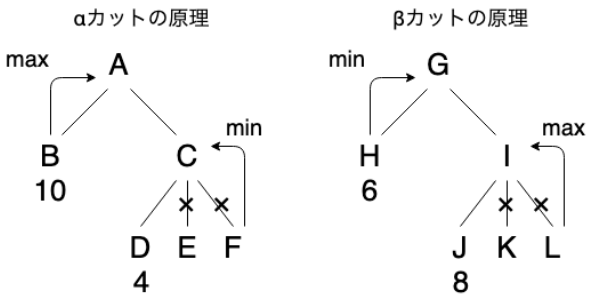

図 43　α カット，β カットの例

10　どうぶつしょうぎアプリケーションの使い方

本章では，本研究で作成したどうぶつ将棋アプリケーションの使い方を述べる．

まずは，MainSwing.java を起動させる．アプリケーションの起動画面を図 44 に示す．

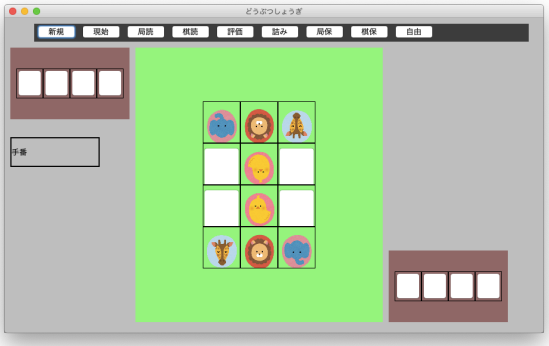

図 44　どうぶつしょうぎ 起動画面

画面上部のボタンを押すことによって，以下の操作を行うことができる．

- 新規：初期盤面で対局を開始する
- 現始：自由に作った局面や，読み込んだ局面から対局を開始する
- 局読：保存した局面を読み込む
- 棋読：保存した棋譜を読み込む
- 評価：現在の評価値を読み込む
- 詰み：局面が詰んでいるか確認する
- 局保：局面を保存する
- 棋保：棋譜を保存する
- 自由：自由に局面を設定する

ここで新規，現始ボタンを押すことで，先手後手のプレイヤーを選んでから対局が開始される．このとき遷移される対局画面を図 45 に示す．

駒を指す場合，自分の駒をクリックしてから，盤上の空白をクリックすることによって駒を移動できる．これは駒台から盤に指す場合も同じである．

また，この画面でも画面上部のボタンを押すことによって，以下の操作を行うことができる．

- 投了：対局を終了する
- 評価：現在の評価値を読み込む
- 詰み：局面が詰んでいるか確認する
- 局保：局面を保存する
- 棋保：棋譜を保存する
- 待た：待ったを掛けて局面を一つ前の自分の番に戻す

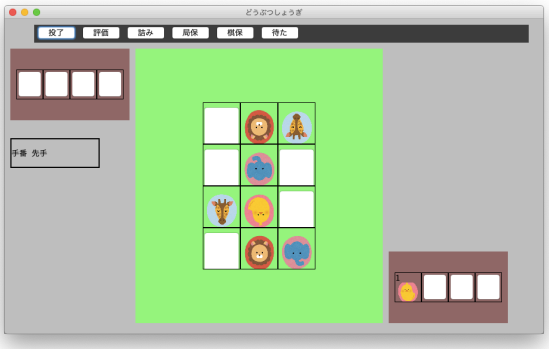

図 45　どうぶつしょうぎ 対局画面

11　考察

今回のアプリケーションはインタフェース面を強化して作成したことで，既存のアプリケーションと比べ，局面の巻き戻しや棋譜，局面の保存，読み込みが可能となり，好きな盤面から局面を開始し，詰め将棋のような対局も行えるようになった．

12　結論・今後の課題

本研究では，対人対 CPU での対局が出来るどうぶつしょうぎのアプリを作成した．結論としては，インタフェース面を強化したことで知育玩具として，低年齢層にも視覚的に分かりやすく作成出来たと考える．今後の課題としては，CPU 同士で対局を繰り返し最良の評価値となるパラメタの組み合わせを求め，より強力な AI を組んでいきたいと考えている．

謝辞

本研究を作成するにあたり、指導教員の石水隆講師には大変お世話になりました．ここ に感謝の意を表します．

参考文献

[1] 田中哲郎 :「どうぶつしょうぎ」の完全解析, 情報処理学会研究報告　Vol.2009-GI-22 No.3, pp.1—8 (2009)http://id.nii.ac.jp/1001/00062415/

[2] Java で将棋盤作成: https://stu345.hatenablog.com/entry/2019/09/04/002035

[3] 池泰弘 : Java 将棋のアルゴリズム [改定版], 工学社 (2016)

[4] どうぶつしょうぎ入門アプリ「はじめてのどうぶつしょうぎレッスン」, アイハット株式会社 (2016) https://apps.apple.com/jp/app/はじめてのどうぶつしょうぎレッスン-ルールから学べる入門アプリ/id1142677037

[5] どうぶつしょうぎ (公式), 株式会社ねこまど (2015) https://play.google.com/store/apps/details?id=jp.co.gmode.dobutsushogi

[6] みんなのどうぶつしょうぎ, Nintendo Switch, シルバースタージャパン, (2019) https://www.silverstar.co.jp/02products/dobutsushogi/switch/

[7] Janos Wagner and Istvan Virag : Solving renju, ICGA Journal, Vol.24, No.1, pp.30-35 (2001) http://www.sze.hu/~gtakacs/download/wagnervirag_2001.pdf

[8] Jonathan Schaeffer, Neil Burch, Yngvi Bjorsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Suphen : Checkers is solved, Science Vol.317, No,5844, pp.1518-1522 (2007) http://www.sciencemag.org/content/317/5844/1518.full.pdf

[9] 塩田好 :「アンパンマンはじめてしょうぎ」の完全解析, 情報処理学会関西支部 支部大会講演論文集 (2013).

[10] Joel Feinstein : Amenor Wins World 6x6 Championships!, Forty billion noted under the tree (July 1993), pp.6-8, British Othello Federation's newsletter., (1993), http://www.britishothello.org.uk/fbnall.pdf

[11] 清慎一, 川嶋俊 : 探索プログラムによる四路盤囲碁の解, 情報処理学会研究報告, GI 2000(98), pp.69–76 (2000), http://id.nii.ac.jp/1001/00058633/

[12] Eric C.D. van der Welf, H.Jaap van den Herik, and Jos W.H.M.Uiterwijk : Solving Go on Small Boards, ICGA Journal, Vol.26, No.2, pp.92-107 (2003).

[13] 田中哲朗 :「どうぶつしょうぎ」の完全解析, 情報処理学会研究報告, Vol.2009-GI-22 No.3, pp.1-8 (2009). http://id.nii.ac.jp/1001/00062415/

[14] 塩田好 :「アンパンマンはじめてしょうぎ」の完全解析, 情報処理学会関西支部 支部大会講演論文集 (2013).

付録

本研究で作成したどうぶつしょうぎプログラムのソースを以下に示す.

- Ban クラス

```
package board;

import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;

import koma.CalcKoma;
import koma.Hiyoko;
import koma.Kirin;
import koma.Koma;
import koma.Lion;
import koma.Zou;

public class Ban {
  private Koma banarray[][] = new Koma[4][5]; //盤面
  public Ban() {
    setFirstBan();
  }
  public Ban(Koma[][] banarray) {
    this.banarray = banarray;
  }
//クローン生成
  public Ban clone() {
    Koma array[][] = new Koma[4][5];
    for(int i=0;i<4;i++) {
      for(int j=0;j<5;j++) {
        array[i][j] = banarray[i][j];
      }
    }
    return new Ban(array);
  }
//同値判定
  public boolean equal(Ban ban) {
    for(int i=0;i<4;i++) {
      for(int j=0;j<5;j++) {
        if(banarray[i][j]==null) {
          if(ban.getBanarray(i, j)!=null) {
            return false;
          }
        }else {
          if(!banarray[i][j].equal(ban.getBanarray(i, j))) {
            return false;
```

```
        }
       }
      }
    }
    return true;
  }
//ファイルへの書き出し
  public void saveFile(FileWriter fileWriter) {
    try {
      for(int i=0;i<4;i++) {
        for(int j=0;j<5;j++) {
          if(banarray[i][j]==null) {
            fileWriter.write("0\n");
          }else {
            banarray[i][j].saveFile(fileWriter);
          }
        }
      }
    } catch (IOException e) {
      // TODO 自動生成された catch ブロック
      e.printStackTrace();
    }
  }
// ファイルの読み込み. 正しければ true
  public boolean loadFile(BufferedReader br) {
    try {
      String str = null;
      for(int i=0;i<4;i++) {
        for(int j=0;j<5;j++) {
          str = br.readLine();
          if(str.equals("0")) {
            banarray[i][j] = null;
          }else {
            Koma koma = CalcKoma.makeLoad_Koma(str);
            if(koma == null) {
              return false;
            }
            banarray[i][j] = koma;
          }
        }
      }
    } catch (IOException e) {
      // TODO 自動生成された catch ブロック
      e.printStackTrace();
      return false;
    }
    return true;
  }
// 盤面出力
```

```
  public void outputBan() {
    System.out.println("盤面を出力します。");
    Koma koma = null;
    for(int i=1;i<4;i++) {
      for(int j=4;j>0;j--) {
        koma = getBanarray(j, i);
        if(koma==null) {
          System.out.print("     ");
        }else {
          koma.outputKoma();
        }
      }
      System.out.println("");
    }
  }
//初期盤面生成
  public void setFirstBan() {
    //盤面, 全て null にする
    for(int i=1;i<4;i++) {
      for(int j=1;j<5;j++) {
        setBanarray(i, j, null);
      }
    }
    //駒配置
    //ひよこ
    Koma hiyoko1 = new Hiyoko(1);
    Koma hiyoko2 = new Hiyoko(2);
    setBanarray(2, 2, hiyoko2);
    setBanarray(2, 3, hiyoko1);
    //ぞう
    Koma zou1 = new Zou(1);
    Koma zou2 = new Zou(2);
    setBanarray(3, 1, zou2);
    setBanarray(1, 4, zou1);
    //キリン
    Koma kirin1 = new Kirin(1);
    Koma kirin2 = new Kirin(2);
    setBanarray(1, 1, kirin2);
    setBanarray(3, 4, kirin1);
    //ライオン
    Koma lion1 = new Lion(1);
    Koma lion2 = new Lion(2);
    setBanarray(2, 1, lion2);
    setBanarray(2, 4, lion1);
    //角
  }
// 指定した座標に駒をセット
  public void setBanarray(int a,int b,Koma koma){
    if (0<a && a<4 && 0<b && b<5) {
```

```
      banarray[a][b] = koma;
      if(koma!=null) {
        koma.setPlace(a,b);
       }
    }
  }
// 指定した座標の駒を得る
  public Koma getBanarray(int a,int b) {
    if (0<a && a<4 && 0<b && b<5) {
      return banarray[a][b];
    }
    return null;
  }
}
```

- Board クラス

```
package board;

import java.util.ArrayList;
import java.util.Collections;

import calcValue.Value;
import main.Kihu;
import main.Player;

public class Board {
  //手番は局面から取る
  private Kyokumen kyokumen; //局面
  private Player player; //プレイヤー
  private Move beforeMove;//直前の move, 棋譜を保存するときに使う。
  private ArrayList<Board> boardList;//候補のリスト, 全ての手のリスト
  private Value value;//評価値
  private static int maxDepth = 2;//何手まで読めるようにするか,1 手読むとは、自分が指して相手が指した手
の評価をする
  private ArrayList<Kyokumen> kyokumenList;//千日手チェックに必要, この board の持つ kyokumen は入って
いない
  private Kihu kihu;//棋譜を保存する。

  public Board(Kyokumen kyokumen,Player player,ArrayList<Kyokumen> kyokumenList,Kihu kihu) {
    this.kyokumen = kyokumen;
    this.player = player;
    this.kyokumenList = kyokumeLlist;
    boardList = new ArrayList<Board>();
    value = new Value();
    beforeMove = null;
    this.kihu = kihu;
  }
  // main not swing 用
```

```
  public Board(Kyokumen kyokumen,Player player,ArrayList<Kyokumen> kyokumenList) {
    this.kyokumen = kyokumen;
    this.player = player;
    this.kyokumenList = kyokumenList;
    boardList = new ArrayList<Board>();
    value = new Value();
    beforeMove = null;
    this.kihu = new Kihu(kyokumen);
  }

  public Board(Kyokumen kyokumen) {
    this.kyokumen = kyokumen;
    this.player = new Player();
    this.beforeMove = null;
    this.boardList = null;
    this.value = new Value();
    this.kyokumenList = new ArrayList<Kyokumen>();
    this.kihu = new Kihu(kyokumen);
  }
// 棋譜読み込み
  public void readKihu(String loadFile) {
    setFirst();
    kihu.readKihuFromFirstKyokumen(loadFile);
    kyokumen = kihu.getLirstKyokumen().clone();
  }
// 局面読み込み
  public void readKyokumen(String loadFile) {
    setFirst();
    kyokumen.loadKyokumen(loadFile);
  }
//初期盤面にする
  public void setFirst() {
    this.kyokumen = new Kyokumen();
    this.player = new Player();
    this.beforeMove = null;
    this.boardList = new ArrayList<Board>();
    this.value = new Value();
    this.kyokumenList = new ArrayList<Kyokumen>();
    this.kihu = new Kihu(kyokumen);
  }
//局面以外リセット
  public void resetExceptKyokumen() {
    kyokumen.decideAllKomaMovePlace();
    this.player = new Player();
    this.beforeMove = null;
    this.boardList = new ArrayList<Board>();
    this.value = new Value();
    this.kyokumenList = new ArrayList<Kyokumen>();
    this.kihu = new Kihu(kyokumen);
```

```
  }
//引数の board に変える
  private void setBoard(Board board) {
    this.kyokumen = board.getKyokumen();
    //this.player
    this.beforeMove = board.getBeforeMove();
    this.boardList = board.getBoardList();
    this.value = board.getValue();
    this.kyokumenList = board.getKyokumenList();
    this.kihu = board.getKihu();
  }
//局面を保存する
  public void saveKyokumen(String saveFile) {
    kyokumen.saveKyokumen(saveFile);
  }
// 千日手か確認
  public int checkSennitite(ArrayList<Kyokumen> kyokumenList) {
    return CalcKyokumenFoul.checkSennitite(kyokumenList, kyokumen);
  }
// 千日手か確認
  public int checkSennitite() {
    return CalcKyokumenFoul.checkSennitite(kyokumenList, kyokumen);
  }
// 詰んでいるか確認
  public boolean checkTumi() {
    Board bb = new Board(kyokumen.clone());
    Board nextBoard = bb.decideNextBoardNoParent(maxDepth);
    if(nextBoard==null) {
     //次に行く場所がない。詰み
      return true;
   }
    return false;
  }
// 全ての駒の movePlace を決める
  public void decideAllKomaMovePlace() {
    kyokumen.decideAllKomaMovePlace();
  }
// 局面を出力
  public void outputKyokumen() {
    System.out.println("value: "+value.getValue());
    kyokumen.outputKyokumen();
  }
// Move をもらって戻る
  public void moveBackKyokumen(Move move) {
    kyokumen.moveBackKyokumen(move);
  }
// move から次の局面になる board を返す
  public Board getMoveNextBoard(Move move) {
    kyokumen.clone().moveNextKyokumen(move);//ここで最終的に move が決まる
```

```
    if(boardList!=null && !boardList.isEmpty()) {
      for(Board b:boardList) {
        if(move.equalsMove(b.getBeforeMove())) {
          return b;
        }
      }
    }
    ArrayList<Kyokumen> kyokumenListN = new ArrayList<Kyokumen>();
    for(Kyokumen k:kyokumenList) {
      kyokumenListN.add(k);
    }
    //kyokumenListN.add(kyokumen.clone());
    Board board = new Board(kyokumen.clone(),player,kyokumenListN,kihu.cloneKihu());
    board.moveNextKyokumen(move);
    return board;
  }
// move をもらって局面を進める
  public void moveNextKyokumen(Move move) {
    kyokumen.moveNextKyokumen(move);//ここで最終的に move が決まる
    if(boardList!=null && !boardList.isEmpty()) {
      for(Board b:boardList) {
        if(move.equalsMove(b.getBeforeMove())) {
          setBoard(b);
          return;
        }
      }
    }
    kyokumenList.add(kyokumen.clone());
    kihu.addMove(move.clone());
    beforeMove = move;
  }
// 次の局面へいく
  public Board goNextBoard() {
    Board board = null;
    Move move = null;
    if(player.getPlayer(getTeban())) {
     //コンピュータ, 候補手の数だけ探索する。
      board = decideNextBoardNoParent(1);
    }else {
      //人間,
      //正しい move を受け取る。反則は関係ない。
      CalcMoveHuman calcMoveHuman = new CalcMoveHuman(kyokumen);
      move = calcMoveHuman.getMove();
      kyokumenList.add(kyokumen.clone());
      board = new Board(kyokumen, player,kyokumenList);
      board.moveNextKyokumen(move);
      if(board != null) {
        board.setBeforeMove(move);//いらない///
      }
```

```
    }
    return board;
  }
// 次の局面がなければ 0 返す
  public int goNextBoardFromSwing() {
    System.out.println("calc next te!");///
    Board board = decideNextBoardNoParent(1);
    if(board==null) {
      return 0;
    }
    setBoard(board);
    return 1;
  }
// boardList を先手なら評価値の高い順に並び替える (後手なら低い順)
  private void sortBoardList() {
    if(getTeban()==1) {
      Collections.sort(boardList, new ValueComparatorGote());
    }else {
      Collections.sort(boardList,new ValueComparatorSente());
    }
  }
// boardList を作り, 次の局面を返す
  public Board decideNextBoardNoParent(int depth) {
    //boardList は一回しか作らない
    if(boardList==null || boardList.isEmpty()) {
      makeBoardList();
    }
    selectBoardList();
    if(boardList==null || boardList.isEmpty()) {
      //負けている
      value.setDeterme(true);
      if(getTeban()==1) {
        value.setValue(-100);
      }else {
        value.setValue(100);
      }
      return null;
    }
    sortBoardList();
    if(depth==1) {
      System.out.println("sorted board list size: "+boardList.size());
    }
    if(depth>=maxDepth) {
      Board board = boardList.get(0);//1 手先で最善のやつ
      //ここで value の値も一つ下の値と同じに更新する。
      value.setValueAll(board.getValue());
      return board;
    }
    //a-b 法,boardList の中で一番いいのを選びたい。
```

```
    Board nextBoard = null;//次の局面でその先の選択肢があるか？
    Board boardExpect = null;//次になりそうな board
    int n = 1;
    for(Board board:boardList) {
      if(depth==1) {
        System.out.println("count "+n++);
      }
      if(boardExpect == null) {
        //初めての時
        if(!board.getValue().getDeterme()) {
          //board の評価値が決まっていない時はその下を探索する。評価値が決定されている時は下を探索する必要はない
          nextBoard = board.decideNextBoardNoParent(depth+1);
          if(nextBoard==null) {
            //勝っている.board を選べば次相手は負ける。board は負けている
            value.setWin(getTeban());
            return board;
          }
        }
        ///勝敗決まっているときはその先を計算する必要ないかもしれないので, 省略した方が良い??
        boardExpect = board;
      }else {
        //boardExpect がある時
        if(!board.getValue().getDeterme()) {
          //board の評価値が決まっていない時はその下を探索する。評価値が決定されている時は下を探索する必要はない
          nextBoard = board.decideNextBoardHaveParent(depth+1, boardExpect);
          if(nextBoard==null) {
            //勝っている.board を選べば次相手は負ける。board は負けている
            value.setWin(getTeban());
            return board;
          }
        }
        if(compareBoard(board, boardExpect)) {
          //board の方がいい時
          boardExpect = board;
        }
      }
    }
    value.setValueAll(boardExpect.getValue());
    return boardExpect;
  }
// boardList 作り, 次の局面を返す ab 法
  private Board decideNextBoardHaveParent(int depth,Board boardParent) {
    if(boardList==null || boardList.isEmpty()) {
      makeBoardList();
    }
    selectBoardList();
    if(boardList==null || boardList.isEmpty()) {
```

```
      //負けている
       value.setDeterme(true);
       if(getTeban()==1) {
         value.setValue(-100);
       }else {
         value.setValue(100);
       }
       return null;
     }
     sortBoardList();
     if(depth>=maxDepth) {
       Board board = boardList.get(0);//1 手先で最善のやつ
       //ここで value の値も一つ下の値と同じに更新する。
       value.setValueAll(board.getValue());
       return board;
     }
     //a-b 法,boardList の中で一番いいのを選びたい。親の評価値より自分の評価値がよければ終了!
     Board nextBoard = null;
     Board boardExpect = null;//次になりそうな board
     for(Board board:boardList) {
       if(boardExpect == null) {
         //初めての時
         if(!board.getValue().getDeterme()) {
           //board の評価値が決まっていない時はその下を探索する。評価値が決定されている時は下を探索する必要はない
           nextBoard = board.decideNextBoardNoParent(depth+1);
           if(nextBoard==null) {
             //勝っている.board を選べば次相手は負ける。board は負けている
             value.setWin(getTeban());
             return board;
           }
         }
         ///勝敗決まっているときはその先を計算する必要ないかもしれないので, 省略した方が良い??
         //親の評価値より自分の評価値がよければ終了!
         if(compareBoard(board, boardParent)) {
           value.setValueAll(board.getValue());
           return board;
         }
         boardExpect = board;
       }else {
         //boardExpect がある時
         if(!board.getValue().getDeterme()) {
           //board の評価値が決まっていない時はその下を探索する。評価値が決定されている時は下を探索する必要はない
           nextBoard = board.decideNextBoardHaveParent(depth+1, boardExpect);
           if(nextBoard==null) {
             //勝っている.board を選べば次相手は負ける。board は負けている
             value.setWin(getTeban());
             return board;
```

```
        }
       }
       if(compareBoard(board, boardExpect)) {
        //board の方がいい時
        //親の評価値より自分の評価値がよければ終了!
        if(compareBoard(board, boardParent)) {
          value.setValueAll(board.getValue());
          return board;
        }
        boardExpect = board;
       }
      }
    }
    value.setValueAll(boardExpect.getValue());
    return boardExpect;
  }
// b1 の評価値が高ければ true
  private boolean compareBoard(Board b1,Board b2) {
    if(getTeban()==1) {
      return ValueComparatorSente.cmpareBoardSente(b1, b2);
    }else if(getTeban()==2) {
      return ValueComparatorGote.compareBoardGote(b1, b2);
    }
    System.out.println("error:teban in Board class compareBoard method");
    return true;
  }
// 勝っている board を選択
  private void selectBoardList() {
    ArrayList<Board> removeList = new ArrayList<Board>();
    Value value = null;
    for(Board board:boardList) {
      value = board.getValue();
      if(value.getDeterme() && value.getValue()!=0) {
       //勝敗が決まっている。
       //引き分けは除かない
       if(value.getWin(getTeban())) {
        //勝ち
        ArrayList<Board> winList = new ArrayList<Board>();
        winList.add(board);
        boardList = winList;
        return;
       }else {
        //負け
        removeList.add(board);
       }
      }
    }
    for(Board board:removeList) {
      boardList.remove(board);
```

```
    }
  }
  /**
   * 全ての手のリストを作る
   */
  private void makeBoardList() {
    boardList = MakeBoardList.getNextBoardList(this);
    //それぞれの board クラスでその局面での評価値決める
    for(Board board:boardList) {
      board.setValueThis();
    }
  }
// 評価値を読む
  public void setValueThis() {
    value.calcValuePresent(kyokumen,kyokumenList,beforeMove);
  }
//評価値を出力する
  public void outputValueTest() {
    value.calcValuePresent(kyokumen,kyokumenList,beforeMove);
    System.out.println("評価値は:"+value.getValue());
  }
// 手番を返す
  public String getTebanS() {
    if(getTeban()==1) {
      return "先手";
    }else if(getTeban()==2){
      return "後手";
    }
    return "error";
  }
// 相手の手番を返す
  public String getTebanOppositeS() {
    if(getTeban()==2) {
      return "先手";
    }else if(getTeban()==1){
      return "後手";
    }
    return "error";
  }
// 一手前の駒の動きを返す
  public Move getBeforeMove() {
    return beforeMove;
  }
// 評価値を返す
  public Value getValue() {
    return value;
  }
// 一手前の駒の動きを読む
  public void setBeforeMove(Move move) {
```

```
    this.beforeMove = move;
  }
// 局面を消返す
  public Kyokumen getKyokumen() {
    return kyokumen;
  }
// 盤面のリストを返す
  public ArrayList<Board> getBoardList(){
    return boardList;
  }
// 手番を返す
  public int getTeban() {
    return kyokumen.getTeban();
  }
// 先手プレイヤーを読む
  public void setPlayerSente(boolean sente) {
    this.player.setPlayerSente(sente);
  }
// 後手プレイヤーを読む
  public void setPlayerGote(boolean gote) {
    this.player.setPlayerGote(gote);
  }
// 両方のプレイヤーを返す
  public String getPlayerBoth() {
    return this.player.getPlayerBoth();
  }
// 手番のプレイヤーを返す
  public boolean getTebanPlayer() {
    if(getTeban()==1) {
      return this.player.getPlayerSente();
    }
    return this.player.getPlayerGote();
  }
// 局面のリストを返す
  public ArrayList<Kyokumen> getKyokumenList(){
    return kyokumenList;
  }
// 棋譜を返す
  public Kihu getKihu() {
    return kihu;
  }
// 棋譜を保存する
  public void saveKihu(String fileName) {
    kihu.saveKihuFromFirstKyokumen(fileName);
  }
// 最後に保存した局面を削除する
  public void deleteLastKyokumenList() {
    kyokumenList.remove(kyokumenList.size()-1);
  }
```

```
  /**
   * プレイヤーに cp が含まれていれば true
   */
  public boolean containCp() {
    return player.containCp();
  }
// 局面を読み込む
  public void setKyokumen(Kyokumen kyokumen) {
    this.kyokumen = kyokumen;
  }
// 棋譜を読み込む
  public void setKihu(Kihu kihu) {
    this.kihu = kihu;
  }
}
```

- CalcKyokumenFoul クラス

```
package board;

import java.util.ArrayList;

import koma.Koma;
import main.Calc;

public class CalcKyokumenFoul {
    /**
     * 千日手がかどうかを調べる
     * @return 0:千日手ではない,1:千日手,2; 連続ライオン手の千日手手番がライオン手している,3:連続ライオン手の千日手手番がライオン手されている
     */
    public static int checkSennitite(ArrayList<Kyokumen> kyokumenList,Kyokumen kyokumen) {
      //kyokumenList に同じ局面があるか調べる。
      if(kyokumenList==null || kyokumenList.isEmpty()) {
        return 0;
      }
      int sameKyokumenCount = 0;
      for(Kyokumen k:kyokumenList) {
        if(k.equal(kyokumen)) {
          sameKyokumenCount++;
        }
      }
      if(sameKyokumenCount<4) {
        return 0;
      }
      for(Kyokumen k:kyokumenList) {
        k.outputKyokumen();
      }
      //連続ライオン手が無いか調べる, 手番の玉にライオン手がかかっているか
```

```
  if(checkLionte(kyokumen, kyokumen.getTeban())) {
   //手番の玉にライオン手がかかっている。2 手ずつ戻ってライオン手が連続しているか調べる。
    int number = kyokumenList.size()-2;
    Kyokumen kyokumenS = null;
    while(true) {
      kyokumenS = kyokumenList.get(number);
      if(kyokumenS.equal(kyokumen)){
       //連続ライオン手のまま一つ前の同じ局面まで来ている
        return 3;
      }
      if(!checkLionte(kyokumenS, kyokumen.getTeban())) {
       //手がかかっていない局面があれば連続ライオン手では無い
        break;
      }
      number = number - 2;
      if(number < 0) {
        System.out.println("error: checkSennitite sennitite but no same kyokumen");
        return 0;
      }
    }
  }
  //連続ライオン手、手番がライオン手している。
  int number = kyokumenList.size()-1;//一つ前の局面
  Kyokumen kyokumenFirst = kyokumenList.get(number);
  if(checkLionte(kyokumenFirst, kyokumenFirst.getTeban())) {
    Kyokumen kyokumenS = null;
    number = number - 2;
    while(true) {
      kyokumenS = kyokumenList.get(number);
      if(kyokumenS.equal(kyokumenFirst)){
       //連続ライオン手のまま一つ前の同じ局面まで来ている
        return 2;
      }
      if(!checkLionte(kyokumenS, kyokumenS.getTeban())) {
       //一つ前の局面で相手玉にライオン手がかかっていない。
        break;
      }
      number = number - 2;
      if(number < 0) {
        System.out.println("error: checkSennitite sennitite but no same kyokumen");
        return 0;
      }
    }
  }
  return 1;
}
/**
 * 手番にライオン手がかかっているか確認し、ライオン手がかかって入れば true
 * ライオン手がなければ false
```

```
     * @return
     */
    public static boolean checkLionte(Kyokumen kyokumen,int teban) {
      int placeLion = kyokumen.getPlaceLion(teban);
      int tebanR = Calc.changeTeban(teban);
      ArrayList<Koma> myKomaList = kyokumen.getTebanKomaListAll(tebanR);
      for(Koma koma:myKomaList) {
        for(int place:koma.getMovePlace()) {
          if(place==placeLion) {
            return true;
          }
        }
      }
      return false;
    }
}
```

- CalcMoveHuman クラス

```
package board;

import koma.Koma;

public class CalcMoveHuman {
  private Kyokumen kyokumen; //局面
  public CalcMoveHuman(Kyokumen kyokumen) {
    this.kyokumen = kyokumen;
  }
  //正しい入力の move を返す
  public Move getMove() {
    //move を作る。
    Move move = null;
    while(true) {
      move = makeMove();
      //move が正しいか判定。正しくなければやり直し。正しければそれを return
      if(checkGoNextKyokumen(move)) {
        //System.out.println("next kyokumen");
        break;
      }
      System.out.println("入力が正しくないです。もう一度やり直してください。");//76
    }
    //成るかどうかの確認
    Koma koma = kyokumen.getKomaFromPlace(move.getBeforePlaceA(), move.getBeforePlaceB());
    if(move.checkNaru(koma)) {
      move.setNaru(true);
    }
    return move;
  }
```

```
/**
 * 次の局面にいけるか確認する
 * 入力は Move クラス。見るのは beforePlace と afterPlace だけ。
 * いけたら true, いけなければ false
 */
private boolean checkGoNextKyokumen(Move move) {
  //before が正しいかチェック
  int beforeA = move.getBeforePlaceA();
  int beforeB = move.getBeforePlaceB();
  if(!checkGoBeforePlace(beforeA, beforeB)) {
    return false;
  }
  //afterPlace が正しいか確認する。Koma クラスの movePlace を見れば良い
  Koma koma = kyokumen.getKomaFromPlace(beforeA, beforeB);
  int a = move.getAfterPlaceA();
  int b = move.getAfterPlaceB();
  //持ち駒の時は盤が null かだけ見れば良い。
  if(koma.getMotigoma()) {
    //a,b が 0 はだめ
    if(a==0 || b==0) {
      return false;
    }
    Koma afterKoma = kyokumen.getBanarray(a,b);
    if(afterKoma==null) {
      return true;
    }
    return false;
  }
  if(koma.containMovePlace(a*10+b)) {
    return true;
  }
  System.out.println("after の入力が違います。checkGoNextKyokumen");
  return false;

}
/**
 * checkGoNextKyokumen メソッドで使う。beforeA,beforeB の場所に駒があるか確認する。
 * @return
 */
private boolean checkGoBeforePlace(int beforeA,int beforeB) {
  if(beforeA > 4) {
    //駒台のとき
    if((beforeA/10) == getTeban()) {
      //手番の駒台を指している。駒台に駒があるか確認する。
      if(kyokumen.getKomaFromPlace(beforeA, beforeB) != null) {
        //駒がある
        return true;
      }else {
        //駒がない
```

```
          System.out.println("error: beforepalce is not true in CalcMove class checkGoBeforePlace method");
          return false;
        }
      }else {
        //手番でない駒台を指しているので入力がおかしい。
        System.out.println("before の入力が違います。CalcMove checkGoBeforePlace");
        return false;
      }
    }else {
      //盤面の駒を動かすとき
      Koma koma = kyokumen.getBanarray(beforeA, beforeB);
      if(koma == null) {
        System.out.println("before の入力が違います。ban koma==nullCalcMove checkGoBeforePlace"+beforeA+beforeB);
        return false;
      }
      if(koma.getTeban()==getTeban()) {
        //手番の駒がある
        return true;
      }
      System.out.println("before の入力が違います。teban CalcMove checkGoBeforePlace"+beforeA+beforeB);
      return false;
    }
  }
  /**
   * move クラスを作る。
   * コンピュータか人間かによって作り方違う。
   * 駒を動かせる move を返す。人間の入力の時は二歩などの反則はあっていい。
   * @return
   */
  private Move makeMove() {
    if(getTeban()==1) {
      System.out.println("先手番です。");
    }else if(getTeban()==2){
      System.out.println("後手番です。");
    }
    Move move = new Move();
    move.inputMove();
    //成るかどうか確認していない。入力は間違っていてもいい
    return move;
  }
  private int getTeban() {
    return kyokumen.getTeban();
  }
}
```

- FreeKyokumen クラス

```
package board;
```

```
import koma.CalcKoma;
import koma.Koma;

//局面を自由に編集できるクラス
public class FreeKyokumen extends Kyokumen{
  public FreeKyokumen(Kyokumen kyokumen) {
    setBanKomadai(kyokumen.getBan().clone(),
      kyokumen.getSenteKomadai().clone(), kyokumen.getGoteKomadai().clone());
  }
  public Kyokumen getKyokumen(int teban) {
    Kyokumen kyokumen = new Kyokumen(getBan(),getSenteKomadai(),getGoteKomadai(),teban);
    kyokumen.setKomaArray();
    kyokumen.decideAllKomaMovePlace();
    return kyokumen;
  }
  /**
   * 駒を動かす
   * @param beforePlace
   * @param afterPlace
   * 駒台に置く時は取った駒返す
   */
  public Koma move(int beforePlace,int afterPlace) {
    //beforePlace の駒はなくす。駒台なら減らす
    Koma koma = getKomaFromPlace(beforePlace);
    if(koma == null) {
      System.out.println("error: free kyokumen move: "+beforePlace);
      return null;
    }
    if(beforePlace/10 < 10) {
      //動かす前は盤上の駒
      setKomaFromPlace(null, beforePlace/10, beforePlace%10);
    }else {
      decreaseMotigoma(koma);//前の場所を無くした。
    }
    //同じ場所なら成る反転させる。
    if(beforePlace==afterPlace) {
      if(koma.ableToNaru() || koma.getKomaNumber()>10) {
        int komaNumber = koma.getKomaNumber();
        if(komaNumber<10) {
          komaNumber += 10;
        }else {
          komaNumber -= 10;
        }
        Koma komaN = CalcKoma.makeKoma(komaNumber, koma.getTeban());
        setKomaFromPlace(komaN, afterPlace/10, afterPlace%10);
      }else {
        //成れない駒の時はなにもしない
        setKomaFromPlace(koma, afterPlace/10, afterPlace%10);
      }
```

```
      return null;
    }
    setKomaFromPlace(koma, afterPlace/10, afterPlace%10);
    if(afterPlace/10 < 10) {
      //盤上に移動
      return null;
    }
    return koma;
  }
}
```

- Komadai クラス

```
package board;

import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;

import koma.CalcKoma;
import koma.Koma;
import main.Calc;

public class Komadai {
  //0:なし,1:ひよこ,2:ぞう,3:キリン,4:ライオン
  private int komadai[] = new int[5];
  private int teban;
  public Komadai(int[] komadai,int teban) {
    this.komadai = komadai;
    this.teban = teban;
  }
  public Komadai(int teban) {
    for(int i=0;i<5;i++) {
      //最初は駒台に駒はない
      komadai[i] = 0;
    }
    this.teban = teban;
  }
  public Komadai clone() {
    int dai[] = new int[5];
    for(int i=0;i<5;i++) {
      dai[i] = komadai[i];
    }
    return new Komadai(dai,teban);
  }
```

```
 public boolean equal(Komadai komadaiN) {
   if(this.teban != komadaiN.getTeban()) {
     return false;
   }
   for(int i=0;i<5;i++) {
     if(komadai[i] != komadaiN.getKomaNumber(i)) {
       return false;
     }
   }
   return true;
 }
 /**
  * ファイルに保存する
  * @param fileWriter
  */
 public void saveFile(FileWriter fileWriter) {
   try {
     fileWriter.write(getTebanS()+"\n");
     for(int i=0;i<5;i++) {
       fileWriter.write(komadai[i] + ",");
     }
     fileWriter.write("\n");
   } catch (IOException e) {
     // TODO 自動生成された catch ブロック
     e.printStackTrace();
   }
 }
 /**
  * 正しく読み込めたら true
  * @return
  */
 public boolean loadFile(BufferedReader br) {
   try {
     //手番があっているか確認
     String str = br.readLine();
     if(Calc.isNumber(str)) {
       int teban = Integer.parseInt(str);
       if(this.teban != teban) {
         return false;
       }
     }else {
       return false;
     }
     str = br.readLine();
     String[] komadaiS = str.split(",", 0);
     if(komadaiS.length != 5) {
       System.out.println("駒台の長さが違います。");
       return false;
     }
```

```
      for(int i=0;i<5;i++) {
        if(Calc.isNumber(komadaiS[i])) {
          komadai[i] = Integer.parseInt(komadaiS[i]);
        }else {
          return false;
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
      return false;
    }
    return true;
  }
  /**
   * 駒台にある駒のポイントの合計を返す。
   * 先手ならプラス、後手ならマイナスで返す。
   * @return
   */
  public int getAllPointTeban() {
    Koma koma = null;
    int allPoint = 0;
    for(int i = 1;i<5;i++) {
      koma = getKomadai(i);
      if(koma != null) {
        allPoint += koma.getPointTeban() * komadai[i];
      }
    }
    return (int)(allPoint * 1.1);//持ち駒は持っていた方が特
  }
  /**
   * 駒台にある全ての駒のリストを返す。一つの駒は一回しか入らない
   * @return
   */
  public ArrayList<Koma> getKomadaiAllKoma(){
    ArrayList<Koma> komaList = new ArrayList<Koma>();
    Koma koma = null;
    for(int i=1;i<5;i++) {
      koma = getKomadai(i);
      if(koma != null) {
        komaList.add(koma);
      }
    }
    return komaList;

  }
// 駒台の出力
  public void outputKomadi() {
    if(teban==1) {
      System.out.println("先手の駒台の出力");
```

```
  }else {
    System.out.println("後手の駒台の出力");
  }
   for(int i=1;i<5;i++) {
    if(komadai[i] > 0) {
      System.out.print(changeNS(i)+":"+komadai[i]+"枚、");
    }
  }
   System.out.println("");
}

/**
 * 駒をうつ。減らす
 * @param koma
 */
 public void decreaseKomadai(Koma koma) {
   if(koma.getTeban() != teban){
     System.out.println("error: Komadai decreaseKomadai teban");
     return;//駒台が違う駒台
  }
   int n = koma.getKomaNumber();
   if(komadai[n]<1) {
     System.out.println("error: Komadai decreaseKomadai not have");
     return;
  }
   komadai[n] -= 1;
}
/**
 * 駒をうつ。減らす
 * @param koma
 */
 public void decreaseKomadaiKomaNumber(int komaNumber) {
   if(komaNumber<1 || komaNumber>4) {
     System.out.println("error: Komadai decreaseKomadaiKomaNumber");
     return;
  }
   if(komadai[komaNumber]<1) {
     System.out.println("error: Komadai decreaseKomadai not have");
     return;
  }
   komadai[komaNumber] -= 1;
}
/**
 * 駒を駒台に入れる。
 */
 public void setKomadai(Koma koma) {
  ///自由の時も使うから、手番が同じ駒の時もある
   int n = koma.getKomaNumber();
   if(n>10) {
```

```
      n -= 10;
    }
    komadai[n] += 1;
    //komaArray.add(koma);
  }
  /**
   * 入力 placeB に持ち駒があればそれを返す。
   * その駒が持ち駒になければ null を返す。
   * 実際に動かすわけではない
   * @param place
   * @return
   */
  public Koma getKomadai(int place) {
    if(komadai[place] < 1) {
      return null;
    }
    Koma koma = null;
    koma = CalcKoma.makeKoma(place, teban);
    koma.setMotigoma(true);
    koma.setPlace(teban*10, place);
    return koma;
  }
// 駒の番号を返す
  public int getKomaNumber(int place) {
    return komadai[place];
  }
// 手番を読む
  public void setTeban(int teban) {
    this.teban = teban;
  }
// 手番を返す
  public int getTeban() {
    return teban;
  }
// 手番を String で返す
  private String getTebanS() {
    if(teban==1) {
      return "1";
    }else if(teban==2) {
      return "2";
    }
    return "error";
  }
  private String changeNS(int n) {
    //n:1,2,3,4 をひよこ, 象, キリン, ライオンに変える 10 の位があれば成駒に
    String st;
    //全角で 2 文字
    switch (n) {
      case 0:
```

```
        st = "    ";
        break;
      case 1:
        st = "  ひ";
        break;
      case 2:
        st = "  象";
        break;
      case 3:
        st = "  キ";
        break;
      case 4:
        st = "  ラ";
        break;
      case 11:
        st = "  に";
        break;
      default:
        st = "error";
        break;
    }
    return st;
  }
}
```

- Kyokumen クラス

```
package board;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;

import koma.CalcKoma;
import koma.Koma;

/**
 * 局面を保存する。盤台と駒台
 * 駒 0:なし，1:ひよこ，2:ぞう，3:きりん，4:ライオン
 * 10 の位 0:普通，1:成駒（にわとり）
 * 100 の位 1:先手の駒，2:後手の駒
 */
public class Kyokumen {
  //盤の状態
   private Komadai senteKomadai;
```

```
  private Komadai goteKomadai;
  private Ban ban;
  private int teban;//1:先手 2:後手
  private ArrayList<Koma> komaArray;//全てのコマのリスト。駒を探しやすいように. 盤上にある駒だけ

  public Kyokumen(Ban ban,Komadai senteKomadai,Komadai goteKomadai,int teban) {
    this.teban = teban;
    this.ban = ban;
    this.senteKomadai = senteKomadai;
    this.goteKomadai = goteKomadai;
    komaArray = new ArrayList<Koma>();
    setKomaArray();
    cloneAllKoma();
    decideAllKomaMovePlace();///
  }
  public Kyokumen() {
   //入力がなければ初期局面
    ban = new Ban();
    senteKomadai = new Komadai(1);
    goteKomadai = new Komadai(2);
    teban = 1;
    komaArray = new ArrayList<Koma>();
    setKomaArray();
    decideAllKomaMovePlace();///
  }
// クローン生成
  public Kyokumen clone() {
    return new Kyokumen(ban.clone(),senteKomadai.clone(),goteKomadai.clone(),teban);
  }
  /**
   * @param kyokumen, 比較する局面クラス
   * @return true:同じ局面,false:違う局面
   */
  public boolean equal(Kyokumen kyokumen) {
    if(this.teban != kyokumen.getTeban()) {
      return false;
    }
    if(!this.senteKomadai.equal(kyokumen.getSenteKomadai())) {
      return false;
    }
    if(!this.goteKomadai.equal(kyokumen.getGoteKomadai())) {
      return false;
    }
    if(!this.ban.equal(kyokumen.getBan())) {
      return false;
    }
    return true;
  }
  /**
```

```
 * 全ての駒のクローン作って置き換える
 * 盤上と komaArray
 */
private void cloneAllKoma() {
  Koma newKoma = null;
  ArrayList<Koma> komaArrayNew = new ArrayList<Koma>();
  for(Koma koma:komaArray) {
    newKoma = koma.clone();
    komaArrayNew.add(newKoma);
    setKomaFromPlace(newKoma, newKoma.getPlaceA(), newKoma.getPlaceB());
  }
  komaArray = komaArrayNew;
}
/**
 * 自分の駒のリストを返す。
 * 盤上だけ
 * @return
 */
public ArrayList<Koma> getTebanKomaListBan(int teban){
  ArrayList<Koma> myKomaList = new ArrayList<Koma>();
  for(Koma koma:komaArray) {
    if(koma.getTeban() == teban) {
      //自分の駒。盤上の駒
      myKomaList.add(koma);
    }
  }
  return myKomaList;
}
/**
 * 自分の駒のリストを返す。
 * 持ち駒もリストに入れる。持ち駒の同じ駒は一枚だけ
 * @return
 */
public ArrayList<Koma> getTebanKomaListAll(int teban){
  ArrayList<Koma> myKomaList = getTebanKomaListBan(teban);//盤上の駒
  //駒台. 先手か後手
  for(Koma koma:getKomadai(teban).getKomadaiAllKoma()) {
    myKomaList.add(koma);
  }
  return myKomaList;
}
/**
 * 手番のライオンの場所を返す
 * @return
 */
public int getPlaceLion(int teban) {
  for(Koma koma:komaArray) {
    if(koma.getKomaNumber()==4) {
      if(koma.getTeban()==teban) {
```

```
        return koma.getPlace();
      }
    }
  }
  return 0;
}
/**
 * 打った駒を受け取って駒台のその駒を減らす。
 * @param koma
 */
public void decreaseMotigoma(Koma koma) {
  if(koma.getTeban()==1) {
    senteKomadai.decreaseKomadai(koma);
  }else if(koma.getTeban()==2) {
    goteKomadai.decreaseKomadai(koma);
  }else {
    System.out.println("error: Kyokumen decreaseMotigoma teban");
  }
}
/**
 * move もらって前の局面へ
 * 手番変える
 * @param move
 */
public void moveBackKyokumen(Move move) {
  //動いた駒を元の場所に戻す
  Koma moveKoma = getKomaFromPlace(move.getAfterPlaceA(), move.getAfterPlaceB());//動いた駒
  removeKomaArray(moveKoma);
  if(move.getNaru()) {
    //成っている時
    moveKoma = moveKoma.getNarazukoma();
  }
  if(move.getBeforePlaceA()<10) {
    //盤上の移動の時
    ban.setBanarray(move.getBeforePlaceA(), move.getBeforePlaceB(), moveKoma);
    addKomaArray(moveKoma);
  }else {
    //駒を打った時
    moveKoma.changeTeban();
    setKomaFromPlace(moveKoma,move.getBeforePlaceA() , move.getBeforePlaceB());
  }
  //動いた後の場所に元あった（なかった）駒を配置
  if(move.getGetKoma()==0) {
    //とっていない
    changeTeban();
    ban.setBanarray(move.getAfterPlaceA(), move.getAfterPlaceB(), null);
  }else {
    //駒を取っていた。
    Koma koma = CalcKoma.makeKoma(move.getGetKoma(), teban);
```

```
    ban.setBanarray(move.getAfterPlaceA(), move.getAfterPlaceB(), koma);
    int komaNumber = move.getGetKoma();
    if(komaNumber>9) {
      komaNumber -= 10;
    }
    //駒台減らす
    if(teban==1) {
      //後手番の駒台
      goteKomadai.decreaseKomadaiKomaNumber(komaNumber);
    }else {
      senteKomadai.decreaseKomadaiKomaNumber(komaNumber);
    }
    changeTeban();
    addKomaArray(koma);
  }
  decideAllKomaMovePlace();
}
/**
 * move をもらって次の局面へ
 * before,after
 * なるかどうかはここで確認する。
 * 手番変える
 */
public void moveNextKyokumen(Move move) {
  int a = move.getBeforePlaceA();
  int b = move.getBeforePlaceB();
  Koma koma = getKomaFromPlace(a, b);
  if(koma.getMotigoma()) {
    //持ち駒を打つとき。
    koma.setMotigoma(false);
    decreaseMotigoma(koma);//駒台減らす。
    a = move.getAfterPlaceA();
    b = move.getAfterPlaceB();
    ban.setBanarray(a, b, koma);
    addKomaArray(koma);
  }else {
    //盤上の駒を動かす
    ban.setBanarray(a, b, null);//動かす前の場所
    a = move.getAfterPlaceA();
    b = move.getAfterPlaceB();
    Koma getKoma = getBanarray(a,b);
    if(getKoma != null) {
      //駒を取っていたら駒台へ
      if(teban==1) {
        senteKomadai.setKomadai(getKoma);
      }else {
        goteKomadai.setKomadai(getKoma);
      }
      //取った駒は komaArray から消す
```

```
        komaArray.remove(getKoma);
        //move の getKoma を set
        move.setGetKoma(getKoma.getKomaNumber());
      }
      //なれるかの確認,
      if(move.getNaru()) {
        //成る時
        removeKomaArray(koma);
        koma = koma.getNarigoma();
        addKomaArray(koma);
      }
      ban.setBanarray(a, b, koma);
    }
    changeTeban();
    decideAllKomaMovePlace();
  }
  private void changeTeban() {
    if(teban==1) {
      teban = 2;
    }else if(teban==2) {
      teban = 1;
    }
  }
// 全ての駒の移動場所を決める
  public void decideAllKomaMovePlace() {
    for(Koma koma:komaArray) {
      koma.movePlaceClear();
      koma.decideMovePlace(this);
    }
  }
// 駒を追加する
  public void addKomaArray(Koma koma) {
    if(koma.getPlace()>34) {
      System.out.println("addKomaArray miss");
    }
    komaArray.add(koma);
  }
// 駒を削除する
  public void removeKomaArray(Koma koma) {
    komaArray.remove(koma);
  }
  /**
   * komaArray を作る。
   * ban, 見て作る。
   * 駒台の駒は入れない
   */
  public void setKomaArray() {
    komaArray = new ArrayList<Koma>();
    Koma koma = null;
```

```
  //盤
  for(int i=1;i<4;i++) {
    for(int j=0;j<5;j++) {
      koma = ban.getBanarray(i, j);
      if(koma != null) {
        komaArray.add(koma);
        if(koma.getPlace()>34) {
          System.out.println("setKomaArray miss");///
        }
      }
    }
  }
}
 public void outputKyokumen() {
  //出力する。
  System.out.println("局面を出力します。teban"+teban);
  //後手の駒台の出力
  goteKomadai.outputKomadi();
  ban.outputBan();//盤面の出力
  //先手の駒台の出力
  senteKomadai.outputKomadi();
  //outputKomaArray();//komaArray の出力
}
/**
 * fileWriter を受け取って局面をそのファイルに保存する。
 */
 public void saveKyokumenFromFileWriter(FileWriter fileWriter) {
  try {
    //手番
    fileWriter.write(teban+"\n");
    //先手駒台
    senteKomadai.saveFile(fileWriter);
    //後手駒台
    goteKomadai.saveFile(fileWriter);
    //盤面
    ban.saveFile(fileWriter);
    fileWriter.write("\n");
  } catch (IOException e) {
    // TODO 自動生成された catch ブロック
    e.printStackTrace();
  }
}
/**
 * 局面をファイルに保存する。
 */
 public void saveKyokumen(String saveFile) {
  System.out.println("局面を"+saveFile+"に保存します。");
  File file = new File(saveFile);
  try {
```

```
    FileWriter fileWriter = new FileWriter(file);
    //手番
    fileWriter.write(teban+"\n");
    //先手駒台
    senteKomadai.saveFile(fileWriter);
    //後手駒台
    goteKomadai.saveFile(fileWriter);
    //盤面
    ban.saveFile(fileWriter);
    fileWriter.close();
  } catch (IOException e) {
    // TODO 自動生成された catch ブロック
    e.printStackTrace();
    System.out.println("保存に失敗しました。");
  }
  System.out.println("保存されました。");
}
/**
 * 局面を読み込む
 * @return 成功したら true
 */
public boolean readKyokumenFromFirstKyokumen(BufferedReader br) {
  System.out.println("局面を読み込みます。");
  try {
    //最初は手番を読み込む
    String str = br.readLine();
    if(str.equals("1")) {
      this.teban = 1;
    }else if(str.equals("2")) {
      this.teban = 2;
    }else {
      System.out.println("手番の読み込みに失敗しました。");
      return false;
    }
    if(!senteKomadai.loadFile(br)) {
      System.out.println("先手の駒台の読み込みに失敗しました。");
      return false;
    }
    if(!goteKomadai.loadFile(br)) {
      System.out.println("後手の駒台の読み込みに失敗しました。");
      return false;
    }
    if(!ban.loadFile(br)) {
      System.out.println("盤の読み込みに失敗しました。");
      return false;
    }
    br.readLine();//改行を読む
  } catch (IOException e) {
    // TODO 自動生成された catch ブロック
```

```
      e.printStackTrace();
    }
    return true;
  }
  public boolean loadKyokumen(String loadFile) {
    System.out.println("局面を読み込みます。");
    File file = new File(loadFile);
    try {
      FileReader filereader = new FileReader(file);
      BufferedReader br = new BufferedReader(filereader);
      //最初は手番を読み込む
      String str = br.readLine();
      if(str.equals("1")) {
        this.teban = 1;
      }else if(str.equals("2")) {
        this.teban = 2;
      }else {
        System.out.println("手番の読み込みに失敗しました。");
        br.close();
        return false;
      }
      if(!senteKomadai.loadFile(br)) {
        System.out.println("先手の駒台の読み込みに失敗しました。");
        return false;
      }
      if(!goteKomadai.loadFile(br)) {
        System.out.println("後手の駒台の読み込みに失敗しました。");
        return false;
      }
      if(!ban.loadFile(br)) {
        System.out.println("盤の読み込みに失敗しました。");
        return false;
      }
      br.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
    return true;
  }
// komaArray を出力
  public void outputKomaArray() {
    System.out.println("komaArray を出力します。");
    for(Koma koma:komaArray) {
      koma.outputMovePlace();
    }
  }
  /**
   * 入力の場所の駒を返す。何もないときは null
   * @return
```

```
   */
  public Koma getKomaFromPlace(int placeA,int placeB) {
    if(placeA==10) {
      Koma koma = senteKomadai.getKomadai(placeB);
      return koma;
    }else if(placeA==20) {
      Koma koma = goteKomadai.getKomadai(placeB);
      return koma;
    }else {
      return ban.getBanarray(placeA, placeB);
    }
  }
// 駒の場所を返す
  public Koma getKomaFromPlace(int place) {
    return getKomaFromPlace(place/10, place%10);
  }
  /**
   * 場所に駒をセットする。
   * 盤だけではなく駒台もできるようにする。
   * @param koma
   * @param placeA
   * @param placeB
   */
  public void setKomaFromPlace(Koma koma,int placeA,int placeB) {
    if(placeA<10) {
      //盤の時
      ban.setBanarray(placeA, placeB, koma);
    }else if(placeA==10){
      //先手の駒台
      senteKomadai.setKomadai(koma);
    }else if(placeA==20) {
      //後手の駒台
      goteKomadai.setKomadai(koma);
    }
  }
// 先手の駒台の駒を得る
  public Koma getSenteKomadaiKoma(int placeB) {
    return senteKomadai.getKomadai(placeB);
  }
// 後手の駒台の駒を得る
  public Koma getGoteKomadaiKoma(int placeB) {
    return goteKomadai.getKomadai(placeB);
  }
// 先手の駒台を得る
  public Komadai getSenteKomadai() {
    return senteKomadai;
  }
// 後手の駒台を得る
  public Komadai getGoteKomadai() {
```

```
    return goteKomadai;
  }
// 選んだ手番の駒台を得る
  public Komadai getKomadai(int teban) {
    if(teban==1) {
      return getSenteKomadai();
    }else if(teban==2) {
      return getGoteKomadai();
    }
    System.out.println("error: Kyokumen class getKomadai teban");
    return null;
  }
// 指定した座標の駒を得る
  public Koma getBanarray(int a,int b) {
    if (0<a && a<4 && 0<b && b<5) {
      return ban.getBanarray(a, b);
    }
    return null;
  }
// 指定した座標の駒を得る
  public Koma getBanarray(int place) {
    int a = place / 10;
    int b = place % 10;
    return getBanarray(a, b);
  }
// 手番を得る
  public int getTeban() {
    return teban;
  }
// komaArray を得る
  public ArrayList<Koma> getKomaArray(){
    return komaArray;
  }
// 盤の情報を得る
  public Ban getBan() {
    return ban;
  }
  public String getTebanString() {
    if(teban==1) {
      return "先手";
    }else if(teban==2) {
      return "後手";
    }
    return "error:getTebanString int kyokumen class";
  }
  public void setBanKomadai(Ban ban,Komadai sente,Komadai gote) {
    this.ban = ban;
    this.senteKomadai = sente;
    this.goteKomadai = gote;
```

```
  }
}
```

```
\begin{itemize}
\item  MakeBoardList クラス
\end{itemize}
{\small
\begin{verbatim}
package board;

import java.util.ArrayList;

import koma.Koma;

public class MakeBoardList {
  //次の手でライオン手の局面を作る
  public static ArrayList<Move> getNextMoveListOnlyLionte(Kyokumen kyokumen){
    int teban = kyokumen.getTeban();
    //move の list を作る。
    ArrayList<Move> moveListAll = makeMoveList(kyokumen,teban);
    if(moveListAll==null) {
      return null;
    }else if(moveListAll.isEmpty()) {
      return null;
    }
    return moveListAll;
  }
  /**
   * kyokumen クラスから次に行ける Move クラスのリストを返す
   * 評価値を決めたりはしない。反則は除く。
   * @param kyokumen
   * @return
   */
  public static ArrayList<Move> getNextMoveListNoFoul(Kyokumen kyokumen) {
    int teban = kyokumen.getTeban();
    //move の list を作る。
    ArrayList<Move> moveList = makeMoveList(kyokumen,teban);
    if(moveList==null) {
      return null;
    }else if(moveList.isEmpty()) {
      return null;
    }
    return moveList;
  }

  /**
   * 次の手の Board のリストを返す。
   * 最初は全ての駒の全ての動きを調べる。そのうちいらないのは消していく
```

```
 * 次の手の候補がなければ null 返す
 * @return
 */
public static ArrayList<Board> getNextBoardList(Board board) {
  Kyokumen kyokumen = board.getKyokumen();
  int teban = kyokumen.getTeban();
  //move の list を作る。
  ArrayList<Move> moveList = makeMoveList(kyokumen,teban);
  if(moveList==null) {
    return null;
  }else if(moveList.isEmpty()) {
    return null;
  }
  //moveList に評価値を加えていく。勝敗が決まる手があれば除いたり、CalcValue クラスにする
  ArrayList<Board> boardList  = new ArrayList<Board>();
  Board b = null;
  for(Move move:moveList) {
    b = board.getMoveNextBoard(move);
    boardList.add(b);
  }
  //moveList の中でいらないやつを消していく。候補手の手以下にする。
  return boardList;
}
/**
 * kyokumen から moveList を作る。
 * 全ての駒の全ての動き方のリスト。反則も含まれる
 * 駒がなれる時は成ると不成の両方
 * 持ち駒から打つ時は空いている全部の場所
 * @param myKomaList
 * @return
 */
private static ArrayList<Move> makeMoveList(Kyokumen kyokumen,int teban){
  ArrayList<Koma> myKomaList = kyokumen.getTebanKomaListAll(teban);
  ArrayList<Move> moveList = new ArrayList<Move>();
  Move move = null;
  for(Koma koma:myKomaList) {
    if(koma.getMotigoma()) {
      Koma banKoma = null;
      //駒が持ち駒の時, 盤上の全ての null の場所
      for(int i=1;i<4;i++) {
        for(int j=1;j<5;j++) {
          banKoma = kyokumen.getKomaFromPlace(i, j);
          if(banKoma==null) {
            move = new Move(koma.getPlaceA(),koma.getPlaceB(),i,j);
            moveList.add(move);
          }
        }
      }
    }
```

```
      //盤上の駒の時
      for(int place:koma.getMovePlace()) {
        move = new Move();
        move.setPlace(koma.getPlaceA(), koma.getPlaceB(), place);
        moveList.add(move);
        //なれる場合は成るの時も
        if(koma.ableToNaru()) {
          if(move.ableNaru(koma.getTeban())) {
            move = move.clone();
            move.setNaru(true);
            moveList.add(move);
          }
        }
      }
    }
    return moveList;
  }
}
```

- Move クラス

```
package board;

import java.util.Scanner;

import koma.Koma;
import main.Calc;

//動いた値を反対からも読めるように保存する
public class Move {
  private int beforePlaceA;//23 の 2, 駒台なら 10,20
  private int beforePlaceB;//23 の 3
  private int afterPlaceA;
  private int afterPlaceB;
  private boolean naru;//true でなった。
  private int getKoma;//取っていなければ 0

  public Move() {
    beforePlaceA = 0;
    beforePlaceB = 0;
    afterPlaceA = 0;
    afterPlaceB = 0;
    naru = false;
    getKoma = 0;
  }
  public Move(Move move) {
    this.beforePlaceA = move.getBeforePlaceA();
    this.beforePlaceB = move.getBeforePlaceB();
```

```
  this.afterPlaceA = move.getAfterPlaceA();
  this.afterPlaceB = move.getAfterPlaceB();
  this.naru = move.getNaru();
  this.getKoma = move.getGetKoma();
}
public Move(int beforeA,int beforeB,int afterA,int afterB) {
  setBeforePlaceA( beforeA);
  setBeforePlaceB(beforeB);
  setAfterPlaceA(afterA);
  setAfterPlaceB(afterB);
  naru = false;
  getKoma = 0;
}
public Move clone() {
  return new Move(this);
}
/**
 * move が等しいか判定
 * @param move
 * @return 同じなら true
 */
public boolean equalsMove(Move move) {
  if(this.beforePlaceA==move.getBeforePlaceA() && this.beforePlaceB==move.getBeforePlaceB() &&
      this.afterPlaceA==move.getAfterPlaceA() && this.afterPlaceB==move.getAfterPlaceB() &&
      this.getKoma==move.getGetKoma()) {
    if(this.naru && move.getNaru()) {
      return true;
    }
    if(!this.naru && !move.getNaru()) {
      return true;
    }
  }
  return false;
}
/**
 * 駒がなれるか判断。人間用
 * なれる場合は成り、true を返す
 */
public boolean checkNaru(Koma koma) {
  //持ち駒からは成れない
  if(koma.getMotigoma()) {
    return false;
  }
  if(koma.ableToNaru()) {
    return true;
  }
  return false;
}
/**
```

```
 * なれるかどうかの判断
 * なれるなら true
 * 駒の確認も必要
 * @return
 */
 public boolean ableNaru(int teban) {
   if(teban==1) {
     if(beforePlaceB<2 || afterPlaceB<2) {
       return true;
     }
   }else {
     if(beforePlaceB>3 || afterPlaceB>3) {
       return true;
     }
   }
   return false;
 }
 public boolean checkAbleNaru(Koma koma) {
   //持ち駒からは成れない
   if(koma.getMotigoma()) {
     return false;
   }
   if(koma.ableToNaru()) {
     if(koma.getTeban()==1) {
       if(beforePlaceB<2 || afterPlaceB<2) {
         return true;
       }
     }else {
       if(beforePlaceB>3 || afterPlaceB>3) {
         return true;
       }
     }
   }
   return false;
 }
 /**
  * move の出力。テストで使う。///
  * 棋譜保存の時も使う
  */
 public void outputMoveTest() {
   int n = 0;//不成
   if(naru) {
     n = 1;//成る
   }
   System.out.println(beforePlaceA+","+beforePlaceB+","+afterPlaceA+","+afterPlaceB+","
   +n+","+getKoma+","+getKoma);
 }
 /**
  * 棋譜を保存するときのアウトプット
```

```
 */
 public void outputMoveKihu() {
   int n = 0;//不成
   if(naru) {
     n = 1;//なったとき
  }
   System.out.println(beforePlaceA+","+beforePlaceB+","+afterPlaceA+","+
       afterPlaceB+","+n+","+getKoma);
}
/**
 * 棋譜を保存するときの改行を含む文字列を返す
 * @return
 */
 public String getMoveKihu() {
   int n = 0;//不成
   if(naru) {
     n = 1;//なったとき
  }
   String ans = beforePlaceA+","+beforePlaceB+","+afterPlaceA+","+
       afterPlaceB+","+n+","+getKoma + "\n";
   return ans;
}
/**
 * move の入力。
 */
 public void inputMove() {
   Scanner scan = new Scanner(System.in);
   String str;
  //before の入力
   while(true) {
     System.out.println("動かす駒の今の場所を入力してください。");
     str = scan.next();
     if(Calc.isNumber(str)) {
       int before = Integer.parseInt(str);
       beforePlaceA = before / 10;
       beforePlaceB = before % 10;
       break;
    }
     System.out.println("入力が数字ではありません。");
  }
  //after の入力
   while(true) {
     System.out.println("駒の動かす先の場所を入力してください。");
     str = scan.next();
     if(Calc.isNumber(str)) {
       int after = Integer.parseInt(str);
       afterPlaceA = after / 10;
       afterPlaceB = after % 10;
       break;
```

```
    }
    System.out.println("入力が数字ではありません。");
  }
  ///scan.close();
}
/**
 * ファイルから読み込むときに使う。
 * 一行ずつ
 * 新しく move クラス作って返す
 */
public Move getInputMove(String str) {
  String[] input = str.split(",", 0);
  if(input.length != 6) {
    System.out.println("error: Move getInputMove1"+str);
    return null;
  }
  int[] setNumber = new int[6];
  for(int i=0;i<6;i++) {
    if(Calc.isNumber(input[i])) {
      setNumber[i] = Integer.parseInt(input[i]);
    }else {
      System.out.println("error: Move getInputMove2"+input[i]);
      return null;
    }
  }
  Move move = new Move();
  move.setBeforePlaceA(setNumber[0]);
  move.setBeforePlaceB(setNumber[1]);
  move.setAfterPlaceA(setNumber[2]);
  move.setAfterPlaceB(setNumber[3]);
  if(setNumber[4]==1) {
    move.setNaru(true);
  }else if(setNumber[4]==0) {
    move.setNaru(false);
  }else {
    System.out.println("error: Move getInputMove naru"+setNumber[4]);
    return null;
  }
  move.setGetKoma(setNumber[5]);
  return move;
}
/**
 * 入力と同じ場所が afterPlace か
 * @return
 */
public boolean matchAfterPlace(int a,int b) {
  if(a==afterPlaceA && b==afterPlaceB) {
    return true;
  }
```

```
  return false;
}
 public boolean matchAfterPlace(int place) {
  int a = place /10;
  int b = place % 10;
  return matchAfterPlace(a, b);
}
 public int getBeforePlaceA() {
  return beforePlaceA;
}
 public int getBeforePlaceB() {
  return beforePlaceB;
}
 public int getAfterPlaceA() {
  return afterPlaceA;
}
 public int getAfterPlaceB() {
  return afterPlaceB;
}
 public void setBeforePlaceA(int beforeA) {
  if(beforeA==10 || beforeA==20 || (0<beforeA && beforeA<4) ) {
    this.beforePlaceA = beforeA;
  }
}
 public void setBeforePlaceB(int beforeB) {
  if(beforeB==10 || beforeB==20 || (0<beforeB && beforeB<5) ) {
    this.beforePlaceB = beforeB;
  }
}
 public void setPlace(int beforeA,int beforeB,int after) {
  setBeforePlaceA( beforeA);
  setBeforePlaceB(beforeB);
  setAfterPlace(after);
}
 public void setAfterPlaceA(int afterA) {
  if(0<afterA && afterA<4 ) {
    this.afterPlaceA = afterA;
  }
}
 public void setAfterPlaceB(int afterB) {
  if(0<afterB && afterB<5 ) {
    this.afterPlaceB = afterB;
  }
}
 public void setAfterPlace(int afterPlace) {
  int a = afterPlace / 10;
  int b = afterPlace % 10;
  setAfterPlaceA(a);
  setAfterPlaceB(b);
```

```
  }
  public boolean getNaru() {
    return naru;
  }
  public int getGetKoma() {
    return getKoma;
  }
  public void setNaru(boolean naru) {
    this.naru = naru;
  }
  public void setGetKoma(int koma) {
    this.getKoma = koma;
  }
  public int getAfterPlaceMix() {
    return afterPlaceA*10 + afterPlaceB;
  }
}
```

- ValueComparatorGote クラス

```
package board;

import java.util.Comparator;

import calcValue.Value;

public class ValueComparatorGote implements Comparator<Board>{
  @Override
  public int compare(Board b1, Board b2) {
    Value v1 = b1.getValue();
    Value v2 = b2.getValue();
    //勝ちが決まっているならどちらでも良い?, 千日手の時もある
    if(v1.getDeterme() && v2.getDeterme()) {
      if(v1.getValue() < v2.getValue()) {
        return 1;
      }else if(v1.getValue() == v2.getValue()) {
        return 0;
      }
      return -1;
    }
    if(v1.getDeterme()) {
      if(v1.getValue() < 0) {
        return 1;
      }else if(v1.getValue() > 0) {
        return -1;
      }else {
        //千日手
        if(v1.getValue() < v2.getValue()) {
```

```
        return 1;
      }else if(v1.getValue() == v2.getValue()) {
        return 0;
      }
      return -1;
    }
  }
  if(v2.getDeterme()) {
    if(v2.getValue() < 0) {
      return -1;
    }else if(v2.getValue() > 0) {
      return 1;
    }else {
      //千日手
      if(v1.getValue() < v2.getValue()) {
        return 1;
      }else if(v1.getValue() == v2.getValue()) {
        return 0;
      }
      return -1;
    }
  }
  if(v1.getValue() < v2.getValue()) {
    return 1;
  }else if(v1.getValue() == v2.getValue()) {
    return 0;
  }
  return -1;
}
/**
 * b1 が小さければ true
 */
public static boolean compareBoardGote(Board b1,Board b2) {
  Value v1 = b1.getValue();
  Value v2 = b2.getValue();
  //勝ちが決まっているならどちらでも良い?
  if(v1.getDeterme()) {
    if(v1.getValue() < 0) {
      return true;
    }else if(v1.getValue() > 0) {
      return false;
    }else {
      //千日手
      if(v1.getValue() < v2.getValue()) {
        return true;
      }
      return false;
    }
  }
```

```
    if(v2.getDeterme()) {
      if(v2.getValue() < 0) {
        return false;
      }else if(v2.getValue() > 0) {
        return true;
      }else {
        //千日手
        if(v1.getValue() < v2.getValue()) {
          return true;
        }
        return false;
      }
    }
    if(v1.getValue() < v2.getValue()) {
      return true;
    }
    return false;
  }
}
```

- ValueComparatorSente クラス

```
package board;

import java.util.Comparator;

import calcValue.Value;

public class ValueComparatorSente implements Comparator<Board>{

  /**
   * b1 がよければ 1,b2 がよければ-1 返す。
   */
  @Override
  public int compare(Board b1, Board b2) {
    Value v1 = b1.getValue();
    Value v2 = b2.getValue();
    //勝ちが決まっているならどちらでも良い?
    if(v1.getDeterme() && v2.getDeterme()) {
      if(v1.getValue() < v2.getValue()) {
        return -1;
      }if(v1.getValue() == v2.getValue()) {
        return 0;
      }
      return 1;
    }
    if(v1.getDeterme()) {
      if(v1.getValue() > 0) {
```

```
      return 1;
    }else if(v1.getValue() < 0){
      return -1;
    }else {
      //千日手
      if(v1.getValue() < v2.getValue()) {
        return -1;
      }else if(v1.getValue() == v2.getValue()) {
        return 0;
      }
      return 1;
    }
  }
  if(v2.getDeterme()) {
    if(v2.getValue() > 0) {
      return -1;
    }else if(v2.getValue() < 0) {
      return 1;
    }else {
      //千日手
      if(v1.getValue() < v2.getValue()) {
        return -1;
      }else if(v1.getValue() == v2.getValue()) {
        return 0;
      }
      return 1;
    }
  }
  if(v1.getValue() < v2.getValue()) {
    return -1;
  }else if(v1.getValue() == v2.getValue()) {
    return 0;
  }
  return 1;
}
/**
 * b1 が大きければ true
 * 同じときは気にしていない
 */
public static boolean cmpareBoardSente(Board b1,Board b2) {
  Value v1 = b1.getValue();
  Value v2 = b2.getValue();
  //勝ちが決まっているならどちらでも良い?
  if(v1.getDeterme()) {
    if(v1.getValue() > 0) {
      return true;
    }else if(v1.getValue() < 0){
      return false;
    }else {
```

```
        //千日手
        if(v1.getValue() < v2.getValue()) {
          return false;
        }
        return true;
      }
    }
    if(v2.getDeterme()) {
      if(v2.getValue() > 0) {
        return false;
      }else if(v2.getValue() < 0) {
        return true;
      }else {
        //千日手
        if(v1.getValue() < v2.getValue()) {
          return false;
        }
        return true;
      }
    }
    if(v1.getValue() < v2.getValue()) {
      return false;
    }
    return true;
  }
}
```

- LikelyToBeTokenValue クラス

```
package calcValue;

import java.util.ArrayList;

import board.Kyokumen;
import board.Move;
import koma.Koma;

public class LikelyToBeTokenValue extends ParentValue {
  public LikelyToBeTokenValue(Kyokumen kyokumen) {
    super(kyokumen);
    // TODO 自動生成されたコンストラクター・スタブ
  }
  /**
   * 評価値を決める。
   * valueBoolean,valueInt を決める
   */
  public void calcValueL() {
    setValueInt(0);
```

```
    setValueBoolean(true);
    //自分の駒のリストを作る, 盤上だけ
    ArrayList<Koma> myKomaList = getKyokumen().getTebanKomaListBan(getTeban());
    //自分の駒の動けるところに相手の駒があるか確認する。もし駒があればその先の評価値を決める。
    Koma komaEnemy = null;
    Move move = null;
    Kyokumen kyokumenEnemy = null;
    ArrayList<Integer> nextValueList = new ArrayList<Integer>();//次の局面の評価値のリスト

    for(Koma koma:myKomaList) {
      for(int place:koma.getMovePlace()) {
        komaEnemy = getKyokumen().getKomaFromPlace(place);//自分の駒が動ける場所に相手の駒があるか
        if(komaEnemy != null) {
          //確認
          //敵の駒を取ることができる。move クラスを作ってその局面の評価値を得る,nextValueList に加える。
          move = new Move(koma.getPlaceA(),koma.getPlaceB(),place/10,place%10);
          kyokumenEnemy = getKyokumen().clone();
          kyokumenEnemy.moveNextKyokumen(move);
          //次の局面の評価値を nextValueList に加える。
          int value = 0;
          nextValueList.add(value);
        }
      }
    }
    if(nextValueList.isEmpty()) {
      setValueBoolean(false);//何もない時
      return;
    }
    if(getTeban()==1) {
      int max = nextValueList.get(0);
      for(int v:nextValueList) {
        if(max<v) {
          max = v;
        }
      }
      setValueInt(max);
    }else {
      int min = nextValueList.get(0);
      for(int v:nextValueList) {
        if(min>v) {
          min = v;
        }
      }
      setValueInt(min);
    }
  }
}
```

- LossKomaValue クラス

```
package calcValue;

import board.Kyokumen;
import koma.Koma;

public class LossKomaValue extends ParentValue{
  public LossKomaValue(Kyokumen kyokumen) {
    super(kyokumen);
  }
  /**
   * 評価値を決める。
   * valueBoolean,valueInt を決める
   */
  public void calcValue() {
    int value = 0;
    //盤上の駒
    for(Koma koma:getKyokumen().getKomaArray()) {
      value += koma.getPointTeban();
    }
    //駒台の駒
    value += getKyokumen().getSenteKomadai().getAllPointTeban()
      + getKyokumen().getGoteKomadai().getAllPointTeban();
    setValueInt(value);
  }

}
```

- MovePlaceValue クラス

```
package calcValue;

import board.Kyokumen;
import koma.Koma;

public class MovePlaceValue extends ParentValue{

  public MovePlaceValue(Kyokumen kyokumen) {
    super(kyokumen);
    // TODO 自動生成されたコンストラクター・スタブ
  }
  /**
   * 評価値を決める。
   * valueBoolean,valueInt を決める
   */
  public void calcValue() {
    //valueBoolean は変えない
```

```
    int value = 0;
    int movePlaceN = 0;
    for(Koma koma:getKyokumen().getKomaArray()) {
      movePlaceN = koma.getMovePlace().size();
      if(koma.getTeban()==2) {
        movePlaceN = -1 * movePlaceN;
      }
      value += movePlaceN;
    }
    setValueInt(value);
  }
}
```

- ParentValue クラス

```
package calcValue;

import board.Kyokumen;

/**
 * kyokumen クラスのそれぞれの評価値を決める親クラス
 * このクラスを継承して評価値を決めるクラスを作る
 */
public class ParentValue {
  private Kyokumen kyokumen;
  private boolean valueBoolean;//勝敗が決まっているときは true
  private int valueInt;
  public ParentValue(Kyokumen kyokumen) {
    this.kyokumen = kyokumen;
    calcValue();
  }
  /**
   * 評価値を決める。
   * valueBoolean,valueInt を決める
   */
  public void calcValue() {
    valueInt = 0;
  }
  public boolean getValueBoolean() {
    return valueBoolean;
  }
  /**
   * int 型で評価値を返す。
   * @return
   */
  public int getValueInt() {
    return valueInt;
  }
```

```
  public Kyokumen getKyokumen() {
    return kyokumen;
  }
  public int getTeban() {
    return kyokumen.getTeban();
  }
  public void setValueInt(int valueInt) {
    this.valueInt = valueInt;
  }
  public void setValueBoolean(boolean valueBoolean) {
    this.valueBoolean = valueBoolean;
  }
}
```

- SafetyLionValue クラス

```
package calcValue;

import java.util.ArrayList;

import board.Kyokumen;
import koma.Koma;
import main.Calc;
/**
 * ライオンの安全度を評価する。
 * 責められているかどうかを評価
 * 自分と相手でやる。
 * 大雑把に決める。
 * 自分と相手との比較
 * 自陣に敵の駒がどれくらいあるか、駒の強さによって危険度変わる
 *
 */

public class SafetyLionValue extends ParentValue {

  public SafetyLionValue(Kyokumen kyokumen) {
    super(kyokumen);
    // TODO 自動生成されたコンストラクター・スタブ
  }
  /**
   * 評価値を決める。
   * valueBoolean,valueInt を決める
   */
  public void calcValue() {
    setValueBoolean(false);
    int value = 0;//最初は危険な駒のポイントの合計
    value += makeValue(getTeban());
    value += makeValue(Calc.changeTeban(getTeban()));
```

```
  setValueInt((int)(value * 0.3));
}
/**
 * teban の玉の危険度を返す
 * @param komaList
 * @param teban
 * @return
 */
private int makeValue(int teban) {
  int value = 0;//最初は危険な駒のポイントの合計
  //自陣に敵の駒が何枚あるか, 駒のポイントによって危険度変わる。
  //敵の駒のリストを作る
  ArrayList<Koma> komaEnemyList = getKyokumen().getTebanKomaListBan(Calc.changeTeban(teban));
  //自分のライオンの場所
  int placeLion = getKyokumen().getPlaceLion(teban);
  int placeLionA = placeLion / 10;
  int placeLionB = placeLion % 10;
  //自分のライオンから 1 マス以内に敵の駒あれば危険。自陣に駒があれば危険。///
  int placeEnemyKomaA = 0;
  int placeEnemyKomaB = 0;
  int diffA = 0;
  int diffB = 0;
  for(Koma koma:komaEnemyList) {
    placeEnemyKomaA = koma.getPlaceA();
    placeEnemyKomaB = koma.getPlaceB();
    //ライオンから 1 マス以内にあるか
    diffA = placeLionA - placeEnemyKomaA;
    diffB = placeLionB - placeEnemyKomaB;
    if(checkDiff(diffA) && checkDiff(diffB)) {
      value += koma.getPointTeban();
    }else {
      //自陣に駒があるか
      if(teban==1) {
        if(koma.getPlaceB() > 2) {
          value += (int)(koma.getPointTeban() * 0.6);
        }
      }else if(teban==2) {
        if(koma.getPlaceB() < 3) {
          value += (int)(koma.getPointTeban() * 0.6);
        }
      }
    }
  }
  return value;
}
/**
 * diff が 1 マス以内か判断.1 マス以内なら true
 * @return
 */
```

```
  private boolean checkDiff(int diff) {
    if((diff > -1) && (diff < 1)) {
      return true;
    }
    return false;
  }

}
```

- TadaValue クラス

```
package calcValue;

import java.util.ArrayList;

import board.Kyokumen;
import board.Move;
import koma.Koma;
import main.Calc;

public class TadaValue extends ParentValue {

  public TadaValue(Kyokumen kyokumen) {
    super(kyokumen);
    // TODO 自動生成されたコンストラクター・スタブ
  }
  /**
   * 評価値を決める。
   * valueBoolean,valueInt を決める
   */
  public void calcValue() {
    //タダで取れる駒のリストを作る。
    //自分の駒のリストを作る, 盤上だけ
    ArrayList<Koma> myKomaList = getKyokumen().getTebanKomaListBan(getTeban());
    //自分の駒の動けるところに相手の駒があるか確認する。もし駒があればその場所に相手の駒が動けるか確認する
    Koma komaEnemy = null;
    Move move = null;
    int value = 0;//評価値の合計を作る。
    for(Koma koma:myKomaList) {
      for(int place:koma.getMovePlace()) {
        komaEnemy = getKyokumen().getKomaFromPlace(place);//自分の駒が動ける場所に相手の駒があるか
        if(komaEnemy != null) {
          //自分の駒の動けるところに相手の駒がある。敵の駒を取ることができる。move クラスを作って次の局面に
いく。
          move = new Move(koma.getPlaceA(),koma.getPlaceB(),place/10,place%10);
          if(checkEnemyKomaMove(move)) {
            //相手の駒が効いていない
            value += komaEnemy.getPointTeban();//取れる駒の評価値を加える
```

```
        }
       }
      }
    }
    value = value * -1;
    setValueInt(value);
  }
  /**
   * 相手の駒が move の移動先に効いているか確認する。
   * 効いていなければ true を返す
   * @param move
   * @return
   */
  private boolean checkEnemyKomaMove(Move move) {
    Kyokumen kyokumenEnemy = getKyokumen().clone();
    kyokumenEnemy.moveNextKyokumen(move);
    //相手の駒のリストを作る。
    ArrayList<Koma> komaListEnemy = kyokumenEnemy.getTebanKomaListBan(Calc.changeTeban(getTeban()));
    //相手の駒の動けるところにその場所があるか
    for(Koma koma:komaListEnemy) {
      for(int place:koma.getMovePlace()) {
        if(place == move.getAfterPlaceMix()) {
          return false;
        }
      }
    }
    return true;
  }
}
```

- TumiCheck クラス

```
package calcValue;

import java.util.ArrayList;

import board.CalcKyokumenFoul;
import board.Kyokumen;
import board.MakeBoardList;
import board.Move;

public class TumiCheck {
  /**
   *手番の玉がその時詰んでいるかを確認する。0 手読み
   * @param kyokumen
   * @return 詰みがある時 true
   */
  public static boolean checkTumiTeban(Kyokumen kyokumen) {
```

```
  if(CalcKyokumenFoul.checkLionte(kyokumen, kyokumen.getTeban())) {
   //ライオン手が手番の玉にかかっている。回避できるか調べる。
    ArrayList<Move> moveList = MakeBoardList.getNextMoveListNoFoul(kyokumen);
    if(moveList==null || moveList.isEmpty()) {
      return true;
    }
  }
  return false;
}
/**
 * 局面で n 手以下の詰みがあるか調べる
 * なるべく最短の詰みを見つけられるように横に調べる。本当は横で調べたいがやり方がわからないので縦に調べる。

 * 手に優先順位をつけられるとよい.
 * n は奇数を想定している。
 * @param kyokumen ライオン手のかかっていない局面。詰将棋と同じ
 * @return 詰みがあれば最初の一手の move を返す。
 */
public static Move checkTumiNTe(Kyokumen kyokumen,int n) {
  if(n<1) {
    return null;
  }
  //次の手でライオン手の手のリストを作る。
  ArrayList<Move> moveListLion = MakeBoardList.getNextMoveListOnlyLionte(kyokumen);
  //どれか一つでも詰んでいるのがあればその move を返す
  Kyokumen kyokumenNext = null;
  for(Move move:moveListLion){
    kyokumenNext = kyokumen.clone();
    kyokumenNext.moveNextKyokumen(move);
    //次の局面で玉側が詰されているかを調べる
    if(checkTumiForLion(kyokumenNext, n-1)) {
      //詰んでいる
      return move;
    }
  }
  return null;
}
/**
 * ライオン手されている局面で n 手以下で詰まされるか調べる。
 * @param kyokumen n は偶数を想定
 * @return 詰みがあれば true
 */
public static boolean checkTumiForLion(Kyokumen kyokumen,int n) {
  if(n<0) {
    return false;
  }
  ArrayList<Move> moveListEscape = MakeBoardList.getNextMoveListNoFoul(kyokumen);
  if(moveListEscape==null || moveListEscape.isEmpty()) {
    return true;
```

```
    }
    //全ての逃げ方で詰むかを確認する. 一つでも詰まないのがあれば false
    Kyokumen kyokumenNext = null;
    for(Move move:moveListEscape) {
      kyokumenNext = kyokumen.clone();
      kyokumenNext.moveNextKyokumen(move);
      if(checkTumiNTe(kyokumenNext, n-1)==null) {
       //詰まない
        return false;
      }
    }
    //詰まない逃げ方はない
    return true;
  }
}
```

- Value クラス

```
package calcValue;

import java.util.ArrayList;

import board.CalcKyokumenFoul;
import board.Kyokumen;
import board.Move;

public class Value {
  private int value;//評価値,//歩特で 1,000,000 くらい？,2147480000max
  private boolean determe;//勝負が決まっている時 true,

  public Value(){
    determe = false;
    value = 0;//
  }
  /**
   * 評価値を計算する。
   * 次の手は読まずに現在の局面だけで評価する
   */
  public void calcValuePresent(Kyokumen kyokumen,ArrayList<Kyokumen> kyokumenList,Move beforeMove) {
    if(kyokumenList!=null) {
    //千日手チェック
      int sennitite = CalcKyokumenFoul.checkSennitite(kyokumenList, kyokumen);
      if(sennitite==1) {
        System.out.println("sennitite!");
        setDeterme(true);
        setValue(0);
        return;
      }else if(sennitite==2) {
```

```
      //手番の負け
      setDeterme(true);
      setValue(100);
      if(kyokumen.getTeban()==1) {
        setValue(-100);
      }
      return;
    }else if(sennitite==3) {
      //手番の勝ち
      setDeterme(true);
      setValue(100);
      if(kyokumen.getTeban()==2) {
        setValue(-100);
      }
      return;
    }
  }
  //詰みチェック, 自分の玉にライオン手がかかっていて 3 手で詰むか
  if(TumiCheck.checkTumiForLion(kyokumen,2)) {
    //詰みがある. 手番が負けている。
    setLose(kyokumen.getTeban());
    return;
  }
  //詰みチェック, 相手の玉を詰ますことができるか
  if(TumiCheck.checkTumiNTe(kyokumen, 1)!=null) {
    //詰みがある。勝っている
    setWin(kyokumen.getTeban());
    return;
  }
  //駒の損得
  LossKomaValue lossKomaValue = new LossKomaValue(kyokumen);
  addValue(lossKomaValue.getValueInt());
  //駒の働き、動ける場所
  MovePlaceValue movePlaceValue = new MovePlaceValue(kyokumen);
  addValue(movePlaceValue.getValueInt());
  //タダで取られる駒
  TadaValue tadaValue = new TadaValue(kyokumen);
  addValue(tadaValue.getValueInt());
}
private void addValue(int add) {
  value += add;
}
public int getValue() {
  return value;
}
public boolean getDeterme() {
  return determe;
}
public void setValue(int value) {
```

```
  this.value = value;
}
 public void setDeterme(boolean determe) {
  this.determe = determe;
}
 public void setValueAll(Value value) {
  this.value = value.getValue();
  this.determe = value.getDeterme();
}
/**
 * 勝ちが決まっている時のその設定をする
 * @param teban
 */
 public void setWin(int teban) {
  setDeterme(true);
  if(teban==1) {
    setValue(100);
  }else if(teban==2) {
    setValue(-100);
  }
}
/**
 * 負けが決まっている時にその設定をする。
 * 手番が負け
 * @param teban
 */
 private void setLose(int teban) {
  setDeterme(true);
  if(teban==1) {
    setValue(-100);
  }else if(teban==2) {
    setValue(100);
  }
}
/**
 * teban が勝っていれば true
 * @param teban
 * @return
 */
 public boolean getWin(int teban) {
  if(!determe) {
    return false;
  }
  if(teban==1 && (value > 0)) {
    return true;
  }
  if(teban==2 && (value < 0)) {
    return true;
  }
```

```
    return false;
  }

}
```

- CalcKoma クラス

```
package koma;

import main.Calc;

//駒を作るクラス
public class CalcKoma {
  //ファイルから局面を読み込んだ時の読み込み
   public static Koma makeLoad_Koma(String loadStr){
    String[] str = loadStr.split(",", 0);
    if(str.length != 4) {
      System.out.println("駒の保存の長さが違います。");
      return null;
    }
    //手番
    int teban = changeTebanIntFromString(str[0]);
    if(teban == 0) {
      return null;
    }
    //place
    int a = -1;
    int b = -1;
    if(Calc.isNumber(str[1])) {
      a = Integer.parseInt(str[1]);
    }else {
      return null;
    }
    if(Calc.isNumber(str[2])) {
      b = Integer.parseInt(str[2]);
    }else {
      return null;
    }
    if(!isBan(a,b)) {
      return null;
    }
    int n = -1;
    if(Calc.isNumber(str[3])) {
      n = Integer.parseInt(str[3]);
    }else {
      return null;
    }
    Koma koma = makeKoma(n, teban);
```

```
  if(koma == null) {
    return null;
  }
  koma.setPlace(a, b);
  return koma;
}
private static int changeTebanIntFromString(String str) {
  if(str.equals("1")) {
    return 1;
  }else if(str.equals("2")) {
    return 2;
  }
  return 0;
}
//番号から駒を作る
public static Koma makeKoma(int n,int teban) {
  Koma koma = null;
  if(n==1) {
    koma = new Hiyoko(teban);
  }else if(n==2) {
    koma = new Zou(teban);
  }else if(n==3) {
    koma = new Kirin(teban);
  }else if(n==4) {
    koma = new Lion(teban);
  }else if(n==11) {
    koma = new Niwatori(teban);
  }
  return koma;
}
/**
 * a*10+b が 11-34 の中にあるなら true
 * @param a
 * @param b
 * @return
 */
public static boolean isBan(int a,int b) {
  if(a>0 && a<10 && b>0 && b<10) {
    return true;
  }else {
    return false;
  }
}
/**
 * a*10+b が 11-34 の中にあるなら true
 * @param a
 * @param b
 * @return
 */
```

```
  public boolean isBan(int place) {
    int a = place /10;
    int b = place % 10;
    return isBan(a,b);
  }
}
```

- Hiyoko クラス

```
package koma;

import board.Kyokumen;

public class Hiyoko extends Koma {

  public Hiyoko(int tebanN){
    super(tebanN);
    setPoint(1000000);
  }
  /**
   * 動けるか。反則にならないか。歩香桂だけ
   * 反則でなければ false, 反則なら true
   * @return
   */
  public boolean ableToMove() {
    if(getTeban()==1 && getPlaceB()==1) {
      return true;
    }
    if(getTeban()==2 && getPlaceB() == 4) {
      return true;
    }
    return false;
  }
  //ひよこは成れるので true を返す。
  public boolean ableToNaru() {
    return true;
  }
  public int getKomaNumber() {
    return 1;
  }
  public String getKomaName() {
    return "  ひ";
  }
  //動ける場所を決める
  public void decideMovePlace(Kyokumen kyokumen) {
    Koma koma = null;
    int a = getPlaceA();
    int b = getPlaceB();
```

```
    int difference = 1;
    if(getTeban()==1) {
      difference = -1;
    }
    b += difference;
    koma = kyokumen.getBanarray(a, b);
    if(koma != null) {
      if(koma.getTeban() == getTeban()) {
        return;
      }
    }
    addMovePlace(a*10+b);

  }
}
```

- Kirin クラス

```
package koma;

import board.Kyokumen;

public class Kirin extends Koma{

  public Kirin(int tebanN)  {
    super(tebanN);
    setPoint(12000000);
  }

  public int getKomaNumber() {
    return 3;
  }
  public String getKomaName() {
    return "  キ";
  }

  //動ける場所を決める
  public void decideMovePlace(Kyokumen kyokumen) {
    int a = getPlaceA() * 10;
    int b = getPlaceB();
    CalcKoma calc = new CalcKoma();
    Koma koma = null;
    int[] array = {a+10+b,a-10+b,a+b+1,a+b-1};
    for(int i:array) {
      if(calc.isBan(i)) {
        koma = kyokumen.getBanarray(i);
        if(koma == null) {
```

```
          addMovePlace(i);
        }else {
          if(koma.getTeban() != getTeban()) {
            addMovePlace(i);
          }
        }
      }
    }
  }
}
```

- Koma クラス

```
package koma;

import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;

import board.Kyokumen;

/*コマのクラス。自分の状態を保存する*/

public class Koma {

  private int teban; //どちらの駒かを判断する。1=先手、2=後手
  private int placeA;
  private int placeB;
  private int point;//何点の価値があるか。基本的に動ける場所が多いとポイントが上がる
  private int pointSpecial;
  private ArrayList<Integer> movePlace; //動ける場所。
  private boolean motigoma; //持ち駒かどうか。違うなら false

  public Koma(int tebanN){
    teban = tebanN;
    movePlace = new ArrayList<Integer>();
    motigoma = false;
    placeA = 0;
    placeB = 0;
  }
  public Koma clone() {
    Koma komaClone = CalcKoma.makeKoma(getKomaNumber(), getTeban());
    komaClone.setPlace(getPlaceA(), getPlaceB());
    komaClone.setPoint(getPoint());
    komaClone.setPointSpecial(getPointSpecial());
    komaClone.setMovePlace(getMovePlaceCloneDeep());
    komaClone.setMotigoma(getMotigoma());
    return komaClone;
```

```
}
 public boolean equal(Koma koma) {
   if(koma==null){
     return false;
  }
   if(this.teban != koma.getTeban()){
     return false;
  }
   if(this.placeA != koma.getPlaceA()) {
     return false;
  }
   if(this.placeB != koma.getPlaceB()) {
     return false;
  }
   if(this.motigoma != koma.getMotigoma()) {
     return false;
  }
   return true;
}
 public void saveFile(FileWriter fileWriter) {
   try {
     fileWriter.write(teban + "," + placeA + "," + placeB + "," + getKomaNumber() + "\n");
  } catch (IOException e) {
    // TODO 自動生成された catch ブロック
     e.printStackTrace();
  }
}
//動けるかどうかを判断する。
 public boolean ableToMove() {
   return false;
}
/**
 * 不成の駒作る
 * おかしい時は自分を返す
 * @return
 */
 public Koma getNarazukoma() {
   if(getKomaNumber()>10) {
     Koma narazuKoma = CalcKoma.makeKoma(getKomaNumber()-10, getTeban());
     narazuKoma.setPlace(getPlaceA(), getPlaceB());
     return narazuKoma;
  }
   System.out.println("error: Koma getNarazukoma not narigoma");
   return this;
}
/**
 * 成り駒を返す。
 * しっかり成り駒を作る。
 * @return
```

```
 */
 public Koma getNarigoma() {
   int n = getKomaNumber();
   Koma koma = CalcKoma.makeKoma(n+10, getTeban());
   koma.setPlace(getPlaceA(), getPlaceB());
   return koma;
}
/**
 * なれる駒は true を返すようにする。
 * @return
 */
 public boolean ableToNaru() {
   return false;
}
//全角 2 文字で返す。スペースは左にする
 public String getKomaName() {
   return "  駒";
}
 private String getTebanS() {
   if(teban==1) {
     return "↑";
  }else {
     return "↓";
  }
}
//全角 2, 半角 1, ↑↓, 半角スペース 1 で出力する。
 public void outputKoma() {
   System.out.print(getKomaName() + getTebanS() + " ");
  ///System.out.print(getKomaName()+getPlace() + getTebanS() + " ");///
}
//テスト用
 public void outputTest() {
   System.out.println(getKomaName() + getTebanS() + " " +getPlace());
}
 public void movePlaceClear() {
   movePlace.clear();
}
//移動できる場所を決める
 public void decideMovePlace(Kyokumen kyokumen) {
  //
}
//その場所に移動できるか
 public boolean containMovePlace(int place) {
   return movePlace.contains(place);
}
//自分のいる場所、駒の名前、動ける場所を出力する
 public void outputMovePlace() {
   System.out.print(getKomaName() + getPlaceA() + "" + getPlaceB()
   + "movePlace:");
```

```
  for(int i:movePlace) {
    System.out.print(i+",");
  }
  System.out.println("");
}
public void setTeban(int tebanN) {
  teban = tebanN;
}
public int getTeban() {
  return teban;
}
public void setPlace(int a,int b) {
  this.placeA = a;
  this.placeB = b;
}
public void setPoint(int pointN) {
  point = pointN;
}
public int getPoint() {
  return point;
}
public void setPointSpecial(int pointSpecialN) {
  pointSpecial = pointSpecialN;
}
public int getPointSpecial() {
  return pointSpecial;
}
public void setMovePlace(ArrayList<Integer> movePlace) {
  this.movePlace = movePlace;
}
public  ArrayList<Integer> getMovePlace() {
  return movePlace;
}
public  ArrayList<Integer> getMovePlaceCloneDeep() {
  ArrayList<Integer> movePlaceClone = new ArrayList<Integer>();
  for(Integer i:movePlace) {
    movePlaceClone.add(i);
  }
  return movePlaceClone;
}
public void addMovePlace(int place){
  movePlace.add(place);
}
public int getPlaceA(){
  return placeA;
}
public int getPlaceB(){
  return placeB;
}
```

```
  public int getKomaNumber(){
    return 0;
  }
  public void setMotigoma(boolean bool){
    this.motigoma = bool;
  }
  public boolean getMotigoma(){
    return motigoma;
  }
  public int getPlace(){
    return placeA*10 + placeB;
  }
  public int getPointTeban(){
    if(teban==1){
      getPoint();
    }else if(teban==2){
      return -1*getPoint();
    }
    System.out.println("error: Koma getPointTeban teban");
    return 0;
  }
  public void changeTeban(){
    if(teban==1){
      teban = 2;
    }else if(teban==2){
      teban = 1;
    }else{
      System.out.println("error: Koma changeTeban teban");
    }
  }
}
```

- Lion クラス

```
package koma;

import board.Kyokumen;

public class Lion extends Koma{

  public Lion(int tebanN)  {
    super(tebanN);
    setPoint(30000000);
  }

  public int getKomaNumber() {
    return 4;
  }
```

```
  public String getKomaName() {
    return "  ラ";
  }

  //動ける場所を決める
  public void decideMovePlace(Kyokumen kyokumen) {
    int a = getPlaceA() * 10;
    int b = getPlaceB();
    CalcKoma calc = new CalcKoma();
    Koma koma = null;
    int[] array = {a+10+b+1,a+10+b,a+10+b-1,a+b+1,a+b-1,a-10+b+1,a-10+b,a-10+b-1};
    for(int i:array) {
      if(calc.isBan(i)) {
        koma = kyokumen.getBanarray(i);
        if(koma == null) {
          addMovePlace(i);
        }else {
          if(koma.getTeban() != getTeban()) {
            addMovePlace(i);
          }
        }
      }
    }
  }
}
```

- Niwatori クラス

```
package koma;

import board.Kyokumen;

public class Niwatori extends Koma {

  public Niwatori(int tebanN){
    super(tebanN);
    setPoint(15000000);
  }

  public int getKomaNumber() {
    return 11;
  }
  public String getKomaName() {
    return "  鶏";
  }
  //動ける場所を決める
  public void decideMovePlace(Kyokumen kyokumen) {
    int difference = 1;//先後によって進むか戻るか
```

```
    if(getTeban()==1) {
      difference = -1;//先手の時
    }
    int a = getPlaceA() * 10;
    int b = getPlaceB();
    int b1 = b + difference;//動く先の場所
    CalcKoma calc = new CalcKoma();
    Koma koma = null;
    int[] array = {a+10+b1,a-10+b1,a+10+b,a-10+b,a+b+1,a+b-1};
    for(int i:array) {
      if(calc.isBan(i)) {
        koma = kyokumen.getBanarray(i);
        if(koma == null) {
          addMovePlace(i);
        }else {
          if(koma.getTeban() != getTeban()) {
            addMovePlace(i);
          }
        }
      }
    }
  }


}
```

- Zou クラス

```
package koma;

import board.Kyokumen;

public class Zou extends Koma{

  public Zou(int tebanN)  {
    super(tebanN);
    setPoint(10000000);
  }
  public int getKomaNumber() {
    return 2;
  }
  public String getKomaName() {
    return "  象";
  }

  //動ける場所を決める
  public void decideMovePlace(Kyokumen kyokumen) {
    int a = getPlaceA() * 10;
```

```
    int b = getPlaceB();
    CalcKoma calc = new CalcKoma();
    Koma koma = null;
    int[] array = {a+10+b+1,a+10+b-1,a-10+b+1,a-10+b-1};
    for(int i:array) {
      if(calc.isBan(i)) {
        koma = kyokumen.getBanarray(i);
        if(koma == null) {
          addMovePlace(i);
        }else {
          if(koma.getTeban() != getTeban()) {
            addMovePlace(i);
          }
        }
      }
    }
  }
}
```

- Calc クラス

```
package main;

public class Calc {
  //文字列が数字かどうか判断する
  public static boolean isNumber(String str) {
    try {
      Integer.parseInt(str);
      return true;
    }catch(NumberFormatException e) {
      return false;
    }
  }
  public static int changeTeban(int teban) {
    //手番を変える 1 なら 2,2 なら 1 にする
    if (teban==1) {
      return 2;
    }else if (teban==2) {
      return 1;
    }
    System.out.println("error:Calc changeTeban");
    return 1;
  }
}
```

- Kihu クラス

```
package main;
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;

import board.Kyokumen;
import board.Move;

//棋譜を保存するクラス
public class Kihu {
  private ArrayList<Move> kihuList;
  private Kyokumen firstKyokumen;
  //棋譜を読み込んだ時に作る。保存せず、対局が終わった時にも作る。
  private Kyokumen lastKyokumen;

  public Kihu() {
    kihuList = new ArrayList<Move>();
  }
  public Kihu(Kyokumen kyokumen) {
    kihuList = new ArrayList<Move>();
    firstKyokumen = kyokumen.clone();
    lastKyokumen = null;
  }
  public Kihu(ArrayList<Move> kihuList,Kyokumen firstKyokumen) {
    this.kihuList = kihuList;
    this.firstKyokumen = firstKyokumen;
    lastKyokumen = null;
  }
  public Kihu cloneKihu() {
    ArrayList<Move> kihuListClone = new ArrayList<Move>();
    for(Move move:kihuList) {
      kihuListClone.add(move);
    }
    return new Kihu(kihuListClone,firstKyokumen.clone());
  }
  public void makeLastKyokumen() {
    Kyokumen lastK = firstKyokumen.clone();
    for(Move move:kihuList) {
      lastK.moveNextKyokumen(move);
    }
    lastKyokumen = lastK;
  }
  /**
   * 最初の局面から保存されている棋譜を読み込む
   */
  public void readKihuFromFirstKyokumen(String inputFile) {
```

```
  System.out.println("棋譜を読み込みます。最初の局面が保存されている。");
  kihuList.clear();
  Move move = new Move();
  Move inputMove = null;
  File file = new File(inputFile);
  try {
    FileReader filereader = new FileReader(file);
    BufferedReader br = new BufferedReader(filereader);
    firstKyokumen.readKyokumenFromFirstKyokumen(br);
    String str = br.readLine();
    while(str != null) {
      inputMove = move.getInputMove(str);
      if(inputMove == null) {
        break;
      }else {
        kihuList.add(inputMove);
      }
      str = br.readLine();
    }
    br.close();
    makeLastKyokumen();
  } catch (IOException e) {
    // TODO 自動生成された catch ブロック
    e.printStackTrace();
  }
}
//棋譜を読み込む
 public void inputKihu(String inputFile){
  System.out.println("棋譜を読み込みます。");
  kihuList.clear();
  Move move = new Move();
  Move inputMove = null;
  File file = new File(inputFile);
  try {
    FileReader filereader = new FileReader(file);
    BufferedReader br = new BufferedReader(filereader);
    String str = br.readLine();
    while(str != null) {
      inputMove = move.getInputMove(str);
      if(inputMove == null) {
        break;
      }else {
        kihuList.add(inputMove);
      }
      str = br.readLine();
    }
    br.close();
  } catch (IOException e) {
    // TODO 自動生成された catch ブロック
```

```
    e.printStackTrace();
  }
}
/**
 * 棋譜ファイルを最初の局面から保存する。
 * 最初の局面を保存して、改行して、棋譜保存する。
 * @param saveFile
 */
public void saveKihuFromFirstKyokumen(String saveFile) {
  System.out.println("棋譜を"+saveFile+"に保存します。最初の局面も保存します。");
  File file = new File(saveFile);
  try {
    FileWriter fileWriter = new FileWriter(file);
    ///保存する
    firstKyokumen.saveKyokumenFromFileWriter(fileWriter);
    for(Move move:kihuList) {
      fileWriter.write(move.getMoveKihu());
    }
    fileWriter.close();
  } catch (IOException e) {
    // TODO 自動生成された catch ブロック
    e.printStackTrace();
    System.out.println("保存に失敗しました。");
  }
}
/**
 * 棋譜をファイルに保存する。
 * @param saveFile
 */
public void saveKihuFile(String saveFile) {
  System.out.println("棋譜を"+saveFile+"に保存します。");
  File file = new File(saveFile);
  try {
    FileWriter fileWriter = new FileWriter(file);
    for(Move move:kihuList) {
      fileWriter.write(move.getMoveKihu());
    }
    fileWriter.close();
  } catch (IOException e) {
    e.printStackTrace();
    System.out.println("保存に失敗しました。");
  }
  System.out.println("保存されました。");
}

public void outputKihu() {
  System.out.println("棋譜を出力します。firstKyokumen");
  firstKyokumen.outputKyokumen();
  System.out.println("棋譜を出力します。kihu");
```

```
    for(Move move:kihuList) {
      move.outputMoveKihu();
    }
  }
  public void addMove(Move move) {
    kihuList.add(move);
  }
  public ArrayList<Move> getKihuList(){
    return kihuList;
  }
  public Move getLastMove() {
    if(kihuList==null || kihuList.isEmpty()) {
      return null;
    }
    return kihuList.get(kihuList.size()-1);
  }
  public void removeLastMove() {
    kihuList.remove(kihuList.size()-1);
  }
  public Kyokumen getFirstKyokumen() {
    return firstKyokumen;
  }
  public Kyokumen getLastKyokumen() {
    return lastKyokumen;
  }
  //棋譜を保存する。before*100+after *100 + 駒の番号十の位は成駒なら 1, なる時は 2, 成らずは 3, 普通は 0
}
```

- MainSwing クラス

```
package main;

import java.util.ArrayList;

import board.Board;
import board.Kyokumen;
import swing.ActionKihuListener;
import swing.ActionListener;
import swing.ActionListenerMenu;
import swing.FrameS;
import swing.StateGame;

public class MainSwing {

  public static void main(String[] args) {
    FrameS frame = new FrameS();
    Board board = new Board(new Kyokumen(),new Player(),new ArrayList<Kyokumen>());
    StateGame state = new StateGame(board,frame);
```

```
    ActionListener actionListener = new ActionListener(board,frame,state);
    ActionListenerMenu actionListenerMenu = new ActionListenerMenu(board,frame,state);
    ActionKihuListener actionKihuListener = new ActionKihuListener(board,frame, state);
    frame.setActionListener(actionListener, actionListenerMenu,actionKihuListener);
    frame.swing2();
    state.setNotGaming();
  }
}
```

- Main クラス

```
package main;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

import board.Board;
import board.Kyokumen;
import board.Move;

public class Main {

  public static void main(String[] args) throws IOException{
    System.out.println("対局を始めます。");
    Kyokumen kyokumen = new Kyokumen();

    //先手後手のプレイヤー決める
    Player player = new Player();
    player.decidePlayer();
    ArrayList<Kyokumen> kyokumenList = new ArrayList<Kyokumen>();
    Board board = new Board(kyokumen,player,kyokumenList);

    Kihu kihu = new Kihu();
    kihu.inputKihu("inputhu.txt");//棋譜をインプットするとき

    kyokumenList.add(board.getKyokumen().clone());
    for(Move move:kihu.getKihuList()) {
      board.moveNextKyokumen(move);
      kyokumenList.add(board.getKyokumen().clone());
    }
    kyokumenList.remove(kyokumenList.size()-1);

    //対局中。コンピュータと人間を変えたりするときは対局を中断か投了かしないとできない。
    int i = 1;
    while(i<1000) {
      System.out.println(i+"手目の入力をしてください。"+board.getTebanS());
      //全ての駒の movePlace を決める。
      board.decideAllKomaMovePlace();
```

```
    board.outputKyokumen();

    kyokumenList.add(board.getKyokumen().clone());
    //対局終了、中断、投了、まった（二手戻る）
    int checkInt = checkInterruption();//対局を続けるなら 1、待ったなら 2、投了 3
    if(checkInt==2) {
      if(i<2) {
        System.out.println("error: Main back i");
        break;
      }
      //待った、二手戻る。
      Move backMove = kihu.getLastMove();
      kihu.removeLastMove();
      board.moveBackKyokumen(backMove);
      backMove = kihu.getLastMove();
      kihu.removeLastMove();
      board.moveBackKyokumen(backMove);
      i -= 2;
    }else if(checkInt==1){
      //次の手
      board = board.goNextBoard();
      if(board == null) {
        System.out.println("負けました。");
        break;
      }
      //棋譜を保存する
      kihu.addMove(board.getBeforeMove());
      System.out.println("指した手は以下です。");
      board.getBeforeMove().outputMoveKihu();///
      //kihu.outputKihu();
      i++;
    }else if(checkInt==3) {
      //投了
      System.out.println("負けました。");
      break;
    }else if(checkInt==4) {
      //局面保存
      String saveFile = "kyokumen.txt";
      ///System.out.println("局面を"+saveFile+"に保存します。");
      board.saveKyokumen(saveFile);
    }
  }
   checkOutputKihu(kihu);
   System.out.println("プログラムを終了します。");
}
/**
 * 対局を続けるなら 1、待ったなら 2、投了 3,
 * @return
 */
```

```
  private static int checkInterruption() {
    Scanner scan = new Scanner(System.in);
    String str;
    while(true) {
      System.out.println("対局を続けますか？ n:次の手, back:待った, loss:負けました。 save:局面保存");
      str = scan.next();
      if(str.equals("n")) {
        return 1;
      }else if(str.equals("back")) {
        //待った、行って戻る
        return 2;
      }else if(str.equals("loss")) {
        return 3;
      }else if(str.equals("save")) {
        return 4;
      }
      System.out.println("入力が違います。");
    }
  }
  //棋譜を出力するか確認する
  public static void checkOutputKihu(Kihu kihu) {
    String saveFile = "output.txt";
    String outputStr = "棋譜を出力（保存）しますか？"+saveFile;
    if(checkYes(outputStr)) {
      //棋譜保存
      kihu.saveKihuFile(saveFile);
    }
  }
  public static boolean checkYes(String outputStr) {
    Scanner scan = new Scanner(System.in);
    String str;
    while(true) {
      System.out.println(outputStr);
      System.out.println("y:yes,n:no を入力してください。");
      str = scan.next();
      if(str.equals("y")) {
        return true;
      }else if(str.equals("n")) {
        return false;
      }
      System.out.println("入力が正しくありません。");
    }
  }
}
```

- Player クラス

```
package main;
```

```
import java.util.Scanner;

//先手後手が人か CP かを確認して保存する
public class Player {
  private boolean playerSente;//true ならコンピュータ
  private boolean playerGote;
  public Player() {
    playerSente = false;
    playerGote = false;
  }
  /**
   * 先手と後手のプレイヤー決める。
   * コンピュータか人か。
   * いつでも変えられるようにする。
   * @return
   */
  public void decidePlayer(){
    //先手後手のプレイヤーが人かコンピューターか入力させる
    Scanner scan = new Scanner(System.in);
    String str;
    //先手決める
    while (true) {
      System.out.println("先手番を入力してください。人:human, コンピューター:cp");
      str = scan.next();
      if(str.equals("human")) {
        playerSente = false;
        break;
      }else if(str.equals("cp")) {
        playerSente = true;
        break;
      }else {
        System.out.println("入力が正しくありません!");
      }
    }
    //後手決める
    while (true) {
      System.out.println("後手番を入力してください。人:human, コンピューター:cp");
      str = scan.next();
      if(str.equals("human")) {
        playerGote = false;
        break;
      }else if(str.equals("cp")) {
        playerGote = true;
        break;
      }else {
        System.out.println("入力が正しくありません!");
      }
    }
```

```
  //scan.close();
   outputPlayer();
}
 public void outputPlayer() {
   if(playerSente) {
     System.out.println("先手番はコンピュータです。");
  }else {
     System.out.println("先手番は人間です。");
  }
   if(playerGote) {
     System.out.println("後手番はコンピュータです。");
  }else {
     System.out.println("後手番は人間です。");
  }
}
 public String getPlayerBoth() {
   return getPlayerSenteString() + getPlayerGoteString();
}
 private String getPlayerSenteString() {
   if(playerSente) {
     return "先手番はコンピュータです。";
  }else {
     return "先手番は人間です。";
  }
}
 private String getPlayerGoteString() {
   if(playerGote) {
     return "後手番はコンピュータです。";
  }else {
     return "後手番は人間です。";
  }
}
 public boolean getPlayerSente() {
   return playerSente;
}
 public boolean getPlayerGote() {
   return playerGote;
}
 public boolean getPlayer(int teban) {
   if(teban==1) {
     return playerSente;
  }else if(teban==2) {
     return playerGote;
  }else {
     System.out.println("error:Player getPlayer");
     return false;
  }
}
 public void setPlayerSente(boolean playerSente) {
```

```
    this.playerSente = playerSente;
  }
  public void setPlayerGote(boolean playerGote) {
    this.playerGote = playerGote;
  }
  /**
   * cp が含まれていれば true
   * @return
   */
  public boolean containCp() {
    return playerSente || playerGote;
  }



}
```

- ActionKihuListener クラス

```
package swing;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import board.Board;
import board.Move;

public class ActionKihuListener implements ActionListener{
  private Board board;
  private FrameS frame;
  private StateGame state;

  public ActionKihuListener(Board board,FrameS frame,StateGame state) {
    this.board = board;
    this.frame = frame;
    this.state = state;
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    if(e.getActionCommand().equals("最初") && state.checkReadingKihu()) {
      System.out.println("back to first kyokumen");
     //board = new Board(state.getKihu().getFirstKyokumen().clone());
      board.setKyokumen(state.getKihu().getFirstKyokumen().clone());
      board.resetExceptKyokumen();
      state.getKihu().getFirstKyokumen().outputKyokumen();
     //board.outputKyokumen();
      frame.setKyokumen(board.getKyokumen());
```

```
      frame.changeLabelTeban(board.getTebanS());
      state.setTesuuFirst();
    }else if(e.getActionCommand().equals("次へ") && state.checkReadingKihu()) {
      Move move = state.getMoveNext();
      if(move != null) {
        board.moveNextKyokumen(move);
        frame.changePanel(move.getAfterPlaceA(), move.getAfterPlaceB(), board.getKyokumen());
        frame.changePanel(move.getBeforePlaceA(), move.getBeforePlaceB(), board.getKyokumen());
        if(move.getGetKoma() != 0) {
          //駒を取っていた時
          frame.changePanel((3-board.getTeban())*10, move.getGetKoma()%10, board.getKyokumen());
        }
        frame.changeLabelTeban(board.getTebanS());
        state.setTesuuAdd();
        state.getKihu().getFirstKyokumen().outputKyokumen();///
      }
    }else if(e.getActionCommand().equals("前へ") && state.checkReadingKihu()) {
      //局面を戻す
      Move move = state.getMoveBack();
      if(move != null) {
        board.moveBackKyokumen(move);
        frame.changePanel(move.getAfterPlaceA(), move.getAfterPlaceB(), board.getKyokumen());
        frame.changePanel(move.getBeforePlaceA(), move.getBeforePlaceB(), board.getKyokumen());
        if(move.getGetKoma() != 0) {
          //駒を取っていた時
          frame.changePanel((board.getTeban())*10, move.getGetKoma()%10, board.getKyokumen());
        }
        frame.changeLabelTeban(board.getTebanS());
        state.setTesuuDecrease();
      }
    }else if(e.getActionCommand().equals("最後") && state.checkReadingKihu()) {
      board.setKyokumen(state.getKihu().getLastKyokumen().clone());
      board.resetExceptKyokumen();
      frame.setKyokumen(board.getKyokumen());
      frame.changeLabelTeban(board.getTebanS());
      state.setTesuuEnd();
    }

  }

}
```

- ActionListenerMenu クラス

```
package swing;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import javax.swing.JFrame;
import javax.swing.JOptionPane;

import board.Board;
import board.FreeKyokumen;
import board.Move;
import calcValue.TumiCheck;
import calcValue.Value;
import main.Kihu;

public class ActionListenerMenu extends JFrame implements ActionListener{
  private Board board;
  private FrameS frame;
  private StateGame state;//状態を持つ。1:対局中,2:対局していない
  private static String saveKyokumenName = "kyokumenS.txt";//局面を保存する時のファイル名
  private static String readKyokumenName = "kyokumenR.txt";//局面を読み込む時のファイル名
  private static String saveKihuName = "kihuS.txt";
  private static String readKihuName = "kihuR.txt";
  public ActionListenerMenu(Board board,FrameS frame,StateGame state) {
    this.board = board;
    this.frame = frame;
    this.state = state;
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    if(e.getActionCommand().equals("新規対局") && state.checkNotGaming()) {
       int option = JOptionPane.showConfirmDialog(this, "データは失われます。新規対局を始めます
か？", "最終確認", JOptionPane.YES_NO_OPTION,JOptionPane.PLAIN_MESSAGE);
       if (option == JOptionPane.YES_OPTION){
           setFirstKyokumen();
       }else if (option == JOptionPane.NO_OPTION){
           System.out.println("新規対局は中止しました");
       }
   }else if(e.getActionCommand().equals("投了") && state.checkGamingPlayer()) {
     JOptionPane.showMessageDialog(this, board.getTebanS()+"の負けです。", "投了",JOptionPane.PLAIN_MESSAGE);
     board.getKihu().makeLastKyokumen();
     state.setReadingKihu(board.getKihu().getKihuList().size());
     frame.setReadingKihu();
   }else if(e.getActionCommand().equals("局面評価")) {
     //局面を評価して値を表示. 警告で表示?
     //とりあえず現在の局面だけの評価値,n 手先の評価値まで求められるとよい。
     Value value = new Value();
     value.calcValuePresent(board.getKyokumen(), board.getKyokumenList(), board.getBeforeMove());
     String determine = "す。";
     if(!value.getDeterme()) {
       determine = "せん。";
     }
```

```
      JOptionPane.showMessageDialog(this, "探索なしの局面の評価値!\n 勝敗は決まっていま"+determine+"\n
評価値は:"+value.getValue(),"この局面の評価値",JOptionPane.PLAIN_MESSAGE);
    }else if(e.getActionCommand().equals("待った")) {
      //待ったが押されたら一手戻る。cp と対局する時は 2 手戻る。cp が待ったを押すことはない.beforeMove がなけ
れば戻らない。
      if(board.containCp()) {
        matta();
      }
      matta();
    }else if(e.getActionCommand().equals("局面保存")) {
      //board にある局面を保存する。
      board.saveKyokumen(saveKyokumenName);
       JOptionPane.showMessageDialog(this, saveKyokumenName+" に局面が保存されました。", "局面保
存",JOptionPane.PLAIN_MESSAGE);
    }else if(e.getActionCommand().equals("局面読み込み")) {
      //局面読み込みは対局していない時しかできない。現在の情報が失われる確認をする。
      if(state.checkNotGaming()){
         int option = JOptionPane.showConfirmDialog(this, "データは失われます。局面を読み込みます
か？", "最終確認", JOptionPane.YES_NO_OPTION,JOptionPane.PLAIN_MESSAGE);
          if (option == JOptionPane.YES_OPTION){
              board.readKyokumen(readKyokumenName);
              frame.setKyokumen(board.getKyokumen());
          }else if (option == JOptionPane.NO_OPTION){
              System.out.println("局面読み込みは中止しました");
          }
          state.setNotGaming();
          frame.setKihuPanelNotVisible();
      }
    }else if(e.getActionCommand().equals("棋譜保存")) {
      //棋譜再生中なら state クラスにある棋譜を保存する。
      if(state.checkReadingKihu()) {
        state.getKihu().saveKihuFromFirstKyokumen(saveKihuName);
      }else {
        board.getKihu().saveKihuFromFirstKyokumen(saveKihuName);
      }
        JOptionPane.showMessageDialog(this, saveKihuName+" に棋譜が保存されました。", "棋譜保
存",JOptionPane.PLAIN_MESSAGE);
    }else if(e.getActionCommand().equals("棋譜読み込み")) {
      //棋譜読み込みは対局していない時しかできない。現在の情報が失われる確認をする。
      if(state.checkNotGaming()){
         int option = JOptionPane.showConfirmDialog(this, "データは失われます。棋譜を読み込みます
か？", "最終確認", JOptionPane.YES_NO_OPTION,JOptionPane.PLAIN_MESSAGE);
          if (option == JOptionPane.YES_OPTION){
              board.readKihu(readKihuName);
              frame.setKyokumen(board.getKyokumen());
              state.setReadingKihu(0);
              frame.setKihuPanelVisible();
          }else if (option == JOptionPane.NO_OPTION){
              System.out.println("棋譜読み込みは中止しました");
```

```
        }
      }
    }else if(e.getActionCommand().equals("現在の局面から対局")) {
      //対局していない時なら現在の局面から対局開始。棋譜読み込み中なら棋譜の続きから対局することもできる。
      if(state.checkNothin()) {
        setThisKyokumen();
      }else if(state.checkReadingKihu()) {
         int option = JOptionPane.showConfirmDialog(this, "データは失われます。対局を始めますか？", "
最終確認", JOptionPane.YES_NO_OPTION,JOptionPane.PLAIN_MESSAGE);
          if (option == JOptionPane.YES_OPTION){
              int option_2 = JOptionPane.showConfirmDialog(this, "棋譜の続きから対局をしますか？", "
最終確認", JOptionPane.YES_NO_OPTION,JOptionPane.PLAIN_MESSAGE);
            if (option_2 == JOptionPane.YES_OPTION){
                setThisKihu();
            }else if (option_2 == JOptionPane.NO_OPTION){
                setThisKyokumen();
            }
          }else if (option == JOptionPane.NO_OPTION){
              System.out.println("対局再開を中止しました");
          }
      }
    }else if(e.getActionCommand().equals("自由")) {
      //自由に駒を動かせるようにする
      if(state.checkNotGaming()) {
        //自由開始
         int option = JOptionPane.showConfirmDialog(this, "データは失われます。自由局面をはじめます
か？", "最終確認", JOptionPane.YES_NO_OPTION,JOptionPane.PLAIN_MESSAGE);
          if (option == JOptionPane.YES_OPTION){
              System.out.println("自由局面スタート!");
              FreeKyokumen freeKyokumen = new FreeKyokumen(board.getKyokumen());
              frame.getActionListener().setFreeKyokumen(freeKyokumen);
              state.setFreeBefore();
              frame.setFreeBefore();
          }else if (option == JOptionPane.NO_OPTION){
              System.out.println("自由局面を中止しました");
          }
      }else if(state.checkFreeBefore() || state.checkFreeAfter()) {
        frame.outputMessage("自由対局を終わります。", "");
        state.setNotGaming();
        board.setFirst();
        int teban = ask("この局面での手番は先手にしますか？","手番確認");
        if(teban!=2) {
         //エラーは先手番にする
         teban = 1;
        }
        board.setKyokumen(frame.getActionListener().getFreeKyokumen().getKyokumen(teban));///
        frame.endGame();
      }
    }else if(e.getActionCommand().equals("詰みチェック")) {
```

```
    System.out.println("詰みチェックを始めます。");
    Move move = TumiCheck.checkTumiNTe(board.getKyokumen().clone(), 7);
    if(move!=null) {
      frame.outputMessage("詰みがあります.\n"+move.getMoveKihu(), "詰み");
      System.out.println("詰みがあります。");
      move.outputMoveKihu();
    }else {
      frame.outputMessage("詰みはありません。", "詰みなし");
      System.out.println("詰みがありません。");
    }
  }
}
/**
 * 質問をして yes なら 1,no なら 2
 * 変なのは 0
 * @return
 */
private int ask(String question,String title) {
  int option = JOptionPane.showConfirmDialog(this, question, title,
    JOptionPane.YES_NO_OPTION,JOptionPane.PLAIN_MESSAGE);
    if (option == JOptionPane.YES_OPTION){
        return 1;
    }else if (option == JOptionPane.NO_OPTION){
        return 2;
    }
    return 0;
}
private void matta() {
  Move backMove = board.getKihu().getLastMove();
  if(backMove != null) {
    board.getKihu().removeLastMove();
    board.deleteLastKyokumenList();
    board.moveBackKyokumen(backMove);
    frame.changePanel(backMove.getAfterPlaceA(), backMove.getAfterPlaceB(), board.getKyokumen());
    frame.changePanel(backMove.getBeforePlaceA(), backMove.getBeforePlaceB(), board.getKyokumen());
    if(backMove.getGetKoma() != 0) {
      //駒を取っていた時
      frame.changePanel((board.getTeban())*10, backMove.getGetKoma()%10, board.getKyokumen());
    }
    frame.changeLabelTeban(board.getTebanS());
  }
}
private void setThisKyokumen() {
  board.resetExceptKyokumen();
  frame.setKyokumen(board.getKyokumen());
  //先後決める
  if(!setPlayer()) {
    System.out.println("対局は中止しました");
    return;
```

```
    }
    frame.startGame();
    JOptionPane.showMessageDialog(this, board.getPlayerBoth()+"\n よろしくお願いします。", "対局開
始",JOptionPane.PLAIN_MESSAGE);
    state.setGaming(board.getTebanPlayer());
    board.getKyokumen().setKomaArray();
    board.decideAllKomaMovePlace();
  }
  private void setThisKihu() {
   //棋譜を引き継いで対局再開
    board.setKyokumen(state.getKihu().getLastKyokumen());
    setThisKyokumen();
    Kihu kihu = state.getKihu();
    board.setKihu(kihu);
    board.setBeforeMove(kihu.getKihuList().get(kihu.getKihuList().size()-1));
  }
  /**
   * 新規対局になった時に,board,frame を初期画面にする。
   * 先手後手を人かどうかも決める。
   */
  private void setFirstKyokumen() {
    board.setFirst();
    frame.setKyokumen(board.getKyokumen());
    //先後決める
    if(!setPlayer()) {
      System.out.println("新規対局は中止しました");
      return;
    }
    //先後決まったので対局開始
    frame.startGame();
    JOptionPane.showMessageDialog(this, board.getPlayerBoth()+"\n よろしくお願いします。", "対局開
始",JOptionPane.PLAIN_MESSAGE);
    state.setGaming(board.getTebanPlayer());
  }
  /**
   * 先後を決める。
   * @return 決まらなければ false
   */
  private boolean setPlayer() {
    String selectvalues[] = {"人間, 人間", "人間, コンピューター", "コンピューター, 人間", "コンピュー
ター, コンピューター"};
    String sengo = "";
      Object value = JOptionPane.showInputDialog(this, "先 手, 後 手 を 決 め て く だ さ い!","先 後 選
択", JOptionPane.INFORMATION_MESSAGE,null, selectvalues, selectvalues[0]);
    if (value == null){
       return false;
    }else{
      sengo = (String)value;
      if(sengo.equals("人間, 人間")) {
```

```
        board.setPlayerSente(false);
        board.setPlayerGote(false);
      }else if(sengo.equals("人間, コンピューター")) {
        board.setPlayerSente(false);
        board.setPlayerGote(true);
      }else if(sengo.equals("コンピューター, 人間")) {
        board.setPlayerSente(true);
        board.setPlayerGote(false);
      }else if(sengo.equals("コンピューター, コンピューター")) {
        board.setPlayerSente(true);
        board.setPlayerGote(true);
      }else {
        System.out.println("error:setFirstKyokumen in ActionListenerMenu class no choise = "+sengo);
        return false;
      }
    }
    return true;
  }

}
```

- ActionListener クラス

```
package swing;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JFrame;
import javax.swing.JOptionPane;

import board.Board;
import board.FreeKyokumen;
import board.Move;
import koma.Koma;
import main.Calc;

public class ActionListener extends JFrame implements ActionListener{
  private Board board;
  private FrameS frame;
  private StateGame state;//状態を持つ。1:対局中,2:対局していない
  private FreeKyokumen freeKyokumen;
  public ActionListener(Board board,FrameS frame,StateGame state) {
    this.board = board;
    this.frame = frame;
    this.state = state;
    freeKyokumen = null;
  }
```

```
  @Override
  public void actionPerformed(ActionEvent e) {
   ///System.out.println(e.getActionCommand()+": player"+board.getPlayerBoth());
    if(state.checkChoseBeforeKoma() && checkBeforeKoma(e.getActionCommand())) {
     //人間が手番でまだ駒を選択していない。選択された駒は手番の駒である。
      int place = Integer.parseInt(e.getActionCommand());
      state.setBeforePlace(place);//beforePlace を設定して駒を選択した状態にする。
     //選択した駒の背景変える
      frame.changeButtonColor(place);
    }else if(state.checkChoseAfterPlace()) {
     //人間が手番で beforePlace を選択している。afterPlace が選択された
      if(checkAfterPlace(e.getActionCommand())) {
       //動かせる場所なので動かす。
        int afterPlace = Integer.parseInt(e.getActionCommand());
        Move move = makeMove(state.getBeforePlace(), afterPlace);
        board.moveNextKyokumen(move);
        frame.changePanel(move.getAfterPlaceA(), move.getAfterPlaceB(), board.getKyokumen());
        frame.changePanel(move.getBeforePlaceA(), move.getBeforePlaceB(), board.getKyokumen());
        if(move.getGetKoma() != 0) {
         //駒を取っていた時
          frame.changePanel((3-board.getTeban())*10, move.getGetKoma()%10, board.getKyokumen());
        }
        int sennitite = board.checkSennitite();
        if(sennitite==1) {
         //千日手
          JOptionPane.showMessageDialog(this, "千日手です。", "千日手",JOptionPane.PLAIN_MESSAGE);
          state.setNotGaming();
          frame.endGame();
        }else if(sennitite==2) {
         //連続ライオン手の千日手。手番がライオン手している。
          JOptionPane.showMessageDialog(this, board.getTebanS()+"の反則負けです。", "反則:連続ライオ
ン手",JOptionPane.PLAIN_MESSAGE);
          state.setNotGaming();
          frame.endGame();
        }else if(sennitite==3) {
         //連続ライオン手の千日手。手番がライオン手されている。
          JOptionPane.showMessageDialog(this, board.getTebanOppositeS()+"の反則負けです。", "反則:
連続ライオン手",JOptionPane.PLAIN_MESSAGE);
          state.setNotGaming();
          frame.endGame();
        }
        state.setGaming(board.getTebanPlayer());
      }else {
       //動かせない場所を選択したら最初の選択を解除
        frame.changePanel(state.getBeforePlace()/10, state.getBeforePlace()%10, board.getKyokumen());
        state.setGaming(board.getTebanPlayer());
      }
    }else if(state.checkFreeBefore() && checkFreeBefore(e.getActionCommand())) {
```

```
    int place = Integer.parseInt(e.getActionCommand());
    state.setBeforePlace(place);//beforePlace を設定して駒を選択した状態にする。
    //選択した駒の背景変える
    frame.changeButtonColor(place);
    state.setFreeAfter();
  }else if(state.checkFreeAfter() && checkFreeAfter(e.getActionCommand())) {
    ///System.out.println("自由局面 after!"+state.getBeforePlace());
    //駒を動かす。
    int placeAfter = Integer.parseInt(e.getActionCommand());
    int placeBefore = state.getBeforePlace();
    ///System.out.println("自由局面 after place "+placeBefore+" af: "+placeAfter);
    Koma koma = freeKyokumen.move(placeBefore, placeAfter);
    frame.changePanel(placeBefore/10, placeBefore%10, freeKyokumen);
    if(koma==null) {
      frame.changePanel(placeAfter/10, placeAfter%10, freeKyokumen);
    }else {
      //駒を駒台に移す時
      frame.changePanel(placeAfter/10, koma.getKomaNumber()%10, freeKyokumen);
    }
    state.setFreeBefore();
  }
}
 private boolean checkFreeAfter(String placeS) {
   int afterPlace = 0;
   if(Calc.isNumber(placeS)) {
     afterPlace = Integer.parseInt(placeS);
  }else {
     System.out.println("error: button after place ,free, "+placeS);
     return false;
  }
  //同じ場所をさしていたら動かせる。盤上なら成反転, 駒台なら何もしない
   if(afterPlace == state.getBeforePlace()) {
     return true;
  }
   if(afterPlace/100 > 0) {
    //駒台を選択している。駒台には必ず動ける?自分の駒台から自分の駒台は無駄だが操作する。
     return true;
  }
  //盤上は駒がなければ動ける
   Koma koma = freeKyokumen.getKomaFromPlace(afterPlace);
   if(koma == null) {
     return true;
  }
  //盤上に駒があれば正しくない。同じ場所をさしていることはない。
   return false;
}
/**
 * place が正しいかどうかを判断する。
 * @param place
```

```
 * @return 正しければ true
 */
 private boolean checkFreeBefore(String placeS) {
   int place = 0;
   if(Calc.isNumber(placeS)) {
     place = Integer.parseInt(placeS);
   }else {
     System.out.println("error: button before place free, "+placeS);
     return false;
   }
   if(freeKyokumen==null) {
     System.out.println("error: no free kyokumen , ");
   }
   Koma koma = freeKyokumen.getKomaFromPlace(place);
   if(koma == null) {
     System.out.println("error: button before place no koma, free,"+placeS);
     return false;
   }
   //駒があれば ok
   return true;
 }
 /**
  * Move クラスを作る
  * @return
  */
 private Move makeMove(int beforePlace,int afterPlace){
   Move move = new Move(beforePlace/10,beforePlace%10,afterPlace/10,afterPlace%10);
   //なれるかどうかを確認する
   Koma koma = board.getKyokumen().getKomaFromPlace(state.getBeforePlace());
   if(move.checkAbleNaru(koma)) {
     int option = JOptionPane.showConfirmDialog(this, "成りますか？", "成る確認",
       JOptionPane.YES_NO_OPTION,JOptionPane.PLAIN_MESSAGE);
       if (option == JOptionPane.YES_OPTION){
           move.setNaru(true);
       }else if (option == JOptionPane.NO_OPTION){
           move.setNaru(false);
       }
   }
   return move;
 }
 /**
  * 駒を選択した後に動かせる場所かどうかをチェック
  * 動かせる場所なら true
  * @return
  */
 private boolean checkAfterPlace(String placeS) {
   int afterPlace = 0;
   if(Calc.isNumber(placeS)) {
     afterPlace = Integer.parseInt(placeS);
```

```
  }else {
    System.out.println("error: button after place , "+placeS);
    return false;
  }
  if(afterPlace/100 > 0) {
   //駒台を選択している。
    return false;
  }
  //beforePlace の駒をつくる
  Koma koma = board.getKyokumen().getKomaFromPlace(state.getBeforePlace());
  if(koma == null) {
    return false;
  }
  if(koma.getMotigoma()) {
   //持ち駒ならそこに駒がなければよい
    Koma afterKoma = board.getKyokumen().getKomaFromPlace(afterPlace);
    if(afterKoma==null) {
      return true;
    }
    return false;
  }else {
    if(koma.containMovePlace(afterPlace)) {
      return true;
    }
  }
  return false;
}
/**
 * 人間が手番で駒が選択されていない時に選択できる駒かどうかを判断する。
 * 選択できれば true, 選択できない (相手の駒) なら false
 * @return
 */
private boolean checkBeforeKoma(String placeS) {
  int place = 0;
  if(Calc.isNumber(placeS)) {
    place = Integer.parseInt(placeS);
  }else {
    System.out.println("error: button before place , "+placeS);
    return false;
  }
  Koma koma = board.getKyokumen().getKomaFromPlace(place);
  if(koma == null) {
    System.out.println("error: button before place no koma, "+placeS);
    return false;
  }else {
    if(koma.getTeban()==board.getTeban()) {
      return true;
    }
  }
```

```
    return false;
  }
  public void setFreeKyokumen(FreeKyokumen freeKyokumen) {
    this.freeKyokumen = freeKyokumen;
  }
  public FreeKyokumen getFreeKyokumen() {
    return freeKyokumen;
  }

}
```

- BanPanel クラス

```
package swing;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Image;
import java.awt.Window;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.LineBorder;

import board.Ban;
import board.Kyokumen;
import koma.Koma;

public class BanPanel extends JPanel{
  private GridBagLayout gridBagLayout;
  private JPanel[][] panelArray;
  private Window him;

  public BanPanel(Window her) {
    him = her;
  }

  public void changePanel(int a,int b,Kyokumen kyokumen,ActionListener actionListener) {
    GridBagLayout gbl = gridBagLayout;
    GridBagConstraints gbc = new GridBagConstraints();
    Koma koma = kyokumen.getKomaFromPlace(a, b);
    if(koma == null) {
      JPanel panel0 = panelArray[a-1][b-1];
```

```
    JLabel l = new JLabel("hoge");
    panel0.add(l,BorderLayout.NORTH);
    panel0.setVisible(false);
    JPanel panel = new JPanel();
    panel.setOpaque(false);
    panel.setPreferredSize(new Dimension(63,70));
    panel.setBorder(new LineBorder(Color.black,1));
    gbc.gridx = 3-a;//左右反転
    gbc.gridy = b-1;
    gbl.setConstraints(panel, gbc);
    JButton button = new JButton();
    button.setPreferredSize(new Dimension(63,70));
    button.setBackground(Color.black);///
    //actionlistener 加える
    button.addActionListener(actionListener);
    button.setActionCommand(""+a+""+b);//盤の番号。76 とか
    panel.add(button,BorderLayout.CENTER);
    panel.setBackground(Color.lightGray);
    add(panel);
    panelArray[a-1][b-1] = panel;
  }else {
    JPanel panel0 = panelArray[a-1][b-1];
    panel0.setVisible(false);
    JPanel panel = new JPanel();
    panel.setOpaque(false);
    panel.setPreferredSize(new Dimension(63,70));
    panel.setBorder(new LineBorder(Color.black,1));
    gbc.gridx = 3-a;//左右反転
    gbc.gridy = b-1;
    gbl.setConstraints(panel, gbc);
    ///駒の画像を表示させる。
    ImageIcon icon = MakeKomaImage.makeKomaImage(koma.getKomaNumber(), koma.getTeban(), him);
    Image image = icon.getImage().getScaledInstance(54, 60, Image.SCALE_DEFAULT);
    icon = new ImageIcon(image);
    JButton button = new JButton(icon);
    button.setContentAreaFilled(false);//背景透明化
    button.setBorderPainted(false);
    //actionlistener 加える
    button.addActionListener(actionListener);
    button.setActionCommand(""+a+""+b);//盤の番号。76 とか
    panel.add(button,BorderLayout.CENTER);
    panel.setBackground(Color.lightGray);
    add(panel);
    panelArray[a-1][b-1] = panel;
  }
}
 public void changeColor(int a,int b) {
   if(0<a && a<10 && 0<b && b<10) {
     JPanel panel = panelArray[a-1][b-1];
```

```
    if(panel != null) {
      panel.setOpaque(true);
    }
  }
}
 public void setBanPanel() {
  setBounds(220,50,415,460);//横、縦、幅、高さ
  setBackground(new Color(96,255,128));
  GridBagLayout gbl = new GridBagLayout();
  gridBagLayout = gbl;
  setLayout(gbl);
  makeMasu(gbl);
  panelArray = new JPanel[3][4];//00-23
  for(int i=0;i<3;i++) {
    for(int j=0;j<4;j++) {
      panelArray[i][j] = null;
    }
  }
}
 private void makeMasu(GridBagLayout gbl) {
  GridBagConstraints gbc = new GridBagConstraints();
  for(int i=1;i<4;i++) {
    for(int j=1;j<5;j++) {
      JPanel panel = new JPanel();
      panel.setOpaque(false);
      panel.setPreferredSize(new Dimension(63,70));
      panel.setBorder(new LineBorder(Color.black,1));
      gbc.gridx = i-1;
      gbc.gridy = j-1;
      gbl.setConstraints(panel, gbc);
      add(panel);
    }
  }
}
/**
 * Ban を受け取ってその局面通りに表示させる。
 * 新しいパネルを一つずつ作って行く
 * @param ban
 */
 public void makeMasuFromBan(Ban ban,ActionListener actionListener) {
  removeAll();//前の情報は消す。
  setBanPanel();
  GridBagLayout gbl = gridBagLayout;
  GridBagConstraints gbc = new GridBagConstraints();
  for(int i=1;i<4;i++) {
    for(int j=1;j<5;j++) {
      Koma koma = ban.getBanarray(i, j);
      if(koma != null) {
        JPanel panel = new JPanel();
```

```
          panel.setOpaque(false);
          panel.setPreferredSize(new Dimension(63,70));
          panel.setBorder(new LineBorder(Color.black,1));
          gbc.gridx = 3-i;//左右反転
          gbc.gridy = j-1;
          gbl.setConstraints(panel, gbc);
          ImageIcon icon = MakeKomaImage.makeKomaImage(koma.getKomaNumber(), koma.getTeban(), him);
          Image image = icon.getImage().getScaledInstance(54, 60, Image.SCALE_DEFAULT);
          icon = new ImageIcon(image);
          JButton button = new JButton(icon);
          button.setContentAreaFilled(false);//背景透明化
          button.setBorderPainted(false);
          button.addActionListener(actionListener);
          button.setActionCommand(""+i+""+j);//盤の番号。76 とか
          panel.add(button,BorderLayout.CENTER);
          panel.setBackground(Color.lightGray);
          add(panel);
          panelArray[i-1][j-1] = panel;
        }else {
          //駒のないところにもボタンはつくる
          JPanel panel = new JPanel();
          panel.setOpaque(false);
          panel.setPreferredSize(new Dimension(63,70));
          panel.setBorder(new LineBorder(Color.black,1));
          gbc.gridx = 3-i;//左右反転
          gbc.gridy = j-1;
          gbl.setConstraints(panel, gbc);
          JButton button = new JButton();
          button.setPreferredSize(new Dimension(63,70));
          button.setBackground(Color.black);///
          button.addActionListener(actionListener);
          button.setActionCommand(""+i+""+j);//盤の番号。
          panel.add(button,BorderLayout.CENTER);
          panel.setBackground(Color.lightGray);
          add(panel);
          panelArray[i-1][j-1] = panel;
        }
      }
    }
  }
}
```

- FrameS クラス

```
package swing;

import java.awt.Color;
```

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.SpringLayout;
import javax.swing.border.LineBorder;

import board.Kyokumen;

public class FrameS extends JFrame{
  private MenuPanel panelMenu;
  private BanPanel panelBan;
  private KomadaiPanel panelSente;
  private KomadaiPanel panelGote;
  private JLabel labelTeban;
  private ActionListener actionListener;
  private ActionListenerMenu actionListenerMenu;
  private ActionKihuListener actionKihuListener;
  private KihuPanel panelKihu;
  public void setActionListener(ActionListener actionListener,
    ActionListenerMenu actionListenerMenu,ActionKihuListener actionKihuListener) {
    this.actionListener = actionListener;
    this.actionListenerMenu = actionListenerMenu;
    this.actionKihuListener = actionKihuListener;

  }
  /**
   * 次の局面に行く時にこのメソッドを呼んでおく
   * 次の局面になってから呼ぶ。盤上の駒は変わっている
   */
  public void setNextTe(String teban) {
    changeLabelTeban(teban);
  }
  public void changeLabelTeban(String string) {
    labelTeban.setText(string);
  }
  /**
   * cp 同士の対局の時に
   * 次の手に行くかどうかを確認する。
   * @return true:次に行く
   */
  public boolean checkNextOrExit() {
    int option = JOptionPane.showConfirmDialog
      (this, "次へ進みますか？", "確認", JOptionPane.YES_NO_OPTION,JOptionPane.PLAIN_MESSAGE);
      if (option == JOptionPane.YES_OPTION){
          return true;
      }else if (option == JOptionPane.NO_OPTION){
          return false;
      }
```

```
    return false;
}
public void outputMessage(String mes,String title) {
  JOptionPane.showMessageDialog(this, mes,title,JOptionPane.PLAIN_MESSAGE);
}
public void outputTouryou(String teban) {
  JOptionPane.showMessageDialog(this, teban+"の負けです。", "投了",JOptionPane.PLAIN_MESSAGE);
}
/**
 * panel の指定したところだけ変える
 * place は 23 なら 23
 * @param placeA
 * @param placeB
 */
public void changePanel(int placeA,int placeB,Kyokumen kyokumen) {
  if(0<placeA && placeA<10) {
    //盤上の駒
    panelBan.changePanel(placeA, placeB, kyokumen, actionListener);
  }else if(placeA==10) {
    //先手の駒台
    panelSente.changePanel(placeB, kyokumen.getSenteKomadai(), actionListener);
  }else if(placeA==20) {
    panelGote.changePanel(placeB, kyokumen.getGoteKomadai(), actionListener);
  }
  this.setVisible(true);
}
public void changeButtonColor(int place) {
  //place は 23 なら 23
  int a = place / 10;
  int b = place % 10;
  if(0<a && a<10 && 0<b && b<10) {
    //盤上
    panelBan.changeColor(a, b);
  }
  if(a==10) {
    panelSente.changeColor(b);
  }else if(a==20) {
    panelGote.changeColor(b);
  }
}
/**
 * 対局開始の処理。menuPanel の表示非表示を変える
 */
public void startGame() {
  panelMenu.setGaming();
  panelKihu.setNotVisible();
}
public void endGame() {
  panelMenu.setNotGaming();
```

```
}
 public void setFreeBefore() {
   panelMenu.setFreeBefore();
}
 public void setReadingKihu() {
   panelMenu.setNotGaming();
   panelKihu.setVisible();
}
 public void swing2() {
  //this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
   this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   this.setTitle("どうぶつしょうぎ");
   this.setLayout(new SpringLayout());
  //this.setLayout(null);
   JPanel panel = new JPanel();
   panel.setBackground(Color.lightGray);
   panel.setLayout(null);
   this.setContentPane(panel);
   this.setBounds(0,0,905,550);
   makePanel(panel);
   this.labelTeban = new JLabel("手番:");
   labelTeban.setBounds(10, 200, 150, 50);//x,y,width,heigh
   labelTeban.setBorder(new LineBorder(Color.black,2));
   panel.add(labelTeban);
   this.setVisible(true);
}
 public void makePanel(JPanel panel) {
   panelBan = new BanPanel(this);
   panelBan.setBanPanel();
   panel.add(panelBan);
   panelSente = new KomadaiPanel(this);
   panelSente.setKomadaiPanel(645, 390);
   panel.add(panelSente);
   panelGote = new KomadaiPanel(this);
   panelGote.setKomadaiPanel(10, 50);
   panel.add(panelGote);
   panelMenu = new MenuPanel();
   panelMenu.setMenuPanel(actionListenerMenu);
   panel.add(panelMenu);
   panelKihu = new KihuPanel();
   panelKihu.setKihuPanel(actionKihuListener);
   panel.add(panelKihu);

  //初期局面表示する。
   Kyokumen kyokumen = new Kyokumen();
   panelBan.makeMasuFromBan(kyokumen.getBan(),actionListener);
   panelSente.makeKomadaiPanel(kyokumen.getSenteKomadai(),actionListener);
   panelGote.makeKomadaiPanel(kyokumen.getGoteKomadai(),actionListener);
   panelMenu.setNotGaming();
```

```
  }
  /**
   * 入力の局面を表示する
   * @param kyokumen
   */
  public void setKyokumen(Kyokumen kyokumen) {
    panelBan.makeMasuFromBan(kyokumen.getBan(),actionListener);
    panelSente.makeKomadaiPanel(kyokumen.getSenteKomadai(),actionListener);
    panelGote.makeKomadaiPanel(kyokumen.getGoteKomadai(),actionListener);
    changeLabelTeban("手番: "+kyokumen.getTebanString());
    this.setVisible(true);
  }
  public void setKihuPanelVisible() {
    panelKihu.setVisible();
  }
  public void setKihuPanelNotVisible() {
    panelKihu.setNotVisible();
  }
  public ActionListener getActionListener() {
    return actionListener;
  }
}
```

- ImageIcon2 クラス

```
package swing;

import java.awt.Image;
import java.awt.Toolkit;
import java.awt.Window;
import java.awt.image.ImageProducer;
import java.net.URL;

import javax.swing.ImageIcon;

public class ImageIcon2 extends ImageIcon{
    public ImageIcon2(String f, Window own){
        super();
        Image image;
        try {
          Toolkit tk = own.getToolkit();
          URL url = own.getClass().getResource(f);
          image = tk.createImage((ImageProducer)url.getContent());
          setImage(image);
        } catch (Exception e) {
          System.out.println("Image not Found!");
        }
      }
```

```
}
```

- KihuPanel クラス

```
package swing;

import java.awt.Color;

import javax.swing.JButton;
import javax.swing.JPanel;

public class KihuPanel  extends JPanel{
  private JButton buttonNext;
  private JButton buttonBack;
  private JButton buttonFirst;
  private JButton buttonEnd;
  public void setKihuPanel(ActionKihuListener actionKihuListener) {
    setBackground(Color.darkGray);
    setBounds(10,300,145,40);//横、縦、幅、高さ
    setLayout(null);
    buttonFirst = new JButton("<<");
    buttonFirst.addActionListener(actionKihuListener);
    buttonFirst.setActionCommand("最初");//
    buttonFirst.setBounds(5, 5, 30, 30);
    add(buttonFirst);
    buttonBack = new JButton("<");
    buttonBack.addActionListener(actionKihuListener);
    buttonBack.setActionCommand("前へ");//戻る
    buttonBack.setBounds(40, 5, 30, 30);
    add(buttonBack);
    buttonNext = new JButton(">");
    buttonNext.addActionListener(actionKihuListener);
    buttonNext.setActionCommand("次へ");//
    buttonNext.setBounds(75, 5, 30, 30);
    add(buttonNext);
    buttonEnd = new JButton(">>");
    buttonEnd.addActionListener(actionKihuListener);
    buttonEnd.setActionCommand("最後");//
    buttonEnd.setBounds(110, 5, 30, 30);
    add(buttonEnd);
    this.setVisible(false);
  }
  public void setVisible() {
    this.setVisible(true);
  }
  public void setNotVisible() {
    this.setVisible(false);
  }
```

```
}
```

- KomadaiPanel クラス

```
package swing;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Image;
import java.awt.Window;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.LineBorder;

import board.Komadai;

public class KomadaiPanel extends JPanel{
  private GridBagLayout gridBagLayout;
  private JPanel[] panelArray;//0:歩
  private Window him;

  public KomadaiPanel(Window her) {
    him = her;
  }

  public void changeColor(int b) {
    JPanel panel = panelArray[b-1];
    if(panel != null) {
      panel.setOpaque(true);
    }
  }
  /**
   * 駒台の一箇所の画像を変える
   * @param b
   * @param komadai
   * @param actionListener
   */
  public void changePanel(int b,Komadai komadai,ActionListener actionListener) {
    GridBagLayout gbl = gridBagLayout;
    GridBagConstraints gbc = new GridBagConstraints();
    int teban = komadai.getTeban();
```

```
  int komaMany = komadai.getKomaNumber(b);
  JPanel oldPanel = panelArray[b-1];
  if(oldPanel != null) {
    oldPanel.setVisible(false);
  }
  JPanel panel = new JPanel();
  panel.setOpaque(false);
  panel.setPreferredSize(new Dimension(45,50));
  panel.setBorder(new LineBorder(Color.black,1));
  panel.setLayout(new BorderLayout());
  int x = (b-1)%4;
  int y = (b-1)/4;
  if(teban==1) {
    gbc.gridx = x;
    gbc.gridy = y;
  }else {
    gbc.gridx = 3-x;
    gbc.gridy = y;
  }
  gbl.setConstraints(panel, gbc);
  if(komaMany>0) {
    JLabel label = new JLabel(""+komaMany);
    ///駒の画像を表示させる。
    ImageIcon icon = MakeKomaImage.makeKomaImage(b, teban, him);
    Image image = icon.getImage().getScaledInstance(36, 40, Image.SCALE_DEFAULT);
    icon = new ImageIcon(image);
    JButton button = new JButton(icon);
    button.setContentAreaFilled(false);//背景透明化
    button.setBorderPainted(false);
    //actionlistener 加える
    button.addActionListener(actionListener);
    button.setActionCommand(""+teban+"0"+b);//駒台の駒の番号。盤の番号と同じ。先手の歩なら 11
    panel.add(button,BorderLayout.CENTER);
    panel.add(label,BorderLayout.NORTH);
    add(panel);
    panelArray[b-1] = panel;
  }else {
    //駒がない時
    JButton button = new JButton("");
    //actionlistener 加える
    button.addActionListener(actionListener);
    button.setActionCommand(""+teban+"0"+b);//駒台の駒の番号。盤の番号と同じ。先手の歩なら 11
    panel.add(button,BorderLayout.CENTER);
    add(panel);
    panelArray[b-1] = panel;
  }
}
 public void setKomadaiPanel(int pW,int pH) {
   setBackground(new Color(153,102,102));
```

```
  setBounds(pW,pHi,200,120);//横、縦、幅、高さ
  GridBagLayout gbl = new GridBagLayout();
  gridBagLayout = gbl;
  setLayout(gbl);
  makeKomadai(gbl);
  panelArray = new JPanel[4];
  for(int i=0;i<4;i++) {
    panelArray[i] = null;
  }
}
 private void makeKomadai(GridBagLayout gbl) {
  GridBagConstraints gbc = new GridBagConstraints();
  for(int i=1;i<5;i++) {
    for(int j=1;j<2;j++) {
      JPanel panel = new JPanel();
      panel.setOpaque(false);
      panel.setPreferredSize(new Dimension(45,50));
      panel.setBorder(new LineBorder(Color.black,1));
      gbc.gridx = i-1;
      gbc.gridy = j-1;
      gbl.setConstraints(panel, gbc);
      add(panel);
    }
  }
}
 public void makeKomadaiPanel(Komadai komadai,ActionListener actionListener) {
  int side = this.getX();
  int length = this.getY();
  removeAll();
  setKomadaiPanel(side, length);
  GridBagLayout gbl = gridBagLayout;
  GridBagConstraints gbc = new GridBagConstraints();
  int teban = komadai.getTeban();
  int komaNumber = 1;
  if(teban==2) {
    komaNumber = 4;
  }
  for(int j=1;j<2;j++) {
    for(int i=1;i<5;i++) {
      int komaMany = komadai.getKomaNumber(komaNumber);
      if(komaMany>0) {
        JPanel panel = new JPanel();
        panel.setOpaque(false);
        panel.setPreferredSize(new Dimension(45,50));
        panel.setBorder(new LineBorder(Color.black,1));
        panel.setLayout(new BorderLayout());
        gbc.gridx = i-1;
        gbc.gridy = j-1;
        gbl.setConstraints(panel, gbc);
```

```
          JLabel label = new JLabel(""+komaMany);
          ///駒の画像を表示させる。
          ImageIcon icon = MakeKomaImage.makeKomaImage(komaNumber, teban, him);
          System.out.println(""+komaNumber+","+i+","+j);
          Image image = icon.getImage().getScaledInstance(36, 40, Image.SCALE_DEFAULT);
          icon = new ImageIcon(image);
          JButton button = new JButton(icon);
          button.setContentAreaFilled(false);//背景透明化
          button.setBorderPainted(false);
          //actionlistener 加える
          button.addActionListener(actionListener);
          button.setActionCommand(""+teban+"0"+komaNumber);//駒台の駒の番号。盤の番号と同じ。先手の歩
なら 11
          panel.add(button,BorderLayout.CENTER);
          panel.add(label,BorderLayout.NORTH);
          add(panel);
          panelArray[komaNumber-1] = panel;
        }else {
          //駒台に駒がなくてもボタンは作る
          JPanel panel = new JPanel();
          panel.setOpaque(false);
          panel.setPreferredSize(new Dimension(45,50));
          panel.setBorder(new LineBorder(Color.black,1));
          panel.setLayout(new BorderLayout());
          gbc.gridx = i-1;
          gbc.gridy = j-1;
          gbl.setConstraints(panel, gbc);
          ///駒の画像を表示させる。駒はないのでボタンだけ表示
          JButton button = new JButton("");
          //actionlistener 加える
          button.addActionListener(actionListener);
          button.setActionCommand(""+teban+"0"+komaNumber);//駒台の駒の番号。盤の番号と同じ。先手の歩
なら 11
          panel.add(button,BorderLayout.CENTER);
          add(panel);
          panelArray[komaNumber-1] = panel;
        }
        if(teban==1) {
          komaNumber++;
        }else {
          komaNumber--;
        }
      }
    }
  }
}
```

- MakeKomaImage クラス

```
package swing;

import java.awt.Window;

import javax.swing.ImageIcon;

public class MakeKomaImage {
  public static ImageIcon makeKomaImage(int komaNumber,int teban, Window him) {
    ImageIcon imageIcon = null;
    ClassLoader classloader = him.getClass().getClassLoader();
    switch(komaNumber) {
    case 1:
      if(teban==1) {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/hiyoko.png"));
     }else {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/hiyokoT.png"));
      }
      break;
    case 2:
      if(teban==1) {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/zou.png"));
     }else {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/zouT.png"));
      }
      break;
    case 3:
      if(teban==1) {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/kirin.png"));
     }else {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/kirinT.png"));
      }
      break;
    case 4:
      if(teban==1) {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/lion.png"));
     }else {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/lionT.png"));
      }
      break;
    case 11:
      if(teban==1) {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/niwatori.png"));
     }else {
        imageIcon = new ImageIcon(classloader.getResource("komaImage/niwatoriT.png"));
      }
      break;
   }
    return imageIcon;
  }
```

```
}
```

- MenuPanel クラス

```
package swing;

import java.awt.Color;

import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JPanel;

public class MenuPanel extends JPanel{
  private JButton buttonNew;//新規対局ボタン
  private JButton buttonStart;//現在の局面から対局開始
  private JButton buttonTouryou;
  private JButton buttonReadKyokumen;
  private JButton buttonReadKihu;
  private JButton buttonCalc;
  private JButton buttonTumi;
  private JButton buttonSaveKyokumen;
  private JButton buttonSaveKihu;
  private JButton buttonMatta;
  private JButton buttonFree;
  /**
   * 対局中の状態にする。
   */
  public void setGaming() {
    buttonNew.setVisible(false);
    buttonStart.setVisible(false);
    buttonTouryou.setVisible(true);
    buttonReadKyokumen.setVisible(false);
    buttonReadKihu.setVisible(false);
    buttonCalc.setVisible(true);//人間同士の時は隠したほうがいい?
    buttonTumi.setVisible(true);
    buttonSaveKyokumen.setVisible(true);
    buttonSaveKihu.setVisible(true);
    buttonMatta.setVisible(true);
    buttonFree.setVisible(false);
  }
  public void setNotGaming() {
    buttonNew.setVisible(true);
    buttonStart.setVisible(true);
    buttonTouryou.setVisible(false);
    buttonReadKyokumen.setVisible(true);
    buttonReadKihu.setVisible(true);
    buttonCalc.setVisible(true);
    buttonTumi.setVisible(true);
```

```
  buttonSaveKyokumen.setVisible(true);
  buttonSaveKihu.setVisible(true);
  buttonMatta.setVisible(false);
  buttonFree.setVisible(true);
}
 public void setFreeBefore() {
  buttonNew.setVisible(false);
  buttonStart.setVisible(false);
  buttonTouryou.setVisible(false);
  buttonReadKyokumen.setVisible(false);
  buttonReadKihu.setVisible(false);
  buttonCalc.setVisible(false);
  buttonTumi.setVisible(false);
  buttonSaveKyokumen.setVisible(true);
  buttonSaveKihu.setVisible(false);
  buttonMatta.setVisible(false);
  buttonFree.setVisible(true);
  ///buttonFree.setText("終了");
}

 public void setMenuPanel(ActionListenerMenu actionListenerMenu) {
  setBackground(Color.darkGray);
  setBounds(50,10,830,30);//横、縦、幅、高さ
  setLayout(new BoxLayout(this,BoxLayout.X_AXIS));
  buttonNew = new JButton("新規");//新規対局ボタン
  buttonNew.addActionListener(actionListenerMenu);
  buttonNew.setActionCommand("新規対局");//
  add(buttonNew);
  buttonStart = new JButton("現始");//現在の局面から対局ボタン
  buttonStart.addActionListener(actionListenerMenu);
  buttonStart.setActionCommand("現在の局面から対局");
  add(buttonStart);
  buttonTouryou = new JButton("投了");//投了ボタン
  buttonTouryou.addActionListener(actionListenerMenu);
  buttonTouryou.setActionCommand("投了");
  add(buttonTouryou);
  buttonReadKyokumen = new JButton("局読");//局面読み込みボタン
  buttonReadKyokumen.addActionListener(actionListenerMenu);
  buttonReadKyokumen.setActionCommand("局面読み込み");
  add(buttonReadKyokumen);
  buttonReadKihu = new JButton("棋読");//棋譜読み込みボタン
  buttonReadKihu.addActionListener(actionListenerMenu);
  buttonReadKihu.setActionCommand("棋譜読み込み");
  add(buttonReadKihu);
  buttonCalc = new JButton("評価");//評価ボタン
  buttonCalc.addActionListener(actionListenerMenu);
  buttonCalc.setActionCommand("局面評価");
  add(buttonCalc);
  buttonTumi = new JButton("詰み");//詰みチェックボタン
```

```
    buttonTumi.addActionListener(actionListenerMenu);
    buttonTumi.setActionCommand("詰みチェック");
    add(buttonTumi);
    buttonSaveKyokumen = new JButton("局保");//局面保存チェックボタン
    buttonSaveKyokumen.addActionListener(actionListenerMenu);
    buttonSaveKyokumen.setActionCommand("局面保存");
    add(buttonSaveKyokumen);
    buttonSaveKihu = new JButton("棋保");//棋譜保存チェックボタン
    buttonSaveKihu.addActionListener(actionListenerMenu);
    buttonSaveKihu.setActionCommand("棋譜保存");
    add(buttonSaveKihu);
    buttonMatta = new JButton("待た");//待ったチェックボタン
    buttonMatta.addActionListener(actionListenerMenu);
    buttonMatta.setActionCommand("待った");
    add(buttonMatta);
    buttonFree = new JButton("自由");//自由に駒を動かせるチェックボタン
    buttonFree.addActionListener(actionListenerMenu);
    buttonFree.setActionCommand("自由");
    add(buttonFree);

  }


}
```

- StateGame クラス

```
package swing;

import board.Board;
import board.Move;
import main.Kihu;

public class StateGame {
  //1:何もしていない,2:対局中手番は cp,3:対局中手番は人間で何も選択していない,4:対局中手番は人間で何かの駒
を選択している。5:棋譜読み込んでいる。6:フリー自由に動かせる. 選択していない.7:選択されている
  private int state;
  private int beforePlace;
  private Kihu kihu;//棋譜を読み込んだ時に棋譜が保存される。
  private int countTesuu;//棋譜を読み込んだ時になんて目のところにいるかを保存する。最初の局面にいる時は
0.
   private Board board;
  private FrameS frame;
  public StateGame(Board board,FrameS frame) {
    this.state = 1;
    this.board = board;
    this.frame = frame;
    kihu = null;
```

```
  countTesuu = 0;
}
public void setNotGaming() {
  beforePlace = 0;
  state = 1;
}
public void setBeforePlace(int beforePlace) {
 //beforePlace を設定して手番が人間で駒を選択した状態にする。
  this.beforePlace = beforePlace;
  this.state = 4;
}
/**
 * 対局中で人間が手番の時 true
 * @return
 */
public boolean checkGamingPlayer() {
  if(state==3 || state == 4) {
    return true;
 }
  return false;
}
public boolean checkChoseAfterPlace() {
  if(this.state == 4) {
    return true;
 }
  return false;
}
/**
 * 対局中かどうかを判断する。
 * 対局をしていないなら true
 * @return
 */
public boolean checkNotGaming() {
  if(state==1 || state==5) {
    return true;
 }
  return false;
}
public boolean checkNothin() {
  if(state==1) {
    return true;
 }
  return false;
}
public boolean checkChoseBeforeKoma() {
  if(state==3) {
    return true;
 }
  return false;
```

```
}
 public boolean checkReadingKihu() {
   if(state==5) {
     return true;
  }
   return false;
}
 public boolean checkFreeBefore() {
   if(state==6) {
     return true;
  }
   return false;
}
 public boolean checkFreeAfter() {
   if(state==7) {
     return true;
  }
   return false;
}
 public int getBeforePlace() {
  /*
   if(state != 4) {
     return -1;
  }
  */
   return beforePlace;
}
 public int getState() {
   return state;
}
 public void setGaming(boolean player) {
  //対局開始の状態にする。開始でない時も呼ばれる
   if(player) {
     this.state = 2;//cp
    //次の一手を計算する。次の局面に進める。
     String teban = board.getTebanS();
     if(board.goNextBoardFromSwing()==0) {
      //投了
       frame.outputTouryou(teban);
       board.getKihu().makeLastKyokumen();
       setReadingKihu(board.getKihu().getKihuList().size());
       frame.setReadingKihu();
       return;
    }
     Move move = board.getBeforeMove();
     frame.changePanel(move.getAfterPlaceA(), move.getAfterPlaceB(), board.getKyokumen());
     frame.changePanel(move.getBeforePlaceA(), move.getBeforePlaceB(), board.getKyokumen());
     move.outputMoveKihu();
     if(move.getGetKoma() != 0) {
```

```
      //駒を取っていた時
      frame.changePanel((3-board.getTeban())*10, move.getGetKoma()%10, board.getKyokumen());
    }
     int sennitite = board.checkSennitite();
     if(sennitite==1) {
      //千日手
      frame.outputMessage("千日手です。", "千日手");
      setNotGaming();
      frame.endGame();
    }else if(sennitite==2) {
      //連続ライオン手の千日手。手番がライオン手している。
      frame.outputMessage(board.getTebanS()+"の反則負けです。", "反則:連続ライオン手");
      setNotGaming();
      frame.endGame();
    }else if(sennitite==3) {
      //連続ライオン手の千日手。手番がライオン手されている。
      frame.outputMessage(board.getTebanOppositeS()+"の反則負けです。", "反則:連続ライオン手");
      setNotGaming();
      frame.endGame();
    }
    //次も cp の時は待つために次へボタン表示
     if(board.getTebanPlayer()) {
      //次へ行くか確認してやめることもできるようにする。
      if(frame.checkNextOrExit()) {
        //次へ行く
        setGaming(board.getTebanPlayer());
      }else {
        //そこで対局を終了する。
        frame.outputMessage("対局を終了します。", "対局終了");
        board.getKihu().makeLastKyokumen();
        setReadingKihu(board.getKihu().getKihuList().size());
        frame.setReadingKihu();
      }
    }else {
      //次が人の手番
      setGaming(board.getTebanPlayer());
    }

  }else {
     this.state = 3;//人間
    //入力を待つ
  }
   frame.setNextTe("手番: "+board.getTebanS());
}
 public void setFreeBefore() {
   state = 6;
   beforePlace = 0;
}
 public void setFreeAfter() {
```

```
  state = 7;
}
public void setReadingKihu(int tesuu) {
  state = 5;
  beforePlace = 0;
  this.kihu = board.getKihu();
  countTesuu = tesuu;
}
public void setTesuuAdd() {
  countTesuu++;
  if(countTesuu>kihu.getKihuList().size()) {
    countTesuu = kihu.getKihuList().size();
    System.out.println("error:count tesuu in state game class add ");
  }
}
public void setTesuuDecrease() {
  countTesuu--;
  if(countTesuu<0) {
    countTesuu = 0;
    System.out.println("error:count tesuu in state game class decrease");
  }
}
public void setTesuuFirst() {
  countTesuu = 0;
}
public void setTesuuEnd() {
  countTesuu = kihu.getKihuList().size();
}
public Kihu getKihu() {
  return kihu;
}
public int getCountTesuu() {
  return countTesuu;
}
public Move getMoveNext() {
  if(countTesuu >= kihu.getKihuList().size()) {
    return null;
  }
  return kihu.getKihuList().get(countTesuu);
}
public Move getMoveBack() {
  if(countTesuu <= 0) {
    return null;
  }
  return kihu.getKihuList().get(countTesuu-1);
}
public Board getBoard() {
  return board;
}
```

```
}
```