Code Review

Section to Review

The focus for this review is on the code that is responsible for enacting the penalty on the white player in the Save The Network game should the white player ignore the opportunity to make a jump move to capture a black piece.

Penalties should occur in the following cases:

- If only one white piece has the opportunity to jump, but does not move (i.e. the other white piece moves instead), then that white piece is penalized and removed
- If only one white piece has the opportunity to jump, but makes a slide move instead of a jump move, then that whie piece is penalized and removed

Penalties should not occur in the following cases:

- If both white pieces have the opportunity to jump, provided one of them will make a jump move, there is no penalization
- If only white piece has the opportunity to jump, and it jumps, it will not be penalized

If neither white piece has the opportunity to jump, then there is no penalization.

Synopsis

From examining the code, it appears that all of the logic for this is handled upon the culmination of the turn, i.e. in the *reverseTurn()* method in the *SaveTheNetworkModel* class.

```
public void reverseTurn() {
      if(turn==Peg.WHITE)
      {
            selectPeg(PEG_ID_NONE);
            checkPenalty();
            lastWhiteMove = Move.NONE;
            if(this.isBlackLose()){
                  setStatus(Status.WINNER_WHITE);
                  selectPeg(PEG_ID_NONE);
                  return;
            }
            if (getStatus() != Status.PENALTY_REQUIRED) {
                  setStatus(Status.BLACK_MOVE);
                  turn=Peg.BLACK;
            }
      }
      else
      {
            turn=Peg.WHITE;
            lastWhiteMove = Move.NONE;
            setStatus(Status.WHITE_MOVE);
            preWhiteMove();
      }
      selectPeg(PEG_ID_NONE);
}
```

Here, it checks to see if a penalty action is required by examining the current position of the white pegs, and if a penalty action is required, it sets the necessary flags with *checkPenalty()*.

```
private void checkPenalty() {
      if (lastWhiteMove == Move.JUMP) return;
      for (int i = 0; i < 2; i++) {
            if (white[i][0] != white[i][1]) tempFlipWhites(i);
            if (isFutureJumpPossible(white[i][0])) {
                  if (white[i][0] != white[i][1]) tempFlipWhites(i);
                  whitePenalty = i+1;
                  setStatus(Status.PENALTY_REQUIRED);
                  return;
            }
            if (white[i][0] != white[i][1]) tempFlipWhites(i);
      }
      whitePenalty = 0;
}
```

The view (*SaveTheNetworkBoardPanel*) will only enable the Continue button after it checks if there has been a penalty. Otherwise a penalty has occured, gameplay is paused, with the appropriate peg locations marked, until the Continue button is clicked, which invokes *endWhiteJump()*.

```
public boolean endWhiteJump(){
      SaveTheNetworkModel m=(SaveTheNetworkModel)getModel();
      if(m.getStatus()==Status.PENALTY_REQUIRED){
            m.doPenalty();
            updateGUI();
            return true;
      }
      else{
            boolean result = true;
            m.reverseTurn();
            if(m.getStatus()==Status.PENALTY_REQUIRED){
                  result = false;
            }
            updateGUI();
            return result;
      }
}
```

Here, control passes back to the model in order to actually perform the penalty action of removing the offending white piece in *doPenalty()*. The white peg trackers are reset, and the game can then continue.

```
public void doPenalty() {
      if (whitePenalty > 0) setPeg(Peg.NONE, white[whitePenalty-1][1]);
      if(!whiteExists()) setStatus(Status.WINNER_BLACK);
      else {
            setStatus(Status.BLACK_MOVE);
            turn=Peg.BLACK;
      }
      resetWhiteTrackers();
}
```

If there is no penalty, the game continues as normal via a call to *updateGUI()*, which has the controller (*SaveTheNetworkBoardPanel*) pass through all interactions via *processClick()* to *togglePeg()* in the model *(SaveTheNetworkModel*).

Structure, Code, and Comments
There is a little confusion on the structural flow of how the penalty is exacted, as code control jumps about the place in order to determine the case for a penalty. The use of polymorphism to break apart the player move processing between base *Model* class and *SaveTheNetworkModel* class also impedes readability. There are very few comments in the classes responsible for both the model and the view/controller. This, coupled with the presence of ambiguous method names, makes the code rather unreadable, especially when trying to puzzle out the exact logic of what occurs in the program. For example, *endWhiteJump()* is responsible for handling both the ending of White's turn, in addition to processing the continuation of the game after handling the penalty case.

However, at the same time, there is no logic handling in the in the view nor in the controller part of the application, therefore any other GUI can simply "plug-and-play" into interacting with the model. As one of the aims of the developers working on this project is to follow the Model-View-Controller paradigm, this aspect of the application is very well done.