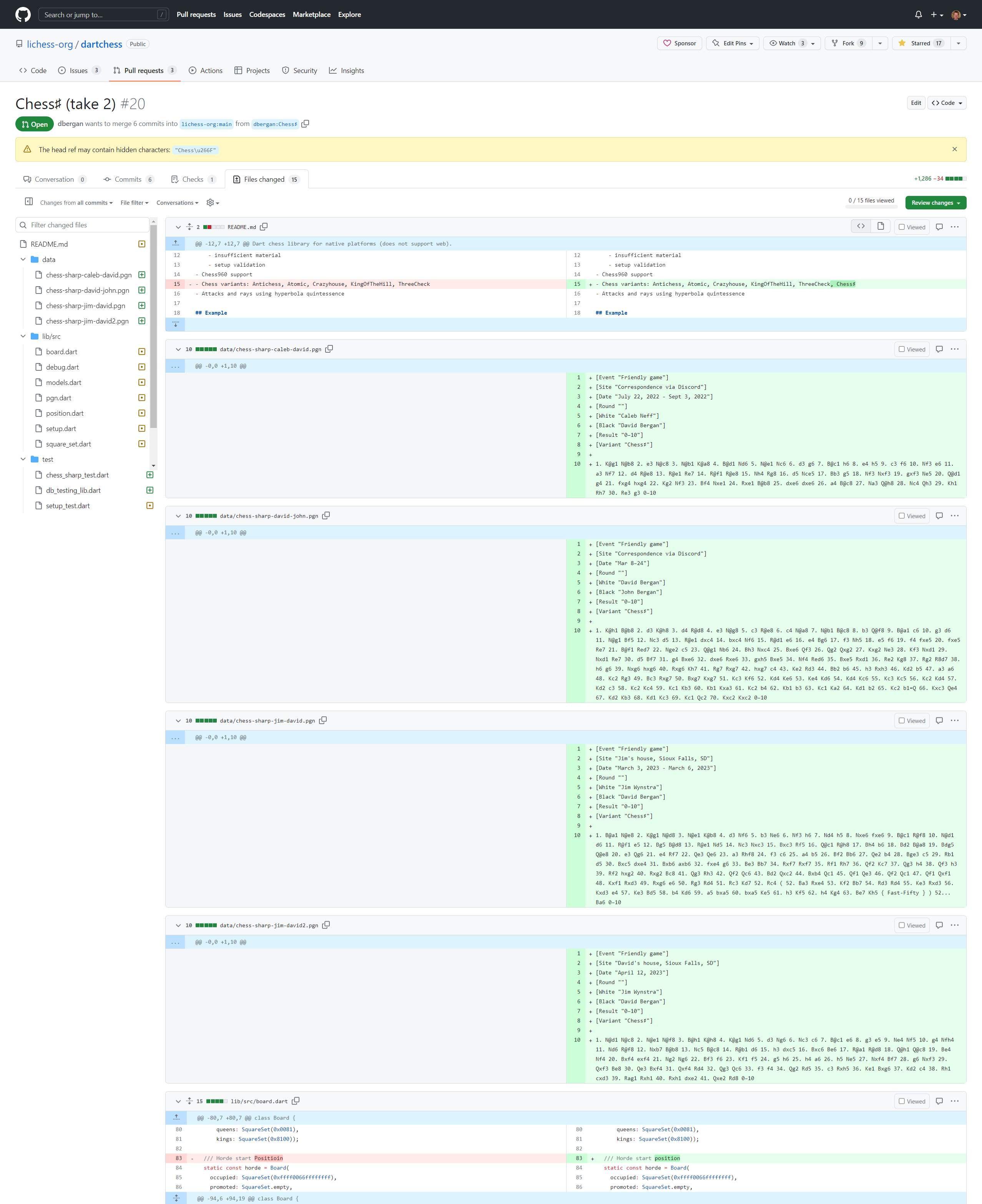


Chess♯ (take 2) #20

Open **dbergan** wants to merge 6 commits into `lichess-org:main` from `dbergan:Chess♯`

⚠ The head ref may contain hidden characters: `"Chess\u266F"`

2 README.md

```
@@ -12,7 +12,7 @@ Dart chess library for native platforms (does not support web).
    - insufficient material
    - setup validation
  - Chess960 support
- - Chess variants: Antichess, Atomic, Crazyhouse, KingOfTheHill, ThreeCheck
+ - Chess variants: Antichess, Atomic, Crazyhouse, KingOfTheHill, ThreeCheck, Chess♯
  - Attacks and rays using hyperbola quintessence

  ## Example
```

10 data/chess-sharp-caleb-david.pgn

```
@@ -0,0 +1,10 @@
+ [Event "Friendly game"]
+ [Site "Correspondence via Discord"]
+ [Date "July 22, 2022 - Sept 3, 2022"]
+ [Round ""]
+ [White "Caleb Neff"]
+ [Black "David Bergan"]
+ [Result "0-10"]
+ [Variant "Chess♯"]
+
+ 1. K@g1 N@b8 2. e3 N@c8 3. N@b1 K@a8 4. B@d1 Nd6 5. N@e1 Nc6 6. d3 g6 7. B@c1 h6 8. e4 h5 9. c3 f6 10. Nf3 e6 11. a3 Nf7 12. d4 R@e8 13. R@e1 Re7 14. R@f1 R@e8 15. Nh4 Rg8 16. d5 Nce5 17. Bb3 g5 18. Nf3 Nxf3 19. gxf3 Ne5 20. Q@d1 g4 21. fxg4 hxg4 22. Kg2 Nf3 23. Bf4 Nxe1 24. Rxe1 B@b8 25. dxe6 dxe6 26. a4 B@c8 27. Na3 Q@h8 28. Nc4 Qh3 29. Kh1 Rh7 30. Re3 g3 0-10
```

10 data/chess-sharp-david-john.pgn

```
@@ -0,0 +1,10 @@
+ [Event "Friendly game"]
+ [Site "Correspondence via Discord"]
+ [Date "Mar 8-24"]
+ [Round ""]
+ [White "David Bergan"]
+ [Black "John Bergan"]
+ [Result "0-10"]
+ [Variant "Chess♯"]
+
+ 1. K@h1 B@b8 2. d3 K@h8 3. d4 R@d8 4. e3 N@g8 5. c3 R@e8 6. c4 N@a8 7. N@b1 B@c8 8. b3 Q@f8 9. B@a1 c6 10. g3 d6 11. N@g1 Bf5 12. Nc3 d5 13. R@e1 dxc4 14. bxc4 Nf6 15. R@d1 e6 16. e4 Bg6 17. f3 Nh5 18. e5 f6 19. f4 fxe5 20. fxe5 Re7 21. B@f1 Red7 22. Nge2 c5 23. Q@g1 Nb6 24. Bh3 Nxc4 25. Bxe6 Qf3 26. Qg2 Qxg2 27. Kxg2 Ne3 28. Kf3 Nxd1 29. Nxd1 Re7 30. d5 Bf7 31. g4 Bxe6 32. dxe6 Rxe6 33. gxh5 Bxe5 34. Nf4 Red6 35. Bxe5 Rxd1 36. Re2 Kg8 37. Rg2 R8d7 38. h6 g6 39. Nxg6 hxg6 40. Rxg6 Kh7 41. Rg7 Rxg7 42. hxg7 c4 43. Ke2 Rd3 44. Bb2 b6 45. h3 Rxh3 46. Kd2 b5 47. a3 a6 48. Kc2 Rg3 49. Bc3 Rxg7 50. Bxg7 Kxg7 51. Kc3 Kf6 52. Kd4 Ke6 53. Ke4 Kd6 54. Kd4 Kc6 55. Kc3 Kc5 56. Kc2 Kd4 57. Kd2 c3 58. Kc2 Kc4 59. Kc1 Kb3 60. Kb1 Kxa3 61. Kc2 b4 62. Kb1 b3 63. Kc1 Ka2 64. Kd1 b2 65. Kc2 b1=Q 66. Kxc3 Qe4 67. Kd2 Kb3 68. Kd1 Kc3 69. Kc1 Qc2 70. Kxc2 Kxc2 0-10
```

10 data/chess-sharp-jim-david.pgn

```
@@ -0,0 +1,10 @@
+ [Event "Friendly game"]
+ [Site "Jim's house, Sioux Falls, SD"]
+ [Date "March 3, 2023 - March 6, 2023"]
+ [Round ""]
+ [White "Jim Wynstra"]
+ [Black "David Bergan"]
+ [Result "0-10"]
+ [Variant "Chess♯"]
+
+ 1. B@a1 N@e8 2. K@g1 N@d8 3. N@e1 K@b8 4. d3 Nf6 5. b3 Ne6 6. Nf3 h6 7. Nd4 h5 8. Nxe6 fxe6 9. B@c1 R@f8 10. N@d1 d6 11. R@f1 e5 12. Bg5 B@d8 13. R@e1 Nd5 14. Nc3 Nxc3 15. Bxc3 Rf5 16. Q@c1 R@h8 17. Bh4 b6 18. Bd2 B@a8 19. Bdg5 Q@e8 20. e3 Qg6 21. e4 Rf7 22. Qe3 Qe6 23. a3 Rhf8 24. f3 c6 25. a4 b5 26. Bf2 Bb6 27. Qe2 b4 28. Bge3 c5 29. Rb1 d5 30. Bxc5 dxe4 31. Bxb6 axb6 32. fxe4 g6 33. Be3 Bb7 34. Rxf7 Rxf7 35. Rf1 Rh7 36. Qf2 Kc7 37. Qg3 h4 38. Qf3 h3 39. Rf2 hxg2 40. Rxg2 Bc8 41. Qg3 Rh3 42. Qf2 Qc6 43. Bd2 Qxc2 44. Bxb4 Qc1 45. Qf1 Qe3 46. Qf2 Qc1 47. Qf1 Qxf1 48. Kxf1 Rxd3 49. Rxg6 e6 50. Rg3 Rd4 51. Rc3 Kd7 52. Rc4 ( 52. Ba3 Rxe4 53. Kf2 Bb7 54. Rd3 Rd4 55. Ke3 Rxd3 56. Kxd3 e4 57. Ke3 Bd5 58. b4 Kd6 59. a5 bxa5 60. bxa5 Ke5 61. h3 Kf5 62. h4 Kg4 63. Be7 Kh5 { Fast-Fifty } ) 52... Ba6 0-10
```

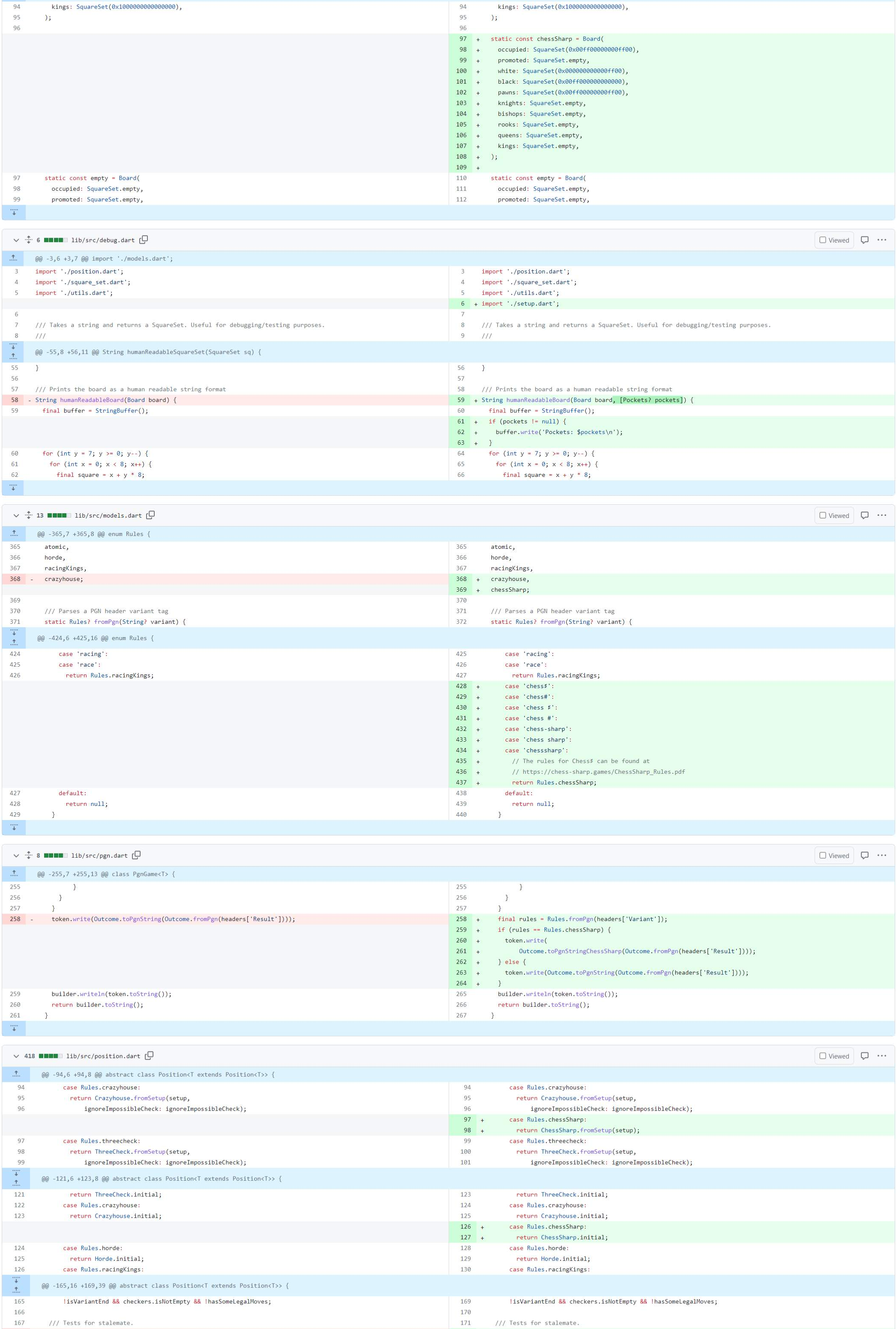
10 data/chess-sharp-jim-david2.pgn

```
@@ -0,0 +1,10 @@
+ [Event "Friendly game"]
+ [Site "David's house, Sioux Falls, SD"]
+ [Date "April 12, 2023"]
+ [Round ""]
+ [White "Jim Wynstra"]
+ [Black "David Bergan"]
+ [Result "0-10"]
+ [Variant "Chess♯"]
+
+ 1. N@d1 N@c8 2. N@e1 N@f8 3. B@h1 K@h8 4. K@g1 Nd6 5. d3 Ng6 6. Nc3 c6 7. B@c1 e6 8. g3 e5 9. Ne4 Nf5 10. g4 Nfh4 11. Nd6 R@f8 12. Nxb7 B@b8 13. Nc5 B@c8 14. R@b1 d6 15. h3 dxc5 16. Bxc6 Be6 17. R@a1 R@d8 18. Q@h1 Q@c8 19. Be4 Nf4 20. Bxf4 exf4 21. Ng2 Ng6 22. Bf3 f6 23. Kf1 f5 24. g5 h6 25. h4 a6 26. h5 Ne5 27. Nxf4 Bf7 28. g6 Nxf3 29. Qxf3 Be8 30. Qe3 Bxf4 31. Qxf4 Rd4 32. Qg3 Qc6 33. f3 f4 34. Qg2 Rd5 35. c3 Rxh5 36. Ke1 Bxg6 37. Kd2 c4 38. Rh1 cxd3 39. Rag1 Rxh1 40. Rxh1 dxe2 41. Qxe2 Rd8 0-10
```

15 lib/src/board.dart

```
@@ -80,7 +80,7 @@ class Board {
        queens: SquareSet(0x0081),
        kings: SquareSet(0x8100));

-   /// Horde start Positioin
+   /// Horde start position
    static const horde = Board(
      occupied: SquareSet(0xffff0066ffffffff),
      promoted: SquareSet.empty,
@@ -94,6 +94,19 @@ class Board {
```



```
     kings: SquareSet(0x1000000000000000),
   );

+  static const chessSharp = Board(
+    occupied: SquareSet(0x00ff00000000ff00),
+    promoted: SquareSet.empty,
+    white: SquareSet(0x000000000000ff00),
+    black: SquareSet(0x00ff000000000000),
+    pawns: SquareSet(0x00ff00000000ff00),
+    knights: SquareSet.empty,
+    bishops: SquareSet.empty,
+    rooks: SquareSet.empty,
+    queens: SquareSet.empty,
+    kings: SquareSet.empty,
+  );
+
   static const empty = Board(
     occupied: SquareSet.empty,
     promoted: SquareSet.empty,
```

6 lib/src/debug.dart ☐ Viewed

```
@@ -3,6 +3,7 @@ import './models.dart';
 import './position.dart';
 import './square_set.dart';
 import './utils.dart';
+import './setup.dart';

 /// Takes a string and returns a SquareSet. Useful for debugging/testing purposes.
 ///
@@ -55,8 +56,11 @@ String humanReadableSquareSet(SquareSet sq) {
 }

 /// Prints the board as a human readable string format
-String humanReadableBoard(Board board) {
+String humanReadableBoard(Board board, [Pockets? pockets]) {
   final buffer = StringBuffer();
+  if (pockets != null) {
+    buffer.write('Pockets: $pockets\n');
+  }
   for (int y = 7; y >= 0; y--) {
     for (int x = 0; x < 8; x++) {
       final square = x + y * 8;
```

13 lib/src/models.dart ☐ Viewed

```
@@ -365,7 +365,8 @@ enum Rules {
   atomic,
   horde,
   racingKings,
-  crazyhouse;
+  crazyhouse,
+  chessSharp;

   /// Parses a PGN header variant tag
   static Rules? fromPgn(String? variant) {
@@ -424,6 +425,16 @@ enum Rules {
       case 'racing':
       case 'race':
         return Rules.racingKings;
+      case 'chess♯':
+      case 'chess#':
+      case 'chess ♯':
+      case 'chess #':
+      case 'chess-sharp':
+      case 'chess sharp':
+      case 'chesssharp':
+        // The rules for Chess♯ can be found at
+        // https://chess-sharp.games/ChessSharp_Rules.pdf
+        return Rules.chessSharp;
       default:
         return null;
     }
```

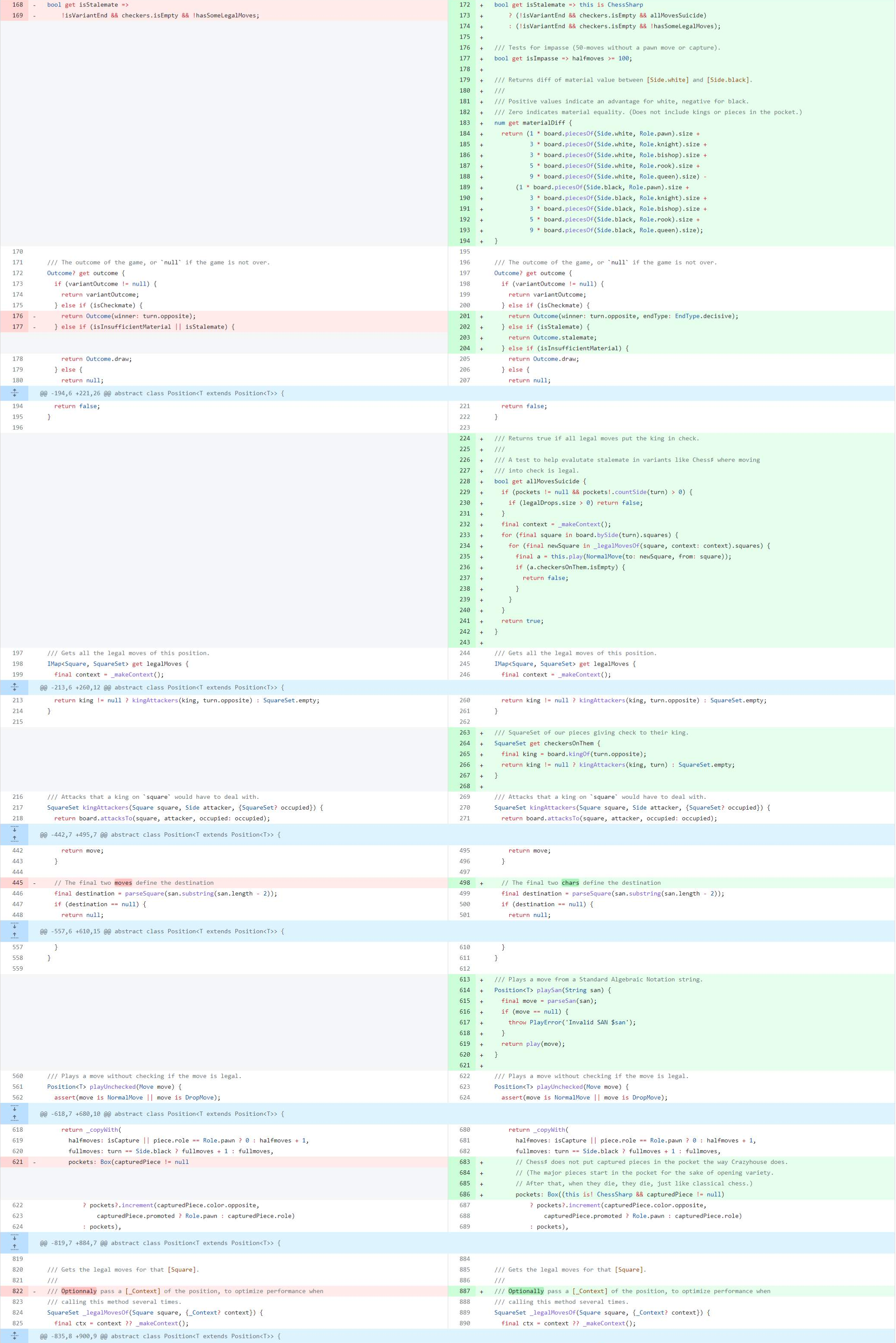
8 lib/src/pgn.dart ☐ Viewed

```
@@ -255,7 +255,13 @@ class PgnGame<T> {
         }
       }
     }
-    token.write(Outcome.toPgnString(Outcome.fromPgn(headers['Result'])));
+    final rules = Rules.fromPgn(headers['Variant']);
+    if (rules == Rules.chessSharp) {
+      token.write(
+          Outcome.toPgnStringChessSharp(Outcome.fromPgn(headers['Result'])));
+    } else {
+      token.write(Outcome.toPgnString(Outcome.fromPgn(headers['Result'])));
+    }
     builder.writeln(token.toString());
     return builder.toString();
   }
```

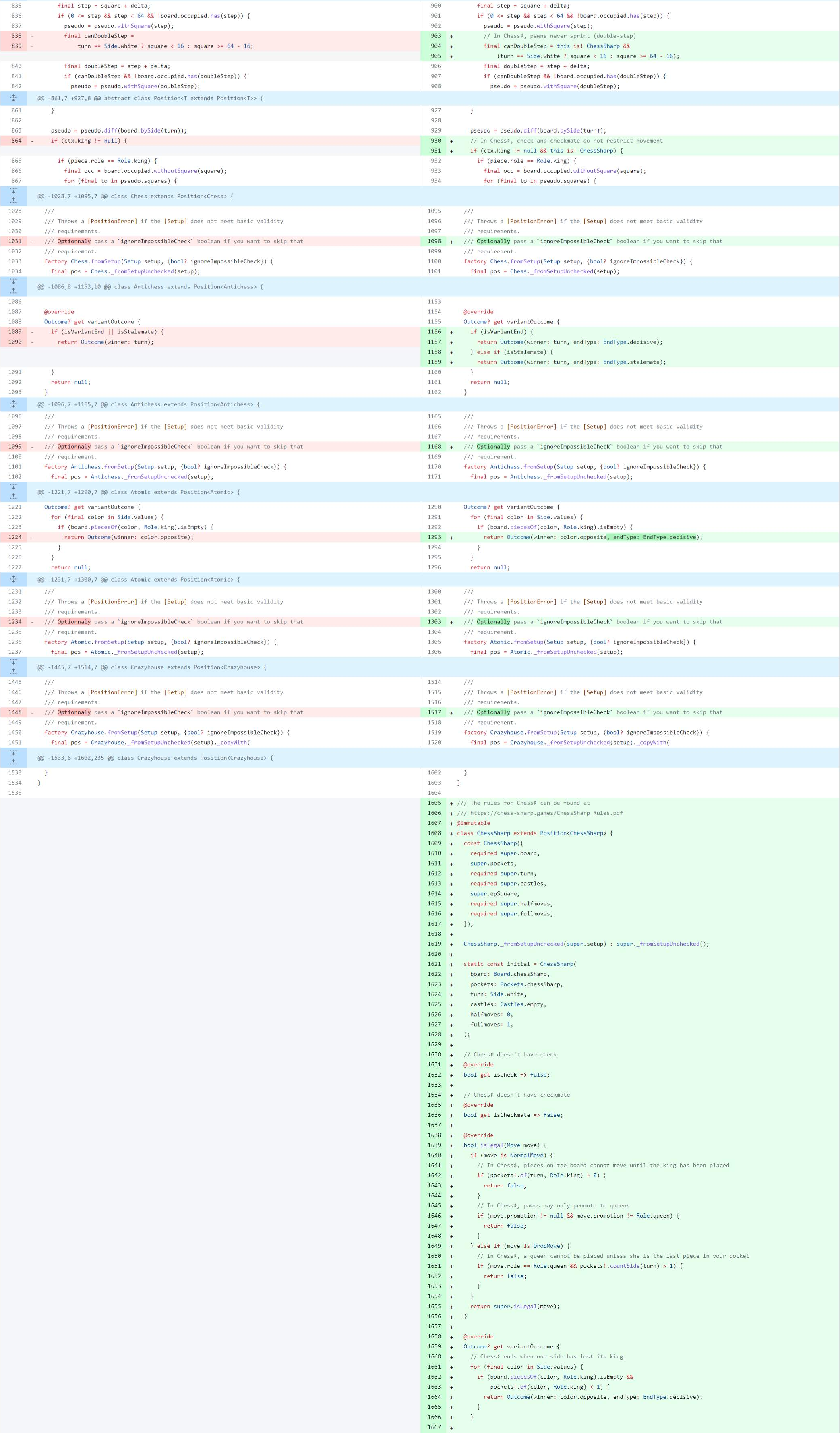
418 lib/src/position.dart ☐ Viewed

```
@@ -94,6 +94,8 @@ abstract class Position<T extends Position<T>> {
       case Rules.crazyhouse:
         return Crazyhouse.fromSetup(setup,
             ignoreImpossibleCheck: ignoreImpossibleCheck);
+      case Rules.chessSharp:
+        return ChessSharp.fromSetup(setup);
       case Rules.threecheck:
         return ThreeCheck.fromSetup(setup,
             ignoreImpossibleCheck: ignoreImpossibleCheck);
@@ -121,6 +123,8 @@ abstract class Position<T extends Position<T>> {
         return ThreeCheck.initial;
       case Rules.crazyhouse:
         return Crazyhouse.initial;
+      case Rules.chessSharp:
+        return ChessSharp.initial;
       case Rules.horde:
         return Horde.initial;
       case Rules.racingKings:
@@ -165,16 +169,39 @@ abstract class Position<T extends Position<T>> {
       !isVariantEnd && checkers.isNotEmpty && !hasSomeLegalMoves;

   /// Tests for stalemate.
```

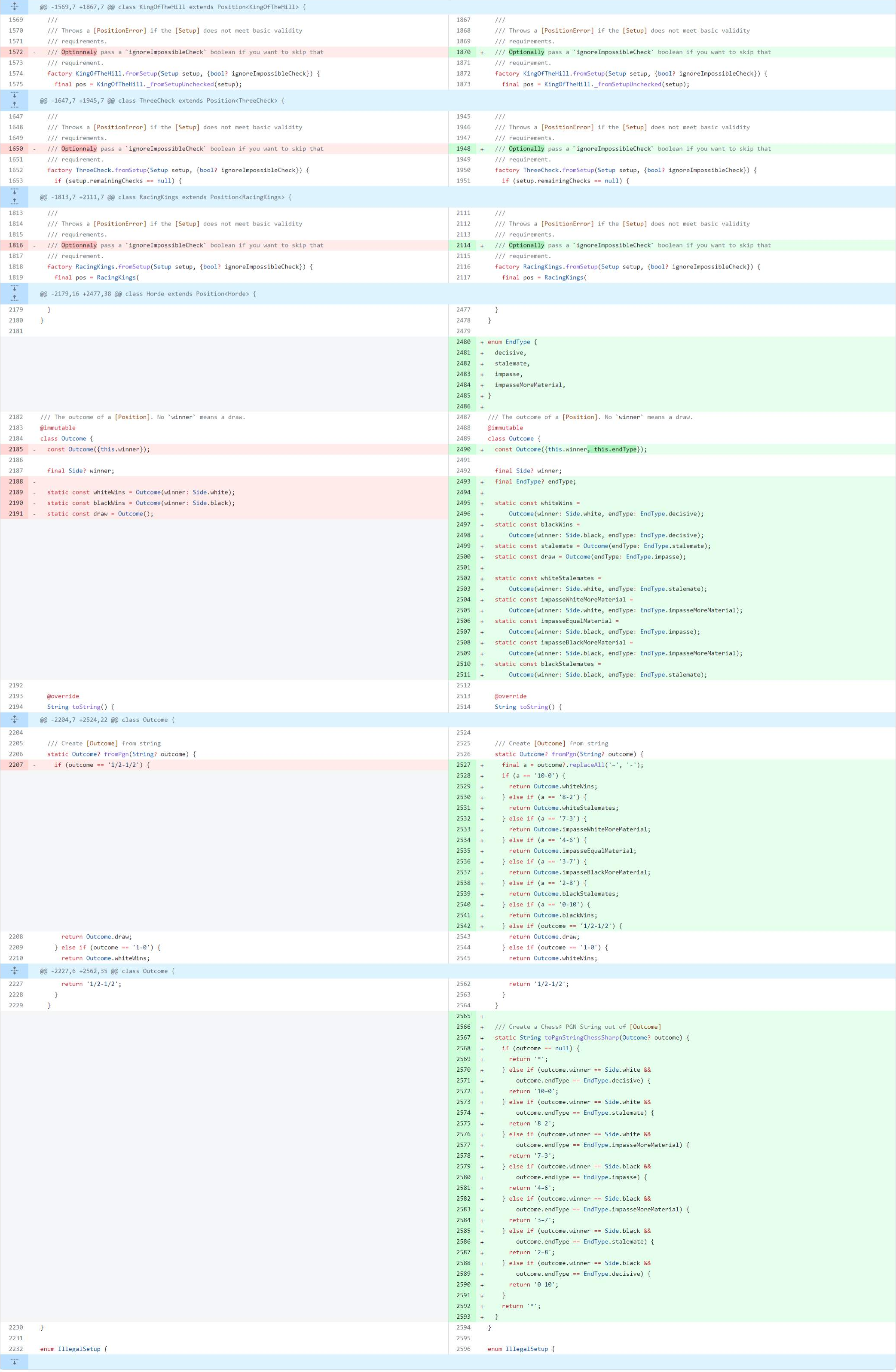


```
-  bool get isStalemate =>
-      !isVariantEnd && checkers.isEmpty && !hasSomeLegalMoves;
+  bool get isStalemate => this is ChessSharp
+      ? (!isVariantEnd && checkers.isEmpty && allMovesSuicide)
+      : (!isVariantEnd && checkers.isEmpty && !hasSomeLegalMoves);
+
+  /// Tests for impasse (50-moves without a pawn move or capture).
+  bool get isImpasse => halfmoves >= 100;
+
+  /// Returns diff of material value between [Side.white] and [Side.black].
+  ///
+  /// Positive values indicate an advantage for white, negative for black.
+  /// Zero indicates material equality. (Does not include kings or pieces in the pocket.)
+  num get materialDiff {
+    return (1 * board.piecesOf(Side.white, Role.pawn).size +
+            3 * board.piecesOf(Side.white, Role.knight).size +
+            3 * board.piecesOf(Side.white, Role.bishop).size +
+            5 * board.piecesOf(Side.white, Role.rook).size +
+            9 * board.piecesOf(Side.white, Role.queen).size) -
+        (1 * board.piecesOf(Side.black, Role.pawn).size +
+            3 * board.piecesOf(Side.black, Role.knight).size +
+            3 * board.piecesOf(Side.black, Role.bishop).size +
+            5 * board.piecesOf(Side.black, Role.rook).size +
+            9 * board.piecesOf(Side.black, Role.queen).size);
+  }
 
   /// The outcome of the game, or `null` if the game is not over.
   Outcome? get outcome {
     if (variantOutcome != null) {
       return variantOutcome;
     } else if (isCheckmate) {
-      return Outcome(winner: turn.opposite);
-    } else if (isInsufficientMaterial || isStalemate) {
+      return Outcome(winner: turn.opposite, endType: EndType.decisive);
+    } else if (isStalemate) {
+      return Outcome.stalemate;
+    } else if (isInsufficientMaterial) {
       return Outcome.draw;
     } else {
       return null;
@@ -194,6 +221,26 @@ abstract class Position<T extends Position<T>> {
     return false;
   }
 
+  /// Returns true if all legal moves put the king in check.
+  ///
+  /// A test to help evalutate stalemate in variants like Chess♯ where moving
+  /// into check is legal.
+  bool get allMovesSuicide {
+    if (pockets != null && pockets!.countSide(turn) > 0) {
+      if (legalDrops.size > 0) return false;
+    }
+    final context = _makeContext();
+    for (final square in board.bySide(turn).squares) {
+      for (final newSquare in _legalMovesOf(square, context: context).squares) {
+        final a = this.play(NormalMove(to: newSquare, from: square));
+        if (a.checkersOnThem.isEmpty) {
+          return false;
+        }
+      }
+    }
+    return true;
+  }
+
   /// Gets all the legal moves of this position.
   IMap<Square, SquareSet> get legalMoves {
     final context = _makeContext();
@@ -213,6 +260,12 @@ abstract class Position<T extends Position<T>> {
     return king != null ? kingAttackers(king, turn.opposite) : SquareSet.empty;
   }
 
+  /// SquareSet of our pieces giving check to their king.
+  SquareSet get checkersOnThem {
+    final king = board.kingOf(turn.opposite);
+    return king != null ? kingAttackers(king, turn) : SquareSet.empty;
+  }
+
   /// Attacks that a king on `square` would have to deal with.
   SquareSet kingAttackers(Square square, Side attacker, {SquareSet? occupied}) {
     return board.attacksTo(square, attacker, occupied: occupied);
@@ -442,7 +495,7 @@ abstract class Position<T extends Position<T>> {
       return move;
     }
 
-    // The final two moves define the destination
+    // The final two chars define the destination
     final destination = parseSquare(san.substring(san.length - 2));
     if (destination == null) {
       return null;
@@ -557,6 +610,15 @@ abstract class Position<T extends Position<T>> {
     }
   }
 
+  /// Plays a move from a Standard Algebraic Notation string.
+  Position<T> playSan(String san) {
+    final move = parseSan(san);
+    if (move == null) {
+      throw PlayError('Invalid SAN $san');
+    }
+    return play(move);
+  }
+
   /// Plays a move without checking if the move is legal.
   Position<T> playUnchecked(Move move) {
     assert(move is NormalMove || move is DropMove);
@@ -618,7 +680,10 @@ abstract class Position<T extends Position<T>> {
       return _copyWith(
         halfmoves: isCapture || piece.role == Role.pawn ? 0 : halfmoves + 1,
         fullmoves: turn == Side.black ? fullmoves + 1 : fullmoves,
-        pockets: Box(capturedPiece != null
+        // Chess♯ does not put captured pieces in the pocket the way Crazyhouse does.
+        // (The major pieces start in the pocket for the sake of opening variety.
+        // After that, when they die, they die, just like classical chess.)
+        pockets: Box((this is! ChessSharp && capturedPiece != null)
             ? pockets?.increment(capturedPiece.color.opposite,
                 capturedPiece.promoted ? Role.pawn : capturedPiece.role)
             : pockets),
@@ -819,7 +884,7 @@ abstract class Position<T extends Position<T>> {
 
   /// Gets the legal moves for that [Square].
   ///
-  /// Optionnaly pass a [_Context] of the position, to optimize performance when
+  /// Optionally pass a [_Context] of the position, to optimize performance when
   /// calling this method several times.
   SquareSet _legalMovesOf(Square square, {_Context? context}) {
     final ctx = context ?? _makeContext();
@@ -835,8 +900,9 @@ abstract class Position<T extends Position<T>> {
```

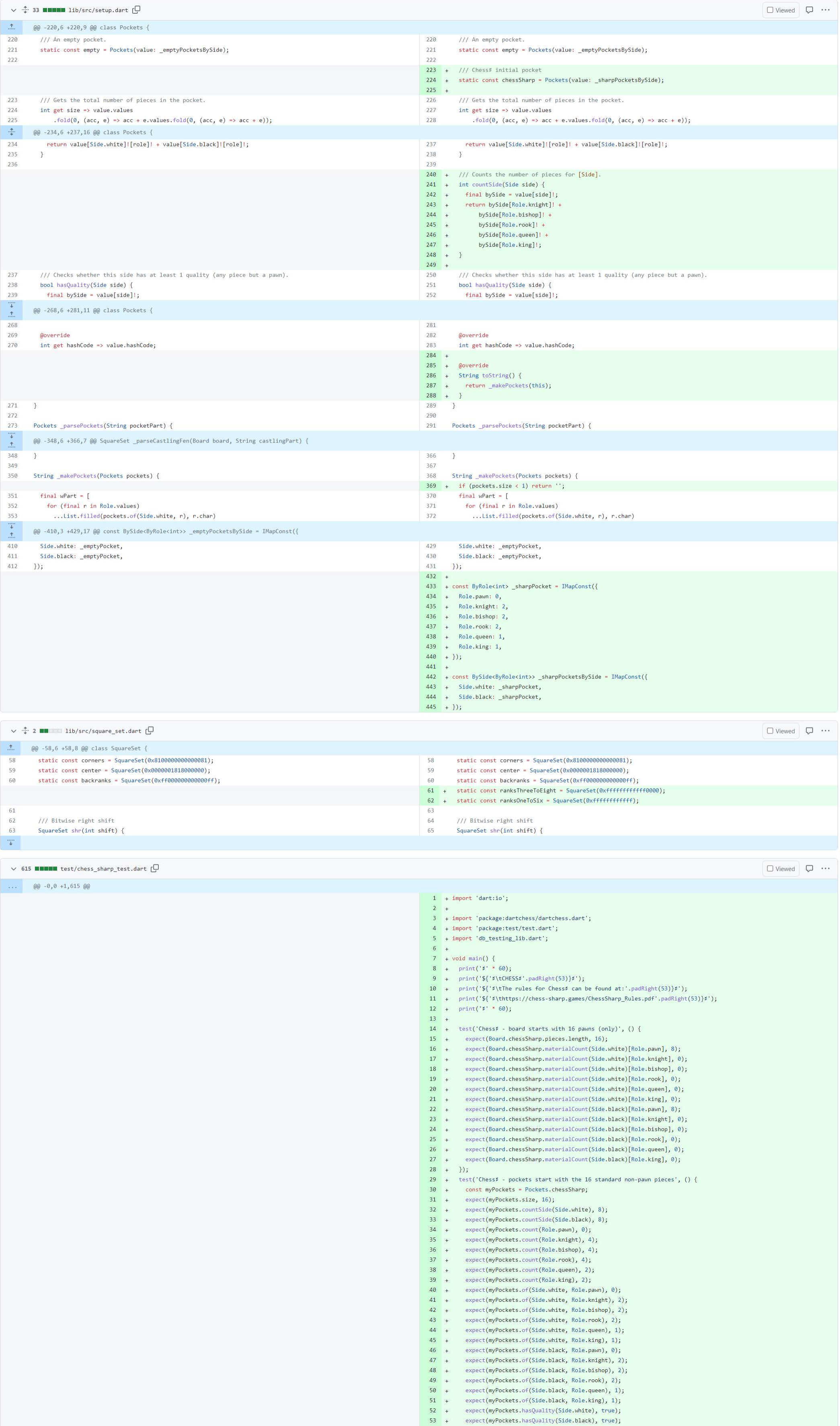


```diff
       final step = square + delta;
       if (0 <= step && step < 64 && !board.occupied.has(step)) {
         pseudo = pseudo.withSquare(step);
-        final canDoubleStep =
-            turn == Side.white ? square < 16 : square >= 64 - 16;
+        // In Chess♯, pawns never sprint (double-step)
+        final canDoubleStep = this is! ChessSharp &&
+            (turn == Side.white ? square < 16 : square >= 64 - 16);
         final doubleStep = step + delta;
         if (canDoubleStep && !board.occupied.has(doubleStep)) {
           pseudo = pseudo.withSquare(doubleStep);
@@ -861,7 +927,8 @@ abstract class Position<T extends Position<T>> {
     }
 
     pseudo = pseudo.diff(board.bySide(turn));
-    if (ctx.king != null) {
+    // In Chess♯, check and checkmate do not restrict movement
+    if (ctx.king != null && this is! ChessSharp) {
       if (piece.role == Role.king) {
         final occ = board.occupied.withoutSquare(square);
         for (final to in pseudo.squares) {
@@ -1028,7 +1095,7 @@ class Chess extends Position<Chess> {
   ///
   /// Throws a [PositionError] if the [Setup] does not meet basic validity
   /// requirements.
-  /// Optionnaly pass a `ignoreImpossibleCheck` boolean if you want to skip that
+  /// Optionally pass a `ignoreImpossibleCheck` boolean if you want to skip that
   /// requirement.
   factory Chess.fromSetup(Setup setup, {bool? ignoreImpossibleCheck}) {
     final pos = Chess._fromSetupUnchecked(setup);
@@ -1086,8 +1153,10 @@ class Antichess extends Position<Antichess> {
 
   @override
   Outcome? get variantOutcome {
-    if (isVariantEnd || isStalemate) {
-      return Outcome(winner: turn);
+    if (isVariantEnd) {
+      return Outcome(winner: turn, endType: EndType.decisive);
+    } else if (isStalemate) {
+      return Outcome(winner: turn, endType: EndType.stalemate);
     }
     return null;
   }
@@ -1096,7 +1165,7 @@ class Antichess extends Position<Antichess> {
   ///
   /// Throws a [PositionError] if the [Setup] does not meet basic validity
   /// requirements.
-  /// Optionnaly pass a `ignoreImpossibleCheck` boolean if you want to skip that
+  /// Optionally pass a `ignoreImpossibleCheck` boolean if you want to skip that
   /// requirement.
   factory Antichess.fromSetup(Setup setup, {bool? ignoreImpossibleCheck}) {
     final pos = Antichess._fromSetupUnchecked(setup);
@@ -1221,7 +1290,7 @@ class Atomic extends Position<Atomic> {
   Outcome? get variantOutcome {
     for (final color in Side.values) {
       if (board.piecesOf(color, Role.king).isEmpty) {
-        return Outcome(winner: color.opposite);
+        return Outcome(winner: color.opposite, endType: EndType.decisive);
       }
     }
     return null;
@@ -1231,7 +1300,7 @@ class Atomic extends Position<Atomic> {
   ///
   /// Throws a [PositionError] if the [Setup] does not meet basic validity
   /// requirements.
-  /// Optionnaly pass a `ignoreImpossibleCheck` boolean if you want to skip that
+  /// Optionally pass a `ignoreImpossibleCheck` boolean if you want to skip that
   /// requirement.
   factory Atomic.fromSetup(Setup setup, {bool? ignoreImpossibleCheck}) {
     final pos = Atomic._fromSetupUnchecked(setup);
@@ -1445,7 +1514,7 @@ class Crazyhouse extends Position<Crazyhouse> {
   ///
   /// Throws a [PositionError] if the [Setup] does not meet basic validity
   /// requirements.
-  /// Optionnaly pass a `ignoreImpossibleCheck` boolean if you want to skip that
+  /// Optionally pass a `ignoreImpossibleCheck` boolean if you want to skip that
   /// requirement.
   factory Crazyhouse.fromSetup(Setup setup, {bool? ignoreImpossibleCheck}) {
     final pos = Crazyhouse._fromSetupUnchecked(setup)._copyWith(
@@ -1533,6 +1602,235 @@ class Crazyhouse extends Position<Crazyhouse> {
   }
 }
 
+/// The rules for Chess♯ can be found at
+/// https://chess-sharp.games/ChessSharp_Rules.pdf
+@immutable
+class ChessSharp extends Position<ChessSharp> {
+  const ChessSharp({
+    required super.board,
+    super.pockets,
+    required super.turn,
+    required super.castles,
+    super.epSquare,
+    required super.halfmoves,
+    required super.fullmoves,
+  });
+
+  ChessSharp._fromSetupUnchecked(super.setup) : super._fromSetupUnchecked();
+
+  static const initial = ChessSharp(
+    board: Board.chessSharp,
+    pockets: Pockets.chessSharp,
+    turn: Side.white,
+    castles: Castles.empty,
+    halfmoves: 0,
+    fullmoves: 1,
+  );
+
+  // Chess♯ doesn't have check
+  @override
+  bool get isCheck => false;
+
+  // Chess♯ doesn't have checkmate
+  @override
+  bool get isCheckmate => false;
+
+  @override
+  bool isLegal(Move move) {
+    if (move is NormalMove) {
+      // In Chess♯, pieces on the board cannot move until the king has been placed
+      if (pockets!.of(turn, Role.king) > 0) {
+        return false;
+      }
+      // In Chess♯, pawns may only promote to queens
+      if (move.promotion != null && move.promotion != Role.queen) {
+        return false;
+      }
+    } else if (move is DropMove) {
+      // In Chess♯, a queen cannot be placed unless she is the last piece in your pocket
+      if (move.role == Role.queen && pockets!.countSide(turn) > 1) {
+        return false;
+      }
+    }
+    return super.isLegal(move);
+  }
+
+  @override
+  Outcome? get variantOutcome {
+    // Chess♯ ends when one side has lost its king
+    for (final color in Side.values) {
+      if (board.piecesOf(color, Role.king).isEmpty &&
+          pockets!.of(color, Role.king) < 1) {
+        return Outcome(winner: color.opposite, endType: EndType.decisive);
+      }
+    }
+
```

```diff
+    // Chess♯ stalemates are partial victories
+    if (isStalemate) {
+      return Outcome(winner: turn.opposite, endType: EndType.stalemate);
+    }
+
+    if (isImpasse || isInsufficientMaterial) {
+      if (materialDiff > 0) {
+        return Outcome.impasseWhiteMoreMaterial;
+      } else if (materialDiff < 0) {
+        return Outcome.impasseBlackMoreMaterial;
+      }
+      return Outcome.impasseEqualMaterial;
+    }
+    return null;
+  }
+
+  @override
+  bool get isVariantEnd => (board.kings.size + pockets!.count(Role.king)) < 2;
+
+  /// Set up a playable [ChessSharp] position.
+  ///
+  /// Throws a [PositionError] if the [Setup] does not meet basic validity
+  /// requirements.
+  factory ChessSharp.fromSetup(Setup setup) {
+    final pos = ChessSharp._fromSetupUnchecked(setup)._copyWith(
+      pockets: Box(setup.pockets ?? Pockets.empty),
+      board: setup.board.withPromoted(setup.board.promoted
+          .intersect(setup.board.occupied)
+          .diff(setup.board.kings)
+          .diff(setup.board.pawns)),
+    );
+    pos.validate();
+    return pos;
+  }
+
+  @override
+  void validate({bool? ignoreImpossibleCheck}) {
+    if (board.occupied.isEmpty) {
+      throw PositionError.empty;
+    }
+    if (SquareSet.backranks.isIntersected(board.pawns)) {
+      throw PositionError.pawnsOnBackrank;
+    }
+    if (pockets == null) {
+      throw PositionError.variant;
+    } else {
+      // > 32 pieces on the board+pocket
+      if ((pockets!.size + board.occupied.size) > 32) {
+        throw PositionError.variant;
+      }
+      // > 2 kings on the board+pocket
+      if ((board.kings.size +
+              pockets!.of(Side.white, Role.king) +
+              pockets!.of(Side.black, Role.king)) >
+          2) {
+        throw PositionError.variant;
+      }
+      // > 4 rooks on the board+pocket (Chess♯ doesn't have underpromotions)
+      if ((board.rooks.size +
+              pockets!.of(Side.white, Role.rook) +
+              pockets!.of(Side.black, Role.rook)) >
+          4) {
+        throw PositionError.variant;
+      }
+      // > 4 bishops on the board+pocket (Chess# doesn't have underpromotions)
+      if ((board.bishops.size +
+              pockets!.of(Side.white, Role.bishop) +
+              pockets!.of(Side.black, Role.bishop)) >
+          4) {
+        throw PositionError.variant;
+      }
+      // > 4 knights on the board+pocket (Chess# doesn't have underpromotions)
+      if ((board.knights.size +
+              pockets!.of(Side.white, Role.knight) +
+              pockets!.of(Side.black, Role.knight)) >
+          4) {
+        throw PositionError.variant;
+      }
+      // Has piece(s) in the pocket, but the queen is not in the pocket (Chess# requires the queen to be placed last)
+      if ((pockets!.countSide(Side.white) > 0 &&
+              pockets!.of(Side.white, Role.queen) < 1) ||
+          pockets!.countSide(Side.black) > 0 &&
+              pockets!.of(Side.black, Role.queen) < 1) {
+        throw PositionError.variant;
+      }
+      // Chess# requires the king to be placed before pieces on the board move
+      // White piece(s) have moved into rows 3-8, but the king is still in the pocket
+      if ((pockets!.of(Side.white, Role.king) > 0) &&
+          (board.white.intersect(SquareSet.ranksThreeToEight).isNotEmpty)) {
+        throw PositionError.variant;
+      }
+      // Black piece(s) have moved into rows 1-6, but the king is still in the pocket
+      if ((pockets!.of(Side.black, Role.king) > 0) &&
+          (board.black.intersect(SquareSet.ranksOneToSix).isNotEmpty)) {
+        throw PositionError.variant;
+      }
+    }
+  }
+
+  @override
+  bool hasInsufficientMaterial(Side side) {
+    if (pockets == null) {
+      return super.hasInsufficientMaterial(side);
+    }
+    return board.occupied.size + pockets!.size <= 3 &&
+        board.pawns.isEmpty &&
+        board.promoted.isEmpty &&
+        board.rooksAndQueens.isEmpty &&
+        pockets!.count(Role.pawn) <= 0 &&
+        pockets!.count(Role.rook) <= 0 &&
+        pockets!.count(Role.queen) <= 0;
+  }
+
+  @override
+  SquareSet get legalDrops {
+    final mask = board.occupied.complement().intersect(turn == Side.white
+        ? const SquareSet.fromRank(0)
+        : const SquareSet.fromRank(7));
+
+    return mask;
+  }
+
+  @override
+  String toSan(Move move) {
+    // Chess# notation does not use check (+) or checkmate (#) symbols
+    if (isLegal(move)) {
+      final san = _makeSanWithoutSuffix(move);
+      return san;
+    } else {
+      throw PlayError('Invalid move $move');
+    }
+  }
+
+  @override
+  Tuple2<Position<ChessSharp>, String> playToSan(Move move) {
+    // Chess# notation does not use check (+) or checkmate (#) symbols
+    if (isLegal(move)) {
+      final san = _makeSanWithoutSuffix(move);
+      final newPos = playUnchecked(move);
+      return Tuple2(newPos, san);
+    } else {
+      throw PlayError('Invalid move $move');
+    }
+  }
+
+  @override
+  ChessSharp _copyWith({
+    Board? board,
+    Box<Pockets?>? pockets,
+    Side? turn,
+    Castles? castles,
+    Box<Square?>? epSquare,
+    int? halfmoves,
+    int? fullmoves,
+  }) {
+    return ChessSharp(
+      board: board ?? this.board,
+      pockets: pockets != null ? pockets.value : this.pockets,
+      turn: turn ?? this.turn,
+      castles: castles ?? this.castles,
+      halfmoves: halfmoves ?? this.halfmoves,
+      fullmoves: fullmoves ?? this.fullmoves,
+    );
+  }
+}
+
 /// A variant similar to standard chess, where you win by putting your king on the center
 /// of the board.
 @immutable
@@ -1559,7 +1857,7 @@ class KingOfTheHill extends Position<KingOfTheHill> {
   Outcome? get variantOutcome {
     for (final color in Side.values) {
       if (board.piecesOf(color, Role.king).isIntersected(SquareSet.center)) {
-        return Outcome(winner: color);
+        return Outcome(winner: color, endType: EndType.decisive);
       }
     }
     return null;
```



```diff
@@ -1569,7 +1867,7 @@ class KingOfTheHill extends Position<KingOfTheHill> {
   ///
   /// Throws a [PositionError] if the [Setup] does not meet basic validity
   /// requirements.
-  /// Optionnaly pass a `ignoreImpossibleCheck` boolean if you want to skip that
+  /// Optionally pass a `ignoreImpossibleCheck` boolean if you want to skip that
   /// requirement.
   factory KingOfTheHill.fromSetup(Setup setup, {bool? ignoreImpossibleCheck}) {
     final pos = KingOfTheHill._fromSetupUnchecked(setup);
@@ -1647,7 +1945,7 @@ class ThreeCheck extends Position<ThreeCheck> {
   ///
   /// Throws a [PositionError] if the [Setup] does not meet basic validity
   /// requirements.
-  /// Optionnaly pass a `ignoreImpossibleCheck` boolean if you want to skip that
+  /// Optionally pass a `ignoreImpossibleCheck` boolean if you want to skip that
   /// requirement.
   factory ThreeCheck.fromSetup(Setup setup, {bool? ignoreImpossibleCheck}) {
     if (setup.remainingChecks == null) {
@@ -1813,7 +2111,7 @@ class RacingKings extends Position<RacingKings> {
   ///
   /// Throws a [PositionError] if the [Setup] does not meet basic validity
   /// requirements.
-  /// Optionnaly pass a `ignoreImpossibleCheck` boolean if you want to skip that
+  /// Optionally pass a `ignoreImpossibleCheck` boolean if you want to skip that
   /// requirement.
   factory RacingKings.fromSetup(Setup setup, {bool? ignoreImpossibleCheck}) {
     final pos = RacingKings(
@@ -2179,16 +2477,38 @@ class Horde extends Position<Horde> {
   }
 }
 
+enum EndType {
+  decisive,
+  stalemate,
+  impasse,
+  impasseMoreMaterial,
+}
+
 /// The outcome of a [Position]. No `winner` means a draw.
 @immutable
 class Outcome {
-  const Outcome({this.winner});
+  const Outcome({this.winner, this.endType});
 
   final Side? winner;
-
-  static const whiteWins = Outcome(winner: Side.white);
-  static const blackWins = Outcome(winner: Side.black);
-  static const draw = Outcome();
+  final EndType? endType;
+
+  static const whiteWins =
+      Outcome(winner: Side.white, endType: EndType.decisive);
+  static const blackWins =
+      Outcome(winner: Side.black, endType: EndType.decisive);
+  static const stalemate = Outcome(endType: EndType.stalemate);
+  static const draw = Outcome(endType: EndType.impasse);
+
+  static const whiteStalemates =
+      Outcome(winner: Side.white, endType: EndType.stalemate);
+  static const impasseWhiteMoreMaterial =
+      Outcome(winner: Side.white, endType: EndType.impasseMoreMaterial);
+  static const impasseEqualMaterial =
+      Outcome(winner: Side.black, endType: EndType.impasse);
+  static const impasseBlackMoreMaterial =
+      Outcome(winner: Side.black, endType: EndType.impasseMoreMaterial);
+  static const blackStalemates =
+      Outcome(winner: Side.black, endType: EndType.stalemate);
 
   @override
   String toString() {
@@ -2204,7 +2524,22 @@ class Outcome {
 
   /// Create [Outcome] from string
   static Outcome? fromPgn(String? outcome) {
-    if (outcome == '1/2-1/2') {
+    final a = outcome?.replaceAll('–', '-');
+    if (a == '10-0') {
+      return Outcome.whiteWins;
+    } else if (a == '8-2') {
+      return Outcome.whiteStalemates;
+    } else if (a == '7-3') {
+      return Outcome.impasseWhiteMoreMaterial;
+    } else if (a == '4-6') {
+      return Outcome.impasseEqualMaterial;
+    } else if (a == '3-7') {
+      return Outcome.impasseBlackMoreMaterial;
+    } else if (a == '2-8') {
+      return Outcome.blackStalemates;
+    } else if (a == '0-10') {
+      return Outcome.blackWins;
+    } else if (outcome == '1/2-1/2') {
       return Outcome.draw;
     } else if (outcome == '1-0') {
       return Outcome.whiteWins;
@@ -2227,6 +2562,35 @@ class Outcome {
       return '1/2-1/2';
     }
   }
+
+  /// Create a Chess# PGN String out of [Outcome]
+  static String toPgnStringChessSharp(Outcome? outcome) {
+    if (outcome == null) {
+      return '*';
+    } else if (outcome.winner == Side.white &&
+        outcome.endType == EndType.decisive) {
+      return '10–0';
+    } else if (outcome.winner == Side.white &&
+        outcome.endType == EndType.stalemate) {
+      return '8–2';
+    } else if (outcome.winner == Side.white &&
+        outcome.endType == EndType.impasseMoreMaterial) {
+      return '7–3';
+    } else if (outcome.winner == Side.black &&
+        outcome.endType == EndType.impasse) {
+      return '4–6';
+    } else if (outcome.winner == Side.black &&
+        outcome.endType == EndType.impasseMoreMaterial) {
+      return '3–7';
+    } else if (outcome.winner == Side.black &&
+        outcome.endType == EndType.stalemate) {
+      return '2–8';
+    } else if (outcome.winner == Side.black &&
+        outcome.endType == EndType.decisive) {
+      return '0–10';
+    }
+    return '*';
+  }
 }
 
 enum IllegalSetup {
```



33 lib/src/setup.dart

```diff
@@ -220,6 +220,9 @@ class Pockets {
   /// An empty pocket.
   static const empty = Pockets(value: _emptyPocketsBySide);
 
+  /// Chess♯ initial pocket
+  static const chessSharp = Pockets(value: _sharpPocketsBySide);
+
   /// Gets the total number of pieces in the pocket.
   int get size => value.values
       .fold(0, (acc, e) => acc + e.values.fold(0, (acc, e) => acc + e));
@@ -234,6 +237,16 @@ class Pockets {
     return value[Side.white]![role]! + value[Side.black]![role]!;
   }
 
+  /// Counts the number of pieces for [Side].
+  int countSide(Side side) {
+    final bySide = value[side]!;
+    return bySide[Role.knight]! +
+        bySide[Role.bishop]! +
+        bySide[Role.rook]! +
+        bySide[Role.queen]! +
+        bySide[Role.king]!;
+  }
+
   /// Checks whether this side has at least 1 quality (any piece but a pawn).
   bool hasQuality(Side side) {
     final bySide = value[side]!;
@@ -268,6 +281,11 @@ class Pockets {
 
   @override
   int get hashCode => value.hashCode;
+
+  @override
+  String toString() {
+    return _makePockets(this);
+  }
 }
 
 Pockets _parsePockets(String pocketPart) {
@@ -348,6 +366,7 @@ SquareSet _parseCastlingFen(Board board, String castlingPart) {
 }
 
 String _makePockets(Pockets pockets) {
+  if (pockets.size < 1) return '';
   final wPart = [
     for (final r in Role.values)
       ...List.filled(pockets.of(Side.white, r), r.char)
@@ -410,3 +429,17 @@ const BySide<ByRole<int>> _emptyPocketsBySide = IMapConst({
   Side.white: _emptyPocket,
   Side.black: _emptyPocket,
 });
+
+const ByRole<int> _sharpPocket = IMapConst({
+  Role.pawn: 0,
+  Role.knight: 2,
+  Role.bishop: 2,
+  Role.rook: 2,
+  Role.queen: 1,
+  Role.king: 1,
+});
+
+const BySide<ByRole<int>> _sharpPocketsBySide = IMapConst({
+  Side.white: _sharpPocket,
+  Side.black: _sharpPocket,
+});
```

2 lib/src/square_set.dart

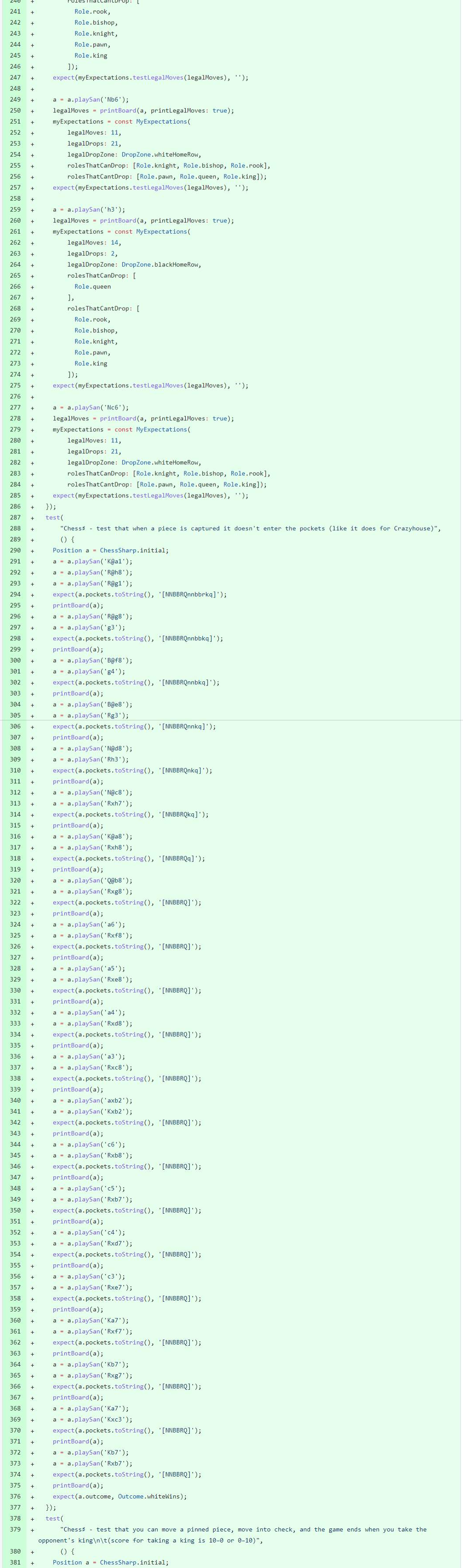
```diff
@@ -58,6 +58,8 @@ class SquareSet {
   static const corners = SquareSet(0x8100000000000081);
   static const center = SquareSet(0x0000001818000000);
   static const backranks = SquareSet(0xff000000000000ff);
+  static const ranksThreeToEight = SquareSet(0xffffffffffff0000);
+  static const ranksOneToSix = SquareSet(0xffffffffffff);
 
   /// Bitwise right shift
   SquareSet shr(int shift) {
```

615 test/chess_sharp_test.dart

```diff
@@ -0,0 +1,615 @@
+import 'dart:io';
+
+import 'package:dartchess/dartchess.dart';
+import 'package:test/test.dart';
+import 'db_testing_lib.dart';
+
+void main() {
+  print('♯' * 60);
+  print('${'♯\tCHESS♯'.padRight(53)}♯');
+  print('${'♯\tThe rules for Chess♯ can be found at:'.padRight(53)}♯');
+  print('${'♯\thttps://chess-sharp.games/ChessSharp_Rules.pdf'.padRight(53)}♯');
+  print('♯' * 60);
+
+  test('Chess♯ - board starts with 16 pawns (only)', () {
+    expect(Board.chessSharp.pieces.length, 16);
+    expect(Board.chessSharp.materialCount(Side.white)[Role.pawn], 8);
+    expect(Board.chessSharp.materialCount(Side.white)[Role.knight], 0);
+    expect(Board.chessSharp.materialCount(Side.white)[Role.bishop], 0);
+    expect(Board.chessSharp.materialCount(Side.white)[Role.rook], 0);
+    expect(Board.chessSharp.materialCount(Side.white)[Role.queen], 0);
+    expect(Board.chessSharp.materialCount(Side.white)[Role.king], 0);
+    expect(Board.chessSharp.materialCount(Side.black)[Role.pawn], 8);
+    expect(Board.chessSharp.materialCount(Side.black)[Role.knight], 0);
+    expect(Board.chessSharp.materialCount(Side.black)[Role.bishop], 0);
+    expect(Board.chessSharp.materialCount(Side.black)[Role.rook], 0);
+    expect(Board.chessSharp.materialCount(Side.black)[Role.queen], 0);
+    expect(Board.chessSharp.materialCount(Side.black)[Role.king], 0);
+  });
+  test('Chess♯ - pockets start with the 16 standard non-pawn pieces', () {
+    const myPockets = Pockets.chessSharp;
+    expect(myPockets.size, 16);
+    expect(myPockets.countSide(Side.white), 8);
+    expect(myPockets.countSide(Side.black), 8);
+    expect(myPockets.count(Role.pawn), 0);
+    expect(myPockets.count(Role.knight), 4);
+    expect(myPockets.count(Role.bishop), 4);
+    expect(myPockets.count(Role.rook), 4);
+    expect(myPockets.count(Role.queen), 2);
+    expect(myPockets.count(Role.king), 2);
+    expect(myPockets.of(Side.white, Role.pawn), 0);
+    expect(myPockets.of(Side.white, Role.knight), 2);
+    expect(myPockets.of(Side.white, Role.bishop), 2);
+    expect(myPockets.of(Side.white, Role.rook), 2);
+    expect(myPockets.of(Side.white, Role.queen), 1);
+    expect(myPockets.of(Side.white, Role.king), 1);
+    expect(myPockets.of(Side.black, Role.pawn), 0);
+    expect(myPockets.of(Side.black, Role.knight), 2);
+    expect(myPockets.of(Side.black, Role.bishop), 2);
+    expect(myPockets.of(Side.black, Role.rook), 2);
+    expect(myPockets.of(Side.black, Role.queen), 1);
+    expect(myPockets.of(Side.black, Role.king), 1);
+    expect(myPockets.hasQuality(Side.white), true);
+    expect(myPockets.hasQuality(Side.black), true);
```

```
+      expect(myPockets.hasPawn(Side.white), false);
+      expect(myPockets.hasPawn(Side.black), false);
+      expect(myPockets.toString(), '[NNBBRRKQnnbbrrkq]');
+    });
+    test('Chess♯ - inital fen', () {
+      expect(ChessSharp.initial.fen,
+          '8/pppppppp/8/8/8/8/PPPPPPPP/8[NNBBRRKQnnbbrrkq] w - - 0 1');
+
+      final loadedPosition = ChessSharp.fromSetup(Setup.parseFen(
+          '8/pppppppp/8/8/8/8/PPPPPPPP/8[NNBBRRKQnnbbrrkq] w - - 0 1'));
+      expect(ChessSharp.initial.fen, loadedPosition.fen);
+    });
+    test('Chess♯ - kingCount', () {
+      const ChessSharp a = ChessSharp.initial;
+      expect(a.board.kings.size, 0);
+      expect(a.pockets!.of(Side.white, Role.king), 1);
+      expect(a.pockets!.of(Side.black, Role.king), 1);
+    });
+    test('Chess♯ - test all legal moves and drops from the beginning', () {
+      Position a = ChessSharp.initial;
+      List<Move> legalMoves = printBoard(a, printLegalMoves: true);
+      MyExpectations myExpectations = const MyExpectations(
+          legalMoves: 0,
+          legalDrops: 32,
+          legalDropZone: DropZone.whiteHomeRow,
+          rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook, Role.king],
+          rolesThatCantDrop: [Role.pawn, Role.queen]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('K@a1');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 0,
+          legalDrops: 32,
+          legalDropZone: DropZone.blackHomeRow,
+          rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook, Role.king],
+          rolesThatCantDrop: [Role.pawn, Role.queen]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('N@a8');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 9,
+          legalDrops: 21,
+          legalDropZone: DropZone.whiteHomeRow,
+          rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook],
+          rolesThatCantDrop: [Role.pawn, Role.queen, Role.king]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('a3');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 0,
+          legalDrops: 28,
+          legalDropZone: DropZone.blackHomeRow,
+          rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook, Role.king],
+          rolesThatCantDrop: [Role.pawn, Role.queen]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('N@b8');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 10,
+          legalDrops: 21,
+          legalDropZone: DropZone.whiteHomeRow,
+          rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook],
+          rolesThatCantDrop: [Role.pawn, Role.queen, Role.king]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('b3');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 0,
+          legalDrops: 18,
+          legalDropZone: DropZone.blackHomeRow,
+          rolesThatCanDrop: [Role.bishop, Role.rook, Role.king],
+          rolesThatCantDrop: [Role.knight, Role.pawn, Role.queen]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('B@c8');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 11,
+          legalDrops: 21,
+          legalDropZone: DropZone.whiteHomeRow,
+          rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook],
+          rolesThatCantDrop: [Role.pawn, Role.queen, Role.king]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('c3');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 0,
+          legalDrops: 15,
+          legalDropZone: DropZone.blackHomeRow,
+          rolesThatCanDrop: [Role.bishop, Role.rook, Role.king],
+          rolesThatCantDrop: [Role.knight, Role.pawn, Role.queen]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('B@d8');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 11,
+          legalDrops: 21,
+          legalDropZone: DropZone.whiteHomeRow,
+          rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook],
+          rolesThatCantDrop: [Role.pawn, Role.queen, Role.king]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('d3');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 0,
+          legalDrops: 8,
+          legalDropZone: DropZone.blackHomeRow,
+          rolesThatCanDrop: [Role.rook, Role.king],
+          rolesThatCantDrop: [Role.bishop, Role.knight, Role.pawn, Role.queen]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('R@e8');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 11,
+          legalDrops: 21,
+          legalDropZone: DropZone.whiteHomeRow,
+          rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook],
+          rolesThatCantDrop: [Role.pawn, Role.queen, Role.king]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('e3');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
+          legalMoves: 0,
+          legalDrops: 6,
+          legalDropZone: DropZone.blackHomeRow,
+          rolesThatCanDrop: [Role.rook, Role.king],
+          rolesThatCantDrop: [Role.bishop, Role.knight, Role.pawn, Role.queen]);
+      expect(myExpectations.testLegalMoves(legalMoves), '');
+
+      a = a.playSan('R@f8');
+      legalMoves = printBoard(a, printLegalMoves: true);
+      myExpectations = const MyExpectations(
```

```
+        legalMoves: 11,
+        legalDrops: 21,
+        legalDropZone: DropZone.whiteHomeRow,
+        rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook],
+        rolesThatCantDrop: [Role.pawn, Role.queen, Role.king]);
+    expect(myExpectations.testLegalMoves(legalMoves), '');
+
+    a = a.playSan('f3');
+    legalMoves = printBoard(a, printLegalMoves: true);
+    myExpectations = const MyExpectations(
+        legalMoves: 0,
+        legalDrops: 2,
+        legalDropZone: DropZone.blackHomeRow,
+        rolesThatCanDrop: [
+          Role.king
+        ],
+        rolesThatCantDrop: [
+          Role.rook,
+          Role.bishop,
+          Role.knight,
+          Role.pawn,
+          Role.queen
+        ]);
+    expect(myExpectations.testLegalMoves(legalMoves), '');
+
+    a = a.playSan('K@g8');
+    legalMoves = printBoard(a, printLegalMoves: true);
+    myExpectations = const MyExpectations(
+        legalMoves: 11,
+        legalDrops: 21,
+        legalDropZone: DropZone.whiteHomeRow,
+        rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook],
+        rolesThatCantDrop: [Role.pawn, Role.queen, Role.king]);
+    expect(myExpectations.testLegalMoves(legalMoves), '');
+
+    a = a.playSan('g3');
+    legalMoves = printBoard(a, printLegalMoves: true);
+    myExpectations = const MyExpectations(
+        legalMoves: 12,
+        legalDrops: 1,
+        legalDropZone: DropZone.blackHomeRow,
+        rolesThatCanDrop: [
+          Role.queen
+        ],
+        rolesThatCantDrop: [
```



```
+        rolesThatCantDrop: [
+          Role.rook,
+          Role.bishop,
+          Role.knight,
+          Role.pawn,
+          Role.king
+        ]);
+    expect(myExpectations.testLegalMoves(legalMoves), '');
+
+    a = a.playSan('Nb6');
+    legalMoves = printBoard(a, printLegalMoves: true);
+    myExpectations = const MyExpectations(
+        legalMoves: 11,
+        legalDrops: 21,
+        legalDropZone: DropZone.whiteHomeRow,
+        rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook],
+        rolesThatCantDrop: [Role.pawn, Role.queen, Role.king]);
+    expect(myExpectations.testLegalMoves(legalMoves), '');
+
+    a = a.playSan('h3');
+    legalMoves = printBoard(a, printLegalMoves: true);
+    myExpectations = const MyExpectations(
+        legalMoves: 14,
+        legalDrops: 2,
+        legalDropZone: DropZone.blackHomeRow,
+        rolesThatCanDrop: [
+          Role.queen
+        ],
+        rolesThatCantDrop: [
+          Role.rook,
+          Role.bishop,
+          Role.knight,
+          Role.pawn,
+          Role.king
+        ]);
+    expect(myExpectations.testLegalMoves(legalMoves), '');
+
+    a = a.playSan('Nc6');
+    legalMoves = printBoard(a, printLegalMoves: true);
+    myExpectations = const MyExpectations(
+        legalMoves: 11,
+        legalDrops: 21,
+        legalDropZone: DropZone.whiteHomeRow,
+        rolesThatCanDrop: [Role.knight, Role.bishop, Role.rook],
+        rolesThatCantDrop: [Role.pawn, Role.queen, Role.king]);
+    expect(myExpectations.testLegalMoves(legalMoves), '');
+  });
+  test(
+      "Chess♯ - test that when a piece is captured it doesn't enter the pockets (like it does for Crazyhouse)",
+      () {
+    Position a = ChessSharp.initial;
+    a = a.playSan('K@a1');
+    a = a.playSan('R@h8');
+    a = a.playSan('R@g1');
+    expect(a.pockets.toString(), '[NNBBRQnnbbrkq]');
+    printBoard(a);
+    a = a.playSan('R@g8');
+    a = a.playSan('g3');
+    expect(a.pockets.toString(), '[NNBBRQnnbbkq]');
+    printBoard(a);
+    a = a.playSan('B@f8');
+    a = a.playSan('g4');
+    expect(a.pockets.toString(), '[NNBBRQnnbkq]');
+    printBoard(a);
+    a = a.playSan('B@e8');
+    a = a.playSan('Rg3');
+    expect(a.pockets.toString(), '[NNBBRQnnkq]');
+    printBoard(a);
+    a = a.playSan('N@d8');
+    a = a.playSan('Rh3');
+    expect(a.pockets.toString(), '[NNBBRQnkq]');
+    printBoard(a);
+    a = a.playSan('N@c8');
+    a = a.playSan('Rxh7');
+    expect(a.pockets.toString(), '[NNBBRQkq]');
+    printBoard(a);
+    a = a.playSan('K@a8');
+    a = a.playSan('Rxh8');
+    expect(a.pockets.toString(), '[NNBBRQq]');
+    printBoard(a);
+    a = a.playSan('Q@b8');
+    a = a.playSan('Rxg8');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('a6');
+    a = a.playSan('Rxf8');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('a5');
+    a = a.playSan('Rxe8');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('a4');
+    a = a.playSan('Rxd8');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('a3');
+    a = a.playSan('Rxc8');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('axb2');
+    a = a.playSan('Kxb2');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('c6');
+    a = a.playSan('Rxb8');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('c5');
+    a = a.playSan('Rxb7');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('c4');
+    a = a.playSan('Rxd7');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('c3');
+    a = a.playSan('Rxe7');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('Ka7');
+    a = a.playSan('Rxf7');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('Kb7');
+    a = a.playSan('Rxg7');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('Ka7');
+    a = a.playSan('Kxc3');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    a = a.playSan('Kb7');
+    a = a.playSan('Rxb7');
+    expect(a.pockets.toString(), '[NNBBRQ]');
+    printBoard(a);
+    expect(a.outcome, Outcome.whiteWins);
+  });
+  test(
+      "Chess♯ - test that you can move a pinned piece, move into check, and the game ends when you take the opponent's king\n\t(score for taking a king is 10-0 or 0-10)",
+      () {
+    Position a = ChessSharp.initial;
```

```
+    a = a.playSan('K@a1');
+    a = a.playSan('K@g8');
+    a = a.playSan('B@h1');
+    a = a.playSan('B@h8');
+    a = a.playSan('B@g1');
+    a = a.playSan('g6');
+    // moves pinned pawn, exposes king to being taken
+    a = a.playSan('b3');
+    a = a.playSan('B@f8');
+    // moves in check
+    a = a.playSan('Kb2');
+    // takes king
+    a = a.playSan('Bxb2');
+    printBoard(a);
+    conditionalPrint(a.outcome);
+    conditionalPrint(Outcome.toPgnStringChessSharp(a.outcome));
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '0-10');
+  });
+  test('Chess♯ - test dropping a king into check', () {
+    Position a = ChessSharp.initial;
+    a = a.playSan('K@a1');
+    a = a.playSan('N@h8');
+    a = a.playSan('B@h1');
+    a = a.playSan('N@g8');
+    a = a.playSan('g3');
+    a = a.playSan('B@f8');
+    a = a.playSan('Bxb7');
+    a = a.playSan('K@a8');
+    a = a.playSan('Bxa8');
+    printBoard(a);
+    conditionalPrint(a.outcome);
+    conditionalPrint(Outcome.toPgnStringChessSharp(a.outcome));
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '10-0');
+  });
+  test('Chess♯ - test that there is no castling', () {
+    Position a = ChessSharp.initial;
+    a = a.playSan('K@e1');
+    a = a.playSan('K@e8');
+    a = a.playSan('R@a1');
+    a = a.playSan('R@a8');
+    a = a.playSan('R@h1');
+    a = a.playSan('R@h8');
+    expect(() => a.playSan('O-O'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('O-O-O'), throwsA(const TypeMatcher<PlayError>()));
+    a = a.playSan('e3');
+    expect(() => a.playSan('O-O'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('O-O-O'), throwsA(const TypeMatcher<PlayError>()));
+    printBoard(a);
+  });
+  test('Chess♯ - test that pawns do not sprint', () {
+    Position a = ChessSharp.initial;
+    a = a.playSan('K@e1');
+    a = a.playSan('K@e8');
+    expect(() => a.playSan('a4'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('b4'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('c4'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('d4'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('e4'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('f4'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('g4'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('h4'), throwsA(const TypeMatcher<PlayError>()));
+    a = a.playSan('a3');
+    expect(() => a.playSan('a5'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('b5'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('c5'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('d5'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('e5'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('f5'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('g5'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('h5'), throwsA(const TypeMatcher<PlayError>()));
+    conditionalPrint('${humanReadableBoard(a.board)}\n');
+  });
+  test('Chess♯ - test that you cannot under-promote', () {
+    Position a = ChessSharp.initial;
+    a = a.playSan('K@e1');
+    a = a.playSan('K@e8');
+    a = a.playSan('a3');
+    a = a.playSan('b6');
+    a = a.playSan('a4');
+    a = a.playSan('b5');
+    a = a.playSan('axb5');
+    a = a.playSan('h6');
+    a = a.playSan('b6');
+    a = a.playSan('h5');
+    a = a.playSan('b7');
+    a = a.playSan('h4');
+    expect(() => a.playSan('b8=N'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('b8=B'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('b8=R'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('b8=P'), throwsA(const TypeMatcher<PlayError>()));
+    expect(() => a.playSan('b8=K'), throwsA(const TypeMatcher<PlayError>()));
+    a = a.playSan('b8=Q');
+    conditionalPrint('${humanReadableBoard(a.board)}\n');
+  });
+  test(
+      "Chess♯ stalemates - test and scoring\n\t(since it's legal to move into check, we made an alternate test to determine stalemate...\n\tthe score for stalemating your opponent is 8-2 or 2-8)",
+      () {
+    // stalemate: king can't move
+    Position a =
+        ChessSharp.fromSetup(Setup.parseFen('1r5k/8/8/8/8/8/7r/K7 w - - 0 1'));
+    printBoard(a);
+    expect(a.outcome?.endType, EndType.stalemate);
+    expect(a.outcome?.winner, Side.black);
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '2-8');
+
+    // no stalemate: there's a piece in my pocket, and somewhere to place it
+    a = ChessSharp.fromSetup(
+        Setup.parseFen('1r5k/8/8/8/8/8/7r/K7[Q] w - - 0 1'));
+    printBoard(a);
+    expect(a.outcome, null);
+
+    // stalemate: there's a piece in my pocket, but nowhere to place it
+    a = ChessSharp.fromSetup(
+        Setup.parseFen('4k3/8/8/8/8/8/7q/qqqnKnqq[NNBBRRQ] w - - 0 1'));
+    printBoard(a);
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '2-8');
+
+    // no stalemate: a pawn can promote
+    a = ChessSharp.fromSetup(
+        Setup.parseFen('1r5k/3P4/8/8/8/8/7r/K7 w - - 0 1'));
+    printBoard(a);
+    expect(a.outcome, null);
+
+    // stalemate: the pawn is pinned
+    a = ChessSharp.fromSetup(Setup.parseFen('6rk/KP5r/7q/8/8/8/8/8 w - - 0 1'));
+    printBoard(a);
+    expect(a.outcome?.endType, EndType.stalemate);
+    expect(a.outcome?.winner, Side.black);
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '2-8');
+
+    // no stalemate: I'm in check (and can get out of it)
+    a = ChessSharp.fromSetup(
+        Setup.parseFen('K5rk/1P5r/1q6/8/8/8/8/8 w - - 0 1'));
+    printBoard(a);
+    expect(a.outcome, null);
+
```

```
+    // no stalemate: I'm in check (and can't get out of it)
+    a = ChessSharp.fromSetup(Setup.parseFen('K5rk/7r/8/8/8/8/8/8 w - - 0 1'));
+    printBoard(a);
+    expect(a.outcome, null);
+  });
+  test(
+      'Chess♯ impasses - test and scoring\n\t(games that go 50 moves without a capture or pawn-move\n\tthe score for an impasse is 7-3 or 3-7 if one side has more material\n\tor 4-6 if both sides have the same material)',
+      () {
+    // fen with 100 half-moves made
+    Position a = ChessSharp.fromSetup(
+        Setup.parseFen('4nnbk/8/8/8/8/8/8/KBNN4 w - - 100 100'));
+    printBoard(a);
+    expect(a.outcome?.endType, EndType.impasse);
+    expect(a.outcome?.winner, Side.black);
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '4-6');
+
+    // fen with 99 half-moves made (move last one manually)
+    // trigger the impasse while walking into check
+    a = ChessSharp.fromSetup(
+        Setup.parseFen('4nnbk/8/8/8/8/8/8/KBNN4 w - - 99 100'));
+    printBoard(a);
+    expect(a.outcome, null);
+    a = a.playSan('Ka2');
+    expect(a.outcome?.endType, EndType.impasse);
+    expect(a.outcome?.winner, Side.black);
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '4-6');
+
+    // fen with 100 half-moves made, black more material
+    a = ChessSharp.fromSetup(
+        Setup.parseFen('4nnbk/8/8/8/8/8/8/KBN5 w - - 100 100'));
+    printBoard(a);
+    expect(a.outcome?.endType, EndType.impasseMoreMaterial);
+    expect(a.outcome?.winner, Side.black);
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '3-7');
+
+    // fen with 100 half-moves made, white more material
+    a = ChessSharp.fromSetup(
+        Setup.parseFen('5nbk/8/8/8/8/8/8/KBNN4 w - - 100 100'));
+    printBoard(a);
+    expect(a.outcome?.endType, EndType.impasseMoreMaterial);
+    expect(a.outcome?.winner, Side.white);
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '7-3');
+
+    // fen with 100 half-moves made, white more material, but black has a queen in the pocket
+    // (pieces in the pocket don't count for material points)
+    a = ChessSharp.fromSetup(
+        Setup.parseFen('5nbk/8/8/8/8/8/8/KBNN4[q] w - - 100 100'));
+    printBoard(a);
+    expect(a.outcome?.endType, EndType.impasseMoreMaterial);
+    expect(a.outcome?.winner, Side.white);
+    expect(Outcome.toPgnStringChessSharp(a.outcome), '7-3');
+  });
+  test('Chess♯ - test pgn & fen import/export', () {
+    void testPGN(String pgnFile, String finalFen) {
+      conditionalPrint('\n\n${pgnFile.replaceAll('./data/', '')}\n');
+      // See if the PGN we generate is the same PGN as the original
+      final String pgn =
+          File(pgnFile).readAsStringSync().replaceAll('\r\n', '\n');
+      final game = PgnGame.parsePgn(pgn);
+      expect(game.makePgn(), pgn);
+
+      // Play the moves to see if any are illegal
+      int queenDropCount = 0;
+      Position position = PgnGame.startingPosition(game.headers);
+      for (final node in game.moves.mainline()) {
+        final move = position.parseSan(node.san);
+        if (move == null) break; // Illegal move
+        position = position.play(move);
+        if (node.san.toUpperCase().contains('Q@')) {
+          queenDropCount++;
+          if (queenDropCount == 2) {
+            // Print board position and FEN after the 2nd queen has been dropped (so we can copy/paste it into Stockfish)
+            printBoard(position);
+            conditionalPrint('Analyzable: ${position.fen}\n');
+          }
+        }
+      }
+
+      // Test final fen
+      expect(position.fen, finalFen);
+
+      printBoard(position);
+      conditionalPrint('Final: ${position.fen}\n${game.headers['Result']}');
+    }
+
+    testPGN(
+        './data/chess-sharp-david-john.pgn', '8/8/p7/8/8/8/2k5/8 w - - 0 71');
+
+    testPGN('./data/chess-sharp-caleb-david.pgn',
+        'kbb3r1/ppp4r/4pp2/8/P1N1PB2/1BP1R1pq/1P3P1P/3Q3K w - - 0 31');
+
+    testPGN('./data/chess-sharp-jim-david.pgn',
+        '8/3k4/bp2p3/4p3/PBRrP3/1P6/7P/5K2 w - - 6 53');
+
+    testPGN('./data/chess-sharp-jim-david2.pgn',
+        '3r3k/6p1/p1q3bp/8/5p2/2P2P2/PP1KQ3/7R w - - 1 42');
+  });
+ }
```

164 ■■■■■ test/db_testing_lib.dart

☐ Viewed

...

@@ -0,0 +1,164 @@

```
+ import 'package:dartchess/dartchess.dart';
+
+ const verbosePrinting = false;
+ bool printedNotice = false;
+
+ void conditionalPrint(Object? a) {
+   if (!printedNotice) {
+     printedNotice = true;
+     print('=' * 60);
+     print('${'=\tVERBOSE PRINTING'.padRight(53)}=');
+     if (verbosePrinting) {
+       print('${'=\tverbosePrinting is ON'.padRight(53)}=');
+       print('${'=\t'.padRight(53)}=');
+       print('${'=\tTo disable, set the "verbosePrinting"'.padRight(53)}=');
+       print('${'=\tconstant to false'.padRight(53)}=');
+     } else {
+       print('${'=\tverbosePrinting is OFF'.padRight(53)}=');
+       print('${'=\t'.padRight(53)}=');
+       print(
+           '${'=\tSet the "verbosePrinting" constant to true to help'.padRight(53)}=');
+       print('${'=\twith debugging these tests'.padRight(53)}=');
+     }
+     print('${'=\t(line 3 of \\test\\db_testing_lib.dart)'.padRight(53)}=');
+     print('=' * 60);
+   }
+   if (verbosePrinting) {
+     print(a);
+   }
+ }
+
+ enum DropZone { none, whiteHomeRow, blackHomeRow, anywhere }
+
+ const noDrops = <int>[];
+ const whiteHomeRow = <int>[
+   0,
+   1,
+   2,
+   3,
```

```diff
+   4,
+   5,
+   6,
+   7,
+ ];
+ const blackHomeRow = <int>[
+   56,
+   57,
+   58,
+   59,
+   60,
+   61,
+   62,
+   63,
+ ];
+
+ class MyExpectations {
+   final int legalDrops;
+   final int legalMoves;
+   final DropZone legalDropZone;
+   final List<Role> rolesThatCanDrop;
+   final List<Role> rolesThatCantDrop;
+
+   const MyExpectations(
+       {required this.legalMoves,
+       required this.legalDrops,
+       required this.legalDropZone,
+       required this.rolesThatCanDrop,
+       required this.rolesThatCantDrop});
+
+   String testLegalMoves(List<Move> a) {
+     if (a.whereType<NormalMove>().length != legalMoves) {
+       return 'Expected $legalMoves legal moves, got ${a.whereType<NormalMove>().length}';
+     }
+     if (a.whereType<DropMove>().length != legalDrops) {
+       return 'Expected $legalDrops legal drops, got ${a.whereType<DropMove>().length}';
+     }
+     for (final move in a) {
+       if (move is DropMove) {
+         if (rolesThatCantDrop.contains(move.role)) {
+           return '${move.role} is listed in rolesThatCantDrop';
+         }
+         if (!rolesThatCanDrop.contains(move.role)) {
+           return '${move.role} is not listed in rolesThatCanDrop';
+         }
+         if (legalDropZone == DropZone.anywhere &&
+             move.role == Role.pawn &&
+             (whiteHomeRow.contains(move.to) ||
+                 blackHomeRow.contains(move.to))) {
+           return 'Drop zone is anywhere, but a pawn cannot be dropped in rows 1 or 8';
+         } else if (legalDropZone == DropZone.whiteHomeRow &&
+             !whiteHomeRow.contains(move.to)) {
+           return 'Drop zone is whiteHomeRow, but ${move.to} is not in whiteHomeRow';
+         } else if (legalDropZone == DropZone.blackHomeRow &&
+             !blackHomeRow.contains(move.to)) {
+           return 'Drop zone is blackHomeRow, but ${move.to} is not in whiteHomeRow';
+         } else if (legalDropZone == DropZone.none &&
+             !noDrops.contains(move.to)) {
+           return 'Drop zone is none, but ${move.to} is not in noDrops';
+         }
+       }
+     }
+     return '';
+   }
+ }
+
+ List<DropMove> dropTestEachSquare(Position position) {
+   final legalDrops = <DropMove>[];
+   final allRoles = <Role>[
+     Role.pawn,
+     Role.knight,
+     Role.bishop,
+     Role.rook,
+     Role.queen,
+     Role.king
+   ];
+   for (final pieceRole in allRoles) {
+     for (int a = 0; a < 64; a++) {
+       if (position.isLegal(DropMove(role: pieceRole, to: a))) {
+         legalDrops.add(DropMove(role: pieceRole, to: a));
+       }
+     }
+   }
+   return legalDrops;
+ }
+
+ List<NormalMove> moveTestEachSquare(Position position) {
+   final legalMoves = <NormalMove>[];
+   for (int a = 0; a < 64; a++) {
+     for (int b = 0; b < 64; b++) {
+       if (position.isLegal(NormalMove(from: a, to: b))) {
+         legalMoves.add(NormalMove(from: a, to: b));
+       }
+     }
+   }
+   return legalMoves;
+ }
+
+ List<Move> printBoard(Position a, {bool printLegalMoves = false}) {
+   final z = StringBuffer();
+   final y = StringBuffer();
+   String x = '';
+   int moves = 0;
+   int drops = 0;
+
+   final legalMoves = <Move>[];
+   if (printLegalMoves) {
+     for (final move in moveTestEachSquare(a)) {
+       z.write('${move.uci}, ');
+       legalMoves.add(move);
+       moves++;
+     }
+     x = 'Legal Moves ($moves):\n$z\n';
+     for (final drop in dropTestEachSquare(a)) {
+       y.write('${drop.uci}, ');
+       legalMoves.add(drop);
+       drops++;
+     }
+     x += 'Legal Drops ($drops):\n$y';
+   }
+
+   conditionalPrint('${humanReadableBoard(a.board, a.pockets)}$x');
+   conditionalPrint(
+       '------------------------------------------------------------');
+   return legalMoves;
+ }
```

4 test/setup_test.dart

☐ Viewed

```diff
@@ -89,8 +89,8 @@ void main() {
     for (final fen in [
       '8/8/8/8/8/8/8/8 w - - 1+2 12 42',
       '8/8/8/8/8/8/8/8[Q] b - - 0 1',
-      'r3k2r/8/8/8/8/8/8/R3K2R[] w Qkq - 0 1',
-      'r3kb1r/p1pN1ppp/2p1p3/8/2Pn4/3Q4/PP3PPP/R1B2q~K1[] w kq - 0 1',
+      'r3k2r/8/8/8/8/8/8/R3K2R w Qkq - 0 1',
+      'r3kb1r/p1pN1ppp/2p1p3/8/2Pn4/3Q4/PP3PPP/R1B2q~K1 w kq - 0 1',
       'rQ~q1kb1r/pp2pppp/2p5/8/3P1Bb1/4PN2/PPP3PP/R2QKB1R[NNpn] b KQkq - 0 9',
       'rnb1kbnr/ppp1pppp/2Pp2PP/1P3PPP/PPP1PPPP/PPP1PPPP/PPP1PPP1/PPPqPP2 w kq - 0 1',
       '5b1r/1p5p/4ppp1/4Bn2/1PPP1PP1/4P2P/3k4/4K2R w K - 1 1',
```

© 2023 GitHub, Inc. Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About