ChessV 2.1 Source Code Documentation

Table of Contents

I. Coding Style

A. Naming

- Names of variables, functions, etc. are longer than in many projects – being descriptive is considered more important than saving a few keystrokes, especially given Visual Studio's Intellisense and auto-complete.
- Names of variables, functions, etc. begin with an initial capital (e.g., MakeMove) if they are public or internal and with an initial lower-case letter (e.g., helperFunction)

if they are protected or private. I realize this is not common for C# projects, and probably not common in general, but I like it and consider it valuable.

B. Collapsible Text

- Significant use is made of the #region and #endregion directives that create collapsible regions of text. Everything is collapsed by default. This makes the code much easier to navigate provided you are using Microsoft Visual Studio, or another compliant text editor that supports code folding. If not, these extra directives will make things look worse, but Visual C# Community Edition is free, and, if you're on another operating system, MonoDevelop supports it also.

II. Sub-Projects

A. ChessV.Base (namespace ChessV)

This contains the base framework for ChessV on which everything else relies. It contains all the essential architecture elements for defining and playing different games. It also contains the code for automating external XBoard engines, and running and adjudicating games. Compiles to ChessV.Base.dll

B. ChessV.Manager

This is a slightly higher-level framework which contains elements for keeping track of all defined chess variants and all configured XBoard engines. It knows which engines support which variants, and is used to initiate new games, including loading saved games. Compiles to ChessV.Manager.dll

C. ChessV.Games

This defines the basic library of games (chess variants) and their supporting rules that ChessV internally understands. It gets compiled to a DLL that is dynamically loaded when the program starts and all the game definitions contained there-in are automatically discovered. In other words, ChessV.Games is completely independent. There is no hard link between it and anything else in ChessV. ChessV.Games could be augmented or replaced entirely and none of the other sub-projects composing ChessV would either know or care. Compiles to ChessV.Games.dll

D. ChessV.GUI

This is the main program. The entire graphic user interface for ChessV is contained here. Compiles to ChessV.exe

E. ChessV.Compiler

This project contains everything related to parsing, compiling, and interpreting "ChessV Code", a simple scripting language for adding support for new games. At present, this is easy to use but allows only limited functionality. Compiles to ChessV.Compiler.dll

F. ChessV.Test

This project contains unit tests for validating the integrity of the ChessV framework to help prevent regressions. Although unit testing is an important part of developing complex, reliable software, at present there are very few unit tests.

G. ChessV.Utilities

This contains utilities for chess-variant related things not directly related to the main program. There isn't much here yet except for a utility for generating mobility reports for various piece types on boards of various sizes. I anticipate adding a lot here, though, such as code for generating and compiling opening books.

III. Architecture - Primary Classes

A. Board

Contains all information related to the nature of the board itself (the size, geometry, and square naming convention) as well as storing a position in a game in progress (locations of the pieces on the board.)

The Board class knows/handles all of the following:

- The board geometry (e.g., 2-dimensonal rectangular)
- The board size (number of ranks, number of files)
- The notation for files/ranks/squares (games can customize this)
- Translating each player's equivalent of a given rank/file/square based on the game's Symmetry (which is also customizable)
- Mapping of Pieces to squares
- The Hash Code of the current position (for repetition detection, transposition lookup, etc.)
- Looking up the next square from any given square and direction
- IsSquareAttacked function, which determines if a given square is currently attacked by any piece of a given player
- Some positional evaluation scores such at the total material per player as well as player material plus piece-square-tables for both midgame and endgame scores

Although most games use the Board class directly, it is extensible and some games use Board-derived classes (for example, Pocket Knight uses Board-derived class BoardWithPockets which provides the pocket squares.)

B. Game

This is the base class from which the classes that define specific chess variants are derived. The definition of the class itself defines the rules of the variant, whereas instances of these classes represent actual games in progress. The Game references the Board object representing the current position and the Board initializes many things based on the Game. These two classes are closely related.

The Game class contains a number of overridable virtual functions that handle the definition of the game rules and the initialization of everything related to the game, from the Board or Board-derived class that represents the current position, to the types of pieces present in the game, to the FEN format for representing positions in the game. New chess variants are defined by deriving a new class either directly from Game, or, more commonly, from the class of another variant that is similar. Then virtual functions are overridden to customize it.

The Game also contains a collection of Rule objects. Rules (described below) also have a number of virtual functions that can be overridden to handle different events. Separating things from the Game class and into Rule classes allows different game rules (such as Checkmate, Castling, and En Passant) to be packaged into Rule objects that can be "plugged in" to different games in a fairly ad hoc way allowing new chess variants to be defined fairly easily.

The architecture of ChessV that allows for the play of a potentially unlimited number of chess variants is based on a complicated network of objects (Games, Rules, Pieces, etc.) and messages (function calls) that are routed between them. The Game object is the central nexus that handles everything.

The Game class knows/handles all of the following:

- General bibliographic information about the game (name, inventor, year of invention, etc.)
- The Symmetry which defines how positions and directions are translated between players. Examples are mirror symmetry, rotational symmetry, and no symmetry (i.e., no translation.)
- The format of the FEN (Forsyth–Edwards Notation) for the given game, the FEN of the starting position, and functions for identifying, initializing, and extracting parts of the FEN

- All GameVariables for the given game – custom configuration parameters Game-derived classes can define to allow behavior to be altered either by derived classes or by the user (for example – the selection of the player's armies in Chess With Different Armies.) Functionality is provided for dynamically identifying all GameVariables and their types, and manipulating their values.
- The Players of the game – be they human, ChessV's internal engine, or an external Xboard-protocol engine
- The types of pieces (PieceType objects) that are used in the game
- The history of a game in progress – the list of moves have been made up to this point and the hash keys for all previous positions
- All the nasty internals of ChessV's internal engine which plays against you are also contained here. This includes information about the PV, killer moves, search stack, and so on. While all the functions that constitute the engine AI are unfortunately contained within this class, they are at least separated into a different source file (Search.cs rather than Game.cs).
- Overrideable virtual functions to handle events such as those listed below:
    - CreateBoard
    - SetGameVariables
    - AddRules
    - AddPieceTypes
    - ExpandVariablesInFEN
    - ChangePieceNames
- The collection of all Rule objects the game uses
- Routing of messages to Rule objects, such as the following messages:
    - MoveBeingGenerated
    - MoveBeingMade
    - MoveBeingUnmade
    - TestForWinLossDraw
    - NoMovesResult
- The list of all Directions that are used. This is based on the directions of movement of the PieceTypes that are used in the given game.

C. Rule

As previously mentioned, Rules provide a way to code behaviors that are then easily shared across Games. A Game contains a collection of Rule objects. The Rule is a collection of virtual functions that are called at certain times such as when moves are generated or played. At these times, the Game goes through its list of Rules calling the appropriate virtual functions. These functions can return event codes that affect the Game. For example, when a move is being made, the Game calls the MoveBeingMade function for all its Rules. The CheckmateRule overrides this function and checks to see if the king of the player who is moving is attacked. If so, the rule notifies the game that this move is illegal, preventing a player from moving a king into check or leaving it in check. Most chess variants are combinations of rules used in other games. The Rule object provides a convenient package for codifying these rules and plugging them into games.

D. PieceType

The PieceType determines the properties of a given type of piece (e.g., King, Queen, Pawn, etc.) The PieceType knows the following things:

- The name and notation used for the type of piece in the given game

- The movement capabilities (defined in MoveCapability objects.) This determines a direction the piece can move, the maximum/minimum range in that direction, and other attributes such as whether or not it can capture when moving in this direction.
- The material value of the piece – both a midgame and endgame value. This is used by the internal engine for evaluating positions. Values are interpolated between the midgame and endgame values as the game progresses allowing pieces to become more or less valuable as the board clears out.
- The midgame and endgame piece-square-table values and the ability to automatically generate these tables based on the geometry of the board and other piece parameters (bonus for occupying the small center or the large center, bonus for each attach on the small center or large center, bonus for forwardness or position, etc.)
- The Zobrist hash keys for each player/square combination

IV. Defining Games

This section describes how the actual games that ChessV supports are defined and provides insight into how to define new games. Examples of the implementations of actual chess variants are provided in the following section.

A. Defining Piece Types

At the core of any game are the pieces, each of which is of a given PieceType. For the most part, creating a new PieceType is simply a matter of defining its movements.

Here is an example of code that defines the normal chess Queen:

```
public class Queen: PieceType
{
    public Queen( string name, string notation, int midgameValue, int endgameValue ):
        base( "Queen", name, notation, midgameValue, endgameValue )
    {
        Slide( new Direction(  0,  1 ) );
        Slide( new Direction(  0, -1 ) );
        Slide( new Direction(  1,  0 ) );
        Slide( new Direction( -1,  0 ) );
        Slide( new Direction(  1,  1 ) );
        Slide( new Direction(  1, -1 ) );
        Slide( new Direction( -1,  1 ) );
        Slide( new Direction( -1, -1 ) );
    }
}
```

The constructor takes the name and notation of the piece in the given game, as well as the midgame and endgame values. The base class takes these same parameters, as well the "internal" name of the piece (in this case "Queen".) The internal name is used as a fall-back for identifying which bitmap to use when rendering the piece.

This class contains only a constructor, and the constructor does nothing but define the movements. A Queen slides in any one of eight directions, and each Direction is simply a file offset and a rank offset.

For a slightly more interesting example, let's look at the Pawn:

```
public class Pawn: PieceType
{
    public Pawn( string name, string notation, int midgameValue, int endgameValue ):
        base( "Pawn", name, notation, midgameValue, endgameValue )
    {
        StepMoveOnly( new Direction( 1, 0 ) );
        StepCaptureOnly( new Direction( 1,  1 ) );
        StepCaptureOnly( new Direction( 1, -1 ) );

        PSTMidgameForwardness = 6;
        PSTEndgameForwardness = 12;
    }
}
```

The Pawn has a single step which allows only a move (not a capture) directly forward, and two steps which allows only a capture diagonally forward. The constructor also, in additional to defining the movement, also customizes the default piece-square-table (PST) handling. Specifically, it significantly increases the bonus for the piece "forwardness" giving it an increasing bonus for advancing forward. Note that evaluation parameters such as these are not necessary for ChessV to be able to play a game; they only affect how well the internal engine plays the game.

Notice that nothing here handles the initial double-move, en passant capture, or promotion. Those are all handled by special Rules for the Game since different games have different rules for these things. Here we define only the basic Pawn.

For defining movements, the following functions are provided:

- Step(Direction) – permits a single step in the given direction (which could be what would be considered a "leap" – the Knight has eight Steps with directions like Direction(1, 2).
- Slide(Direction) – permits a piece to slide in the given direction as far as desired or until an obstruction is reached.
- Slide(Direction, int maxSteps) – Similar to Slide above but with a maximum range
- StepMoveOnly(Direction) – Similar to Step but capture is not permitted. An example is the forward move of the Pawn.
- SlideMoveOnly(Direction) – Similar to Slide above but capture is not permitted
- StepCaptureOnly(Direction) – Similar to Step except the step is only permitted if an enemy piece is captured. An example is the diagonal capture move of the Pawn.
- SlideCaptureOnly(Direction) – Similar to Slide above but the move is only permitted if an enemy piece is captured.
- CannonMove(Direction) – The sliding move of the Cannon piece from XiangQi (Chinese Chess.) The piece slides in the specified direction without capturing. To capture, must leap over exactly one piece.
- AddMoveCapability(MoveCapability) – For the definition of more complicated moves, you can construct a MoveCapability object yourself and pass it in. For example, a sliding move that has both a minimum or maximum number of steps, or can only be made when the piece is on certain squares.

Note that the actual definitions of these pieces is slightly more complicated to provide better code re-use, but these would work fine and are sufficient for now. We'll delve deeper later.

B. Simplest definition of the game of orthodox Chess

First we'll investigate the easiest way to define the game of orthodox Chess. Please note that, like the piece types above, what is shown in this section is not the way ChessV actually defines it. The Chess class in ChessV is more complicated than this for the purpose of making it easier to use it as a base class for defining new games. We will look at the actual definintion of Chess in the next section but first let's start simple.

The following code will accomplish the full definition of Chess with all rules:

```
[Game("Chess", typeof(Geometry.Rectangular), 8, 8,
      InventedBy = "Unknown",
      Invented = "circa 8th century",
      Tags = "Chess Variant,Historic,Regional,Popular",
      GameDescription1 = "Exact origins unknown",
      GameDescription2 = "Believed to be from India around the 6th century AD",
      XBoardName = "normal")]
public class Chess: Game
{
    // *** PIECE TYPES *** //

    public PieceType Queen;
    public PieceType Rook;
    public PieceType Bishop;
```

```
    public PieceType Knight;
    public PieceType Pawn;
    [Royal] public PieceType King;


    // *** CONSTRUCTION *** //

    public Chess():
        base
            ( /* NumPlayers = */ 2,
              /* NumFiles = */ 8,
              /* NumRanks = */ 8,
              /* symmetry = */ new MirrorSymmetry() )
    {
    }


    // *** INITIALIZATION *** //

    public override void SetGameVariables()
    {
        FENFormat = "{array} {current player} {castling} {en-passant} {half-move
            clock} {move number}";
        FENStart = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";
    }

    public override void AddPieceTypes()
    {
        AddPieceType( King = new King( "King", "K", 0, 0 ) );
        AddPieceType( Pawn = new Pawn( "Pawn", "P", 100, 125 ) );
        AddPieceType( Rook = new Rook( "Rook", "R", 500, 550 ) );
        AddPieceType( Bishop = new Bishop( "Bishop", "B", 325, 350 ) );
        AddPieceType( Knight = new Knight( "Knight", "N", 325, 325 ) );
        AddPieceType( Queen = new Queen( "Queen", "Q", 950, 1000 ) );
    }

    public override void AddRules()
    {
        // *** CASTLING *** //
        CastlingRule();
        CastlingMove( 0, "e1", "g1", "h1", "f1", 'K' );
        CastlingMove( 0, "e1", "c1", "a1", "d1", 'Q' );
        CastlingMove( 1, "e8", "g8", "h8", "f8", 'k' );
        CastlingMove( 1, "e8", "c8", "a8", "d8", 'q' );

        // *** PAWN DOUBLE MOVE *** //
        MoveCapability doubleMove = new MoveCapability();
        doubleMove.Direction = new Direction( 1, 0 );
        doubleMove.MinSteps = 2;
        doubleMove.MaxSteps = 2;
        doubleMove.CanCapture = false;
        doubleMove.Condition = location => location.Rank == 1;
        Pawn.AddMoveCapability( doubleMove );

        // *** EN-PASSANT *** //
        EnPassantRule( Pawn, GetDirectionNumber( new Direction( 1, 0 ) ) );

        // *** PAWN PROMOTION *** //
        List<PieceType> availablePromotionTypes = new List<PieceType>() { Queen,
            Rook, Bishop, Knight };
        BasicPromotionRule( Pawn, availablePromotionTypes, loc => loc.Rank == 7 );

        // *** FIFTY-MOVE RULE *** //
        AddRule( new Rules.Move50Rule( Pawn ) );

        // *** DRAW-BY-REPETITION RULE *** //
        AddRule( new Rules.RepetitionDrawRule() );
    }
}
```

We'll take it section by section.

```
[Game("Chess", typeof(Geometry.Rectangular), 8, 8,
      InventedBy = "Unknown",
      Invented = "circa 8th century",
      Tags = "Chess Variant,Historic,Regional,Popular",
      GameDescription1 = "Exact origins unknown",
      GameDescription2 = "Believed to be from India around the 6th century AD",
      XBoardName = "normal")]
```

First, we must apply a C# attribute of type GameAttribute to the Game-derived class to specify some basic information about the game. Just this attribute accomplishes the purpose of making the game available within ChessV automatically. When the application starts, it looks for all these attributes dynamically and enters them into the GameCatalog. Any class that is linked at runtime that is decorated with the GameAttribute will automatically be available, listed in the master index, and playable.

The first parameters are required:

- The name of the game
- The type of the board geometry
- A variable number of integers that define the size of the board according to the specified geometry. For standard 2D-Rectangular boards, we provide first the number of files and then the number of ranks.

The rest of the parameters are identified by name and are all optional:

- InventedBy – the name of the inventor or inventors
- Invented – when the game was invented
- Tags – a list of the tags that should be associated with this game (for searching)
- GameDescription1 and GameDescription2 – two lines of brief description
- XBoardName – designation of the game in the XBoard protocol "variants" option
- Definitions (not shown here) – can define GameVariable values for the given game. This will be demonstrated later.

Next we have member variables that will hold our PieceType objects. We'll initialize these later:

```
public PieceType Queen;
public PieceType Rook;
public PieceType Bishop;
public PieceType Knight;
public PieceType Pawn;
[Royal] public PieceType King;
```

Notice the [Royal] attribute before the definition of the King object. The King PieceType doesn't describe anything about check/checkmate, just as the Pawn doesn't describe promotion or double move. The King PieceType is simply a piece that can make a Step in each of the eight basic directions. But tagging the King object with the [Royal] attribute tells the engine that it's a royal piece, and that activates check, checkmate, and stalemate. That one word in brackets is all it takes.

Then we have the constructor:

```
public Chess():
    base
        ( /* NumPlayers = */ 2,
          /* NumFiles = */ 8,
          /* NumRanks = */ 8,
          /* symmetry = */ new MirrorSymmetry() )
{
}
```

The constructor takes no parameters, but must pass some parameters to the constructor of the base class (Game): the number of players, the number of files, the number of ranks, and the type of Symmetry.

The remainder of the code overrides a few virtual functions that are each called once when a new object of any Game-derived class is being created. This is where everything else about the game is configured.

```
public override void SetGameVariables()
{
    FENFormat = "{array} {current player} {castling} {en-passant} {half-move clock}
        {move number}";
```

```
    FENStart = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";
}
```

The SetGameVariables virtual function is called very early in the initialization process. To make games customizable beyond the hard-coded aspects of the class, you can define GameVariables and then use this function to set their default values (which derived classes can then change.) Games in ChessV make extensive use of GameVariables to make game definitions flexible and maximize code reuse. For now, though, there are only two game variables defined in the Game base class that *must* be set. We need to define the format of the FEN string, (which is just a list of the sections within curly-braces), and we need to define the FEN string of the starting position of a new game. Other parts of the architecture know how to access/use the various parts of the string. This will be described in more detail later.

```
public override void AddPieceTypes()
{
    AddPieceType( King = new King( "King", "K", 0, 0 ) );
    AddPieceType( Pawn = new Pawn( "Pawn", "P", 100, 125 ) );
    AddPieceType( Rook = new Rook( "Rook", "R", 500, 550 ) );
    AddPieceType( Bishop = new Bishop( "Bishop", "B", 325, 350 ) );
    AddPieceType( Knight = new Knight( "Knight", "N", 325, 325 ) );
    AddPieceType( Queen = new Queen( "Queen", "Q", 900, 1000 ) );
}
```

The AddPieceTypes virtual function is where the PieceTypes that we declared earlier are initialized. We construct a new object of each type giving it the name, notation, midgame value and endgame value, and then pass it to the AddPieceType function, defined in the Game class, to register the piece type.

Finally, the AddRules function handles the setting up of all the special rules of the game. This is where most of the interesting things happen. We'll look at each chunk which defines a particular rule in turn.

```
// *** CASTLING *** //
CastlingRule();
CastlingMove( 0, "e1", "g1", "h1", "f1", 'K' );
CastlingMove( 0, "e1", "c1", "a1", "d1", 'Q' );
CastlingMove( 1, "e8", "g8", "h8", "f8", 'k' );
CastlingMove( 1, "e8", "c8", "a8", "d8", 'q' );
```

Here we set up the rules for castling. To enable castling, you first call the CastlingRule() function of the Game class. This will create a new CastlingRule object (derived from Rule) and enter it into the list of rules. The next four CastlingMove() lines add the various castling options. The CastlingMove function takes six parameters:

- The player number (starting at zero)
- The notation of the square the first piece (typically the King) moves from
- The notation of the square the first piece moves to
- The notation of the square the other piece moves from
- The notation of the square the other piece moves to
- The character within the {castling} section of the FEN that designates whether this move is still available.

```
// *** PAWN DOUBLE MOVE *** //
MoveCapability doubleMove = new MoveCapability();
doubleMove.Direction = new Direction( 1, 0 );
doubleMove.MinSteps = 2;
doubleMove.MaxSteps = 2;
doubleMove.CanCapture = false;
doubleMove.Condition = location => location.Rank == 1;
Pawn.AddMoveCapability( doubleMove );
```

This sets up the pawn's initial double move, and is the most complicated part of the whole class. The Pawn PieceType class already has three MoveCapabilities defined. To add the double-space move we create and configure a new MoveCapability object and add it to the Pawn object.

This move has the same Direction as the single-space move – a file offset of 0 and a rank offset of 1. It is a limited slide, having a maximum range of 2 steps. It also has a minimum range of 2 steps. Why? Because the Pawn already has a move defined that allows a single step forward. We do not what to add another move that does the same thing because that would be redundant. So the slide we are defining here is exactly two steps. We set CanCapture to false, since this move cannot capture. Finally, we can set the Condition parameter of the MoveCapability to specify a condition under which this move is allowed since Pawns can't make a double step any old time. The condition is that if the pawn is on the second rank then this move is allowed. Look at what we set the Condition to: `location => location.Rank == 1`. This is a lambda function that will be executed any time moves for a pawn are generated to determine if this move is allowed. The lambda function must take a Location and return a Boolean specifying if the move is allowed. Our function simply returns whether or not the Rank of the piece is 1. Notes: Ranks, like everything, are numbered starting at zero, so 1 is actually the second rank. Also, the location provided is automatically translated, based on the Symmetry of the game, for the piece's player, so this simple code works for both players. Pretty cool, huh?

```
// *** EN-PASSANT *** //
EnPassantRule( Pawn, GetDirectionNumber( new Direction( 1, 0 ) ) );
```

The EnPassantRule function of the Game class creates an EnPassantRule object and adds it to the list of rules. It takes two parameters – the type of piece that is susceptible to En Passant capture, and the direction of travel. The implementation of the rule is pretty flexible so any slide in that direction greater than one step is susceptible to En Passant capture by an enemy piece of the same type on any square passed over.

```
// *** PAWN PROMOTION *** //
List<PieceType> availablePromotionTypes = new List<PieceType>() { Queen, Rook, Bishop, Knight };
BasicPromotionRule( Pawn, availablePromotionTypes, loc => loc.Rank == 7 );
```

The BasicPromotionRule function takes three arguments – the type that promotes, a list of types to which it may promote, and a lambda function determining the locations upon which it promotes. This is just like the lambda function we saw above.

```
// *** FIFTY-MOVE RULE *** //
AddRule( new Rules.Move50Rule( Pawn ) );
```

Activate the 50-move rule. We call the Game class's AddRule function and pass a new Move50Rule object. It takes one parameter, the type of piece whose movement automatically resets the counter. (Any capture will also reset the counter.)

```
// *** DRAW-BY-REPETITION RULE *** //
AddRule( new Rules.RepetitionDrawRule() );
```

Activate the draw-by-repetition rule by calling AddRule and passing a new RepetitionDrawRule object.

That is all it takes to fully support the game of Chess. Of course, this class is only so small because we already have Rule objects defined for so many things, and the code for some of these is fairly complicated. But that's the point of the architecture – package the complicated things into Rule objects and make them as flexible and generic as possible so that Game-derived classes can easily be defined just by plugging them in.

In the next section we'll look at how Chess is actually implemented in ChessV.

C. Actual Definition of the Chess class

The actual definition of the Chess class is different than what is described above to allow for better code reuse – that is, as much as possible, allowing derived classes to define new variants without rewriting code. It is different from the simplified version shown previously in two important ways:

Firstly, GameVariables are defined that allow functionality to be disabled or configured by changing the variables in derived classes. For example, a bool GameVariable called PawnDoubleMove is defined that controls whether pawns have a two-space initial move. Games, such as Shatranj, that do not have that can still derive from existing classes and simply set PawnDoubleMove to false.

Secondly, the code is not all in the Chess class. There is a hierarchy of base classes that provide increasing functionality so that you can derive new games from the class most appropriate. Here is the hierarchy of Chess base classes and what they provide:

- GenericChess – This can be used for games with boards of any size that use Chess pawns and a royal king. It supports the functionality for pawn promotion. The GameVariable "Promotion" controls the type of promotion (standard, promote-by-replacement, custom, or none.) The GameVariable "PromotionTypes" is a string containing the notations of all the types to which promotion is allowed. By default promotion happens on the last rank. GenericChess also implements the BareKing rule, controlled by the boolean GameVariable "BareKing" (false by default.) This is the BareKing rule used in Shatranj.
- Generic__x8 – This class derives from GenericChess and is useful for games on boards with eight ranks (and any number of files.) It adds support for the pawn's double space move (enabled by GameVariable "PawnDoubleSpace") and the En Passant rule (enabled by "EnPassant".)
- Generic8x8 – This class derives from Generic__x8 and is useful for games on boards with eight ranks and eight files. It adds support for castling, which is controlled by a ChoiceVariable called Castling which can take the values "None", "Standard", "Long", "Flexible", or "Custom". The meanings of the values are as follows:
    - None – Castling is disabled
    - Standard – The King piece, which must start on either the D or E file, castles with the pieces in the corners by sliding two squares in either direction (under the usual restrictions.)
    - Long – The same as Standard but the King slides three spaces when castling long.
    - Flexible – The King can slide two or more spaces toward the pieces in the corners and the corner piece jumps over to occupy the adjacent square.
    - Custom – The Generic8x8 class will do nothing so derived classes must provide the custom functionality.
- Chess – The class that actually plays Chess and is derived from Generic8x8. When adding a new game sometimes it's easiest to derive from one of the classes higher up the chain, but for games close enough to Chess, it is advantageous to derive from this class directly.

V. Chess Variant Examples

Now let's go through some examples of how actual chess variants are defined. These examples will show how the architecture really shines.

A. Corridor Chess

We'll start with an incredibly simple variant to program – Corridor Chess by Tony Paletta, described on page 73 of Pritchard's Encyclopedia of Chess Variants. This is simply Chess with a different setup, and no castling, double-space pawn move, or En Passant.

Here's the code:

```
[Game("Corridor Chess", typeof(Geometry.Rectangular), 8, 8,
      InventedBy = "Tony Paletta",
      Invented = "1980",
      Tags = "Chess Variant",
      GameDescription1 = "Standard Chess with a different setup and no castling",
      GameDescription2 = "Players often fight for control of the outside files")]
class CorridorChess: Chess
{
    // *** INITIALIZATION *** //

    public override void SetGameVariables()
    {
        base.SetGameVariables();
        Array = "1nrqkrn1/2b2b2/1pppppp1/8/8/1PPPPPP1/2B2B2/1NRQKRN1";
```

```
        Castling.Value = "None";
    }


    // *** XBOARD ENGINE SUPPORT *** //

    public override EngineGameAdaptor TryCreateAdaptor( EngineConfiguration config )
    {
        if( config.SupportedVariants.Contains( "normal" ) &&
            config.SupportedFeatures.Contains( "setboard" ) )
        {
            EngineGameAdaptor adaptor = new EngineGameAdaptor( "normal" );
            adaptor.IssueSetboard = true;
            return adaptor;
        }
        return null;
    }
}
```

First, there's the GameAttribute that provides the essential metadata and bibliographical information for the game:

```
[Game("Corridor Chess", typeof(Geometry.Rectangular), 8, 8,
      Invented = "1980",
      InventedBy = "Tony Paletta",
      Tags = "Chess Variant",
      GameDescription1 = "Standard Chess with a different setup and no castling",
      GameDescription2 = "Players often fight for control of the outside files")]
```

Every game has this, so we won't bother mentioning it when describing future games unless there is something unusual about it.

Since the base class, Chess, provides every parameter needed for construction, we don't even need to write a constructor in this class. (C# will implicitly create one for us.)

The one and only thing we need in this class to play Corridor Chess is an override for the SetGameVariables function:

```
public override void SetGameVariables()
{
    base.SetGameVariables();
    Array = "1nrqkrn1/2b2b2/1pppppp1/8/8/1PPPPPP1/2B2B2/1NRQKRN1";
    Castling.Value = "None";
}
```

The Game class provides a number of virtual functions that can be overridden. The bulk of the work of adding support for new games involves overriding these functions. Think of them as "hooks" that you can either hook into by overriding them, (to change some behavior), or leave alone and allow the base classes to handle with default behavior. In a later section we will describe these functions in more detail.

As described earlier, we facilitate code reuse by providing base classes which not only provide behavior, but allow behavior to be customized by setting the values of game variables. When adding new games, if you provide new functionality, it is good programming practice to allow it to be controlled with game variables so that it can more easily be reused by derivative works.

In this implementation, we do three things. We call the base class's implementation of SetGameVariables. This is very important and will almost always be the first thing that is done. By doing so, we allow the underlying classes to set things up, and then we change what we need to change. The second thing we do is set the Array. This is the single most important line of code, defining the primary way in which Corridor Chess differs from orthodox Chess. Finally, we turn off castling. Castling is GameVariable of type Option, meaning it allows a fixed set of choices. (An exception will be thrown if it is set to an unsupported value.) The Castling GameVariable used here is provided by the Generic8x8 base class. Setting the value to "None" turns off castling. This may seem unnecessary, since the corner squares with which the king would castle are empty, but pieces can move there, so we'll prevent any potential trouble by disabling castling. On the other hand, we could also turn off the pawn's double move and En Passant by setting game variables too,

but since there are no pawns on the second rank, nor can they move there, this is not necessary (although there would be no harm in doing so.)

The last function in the class overrides another virtual function – TryCreateAdaptor. This is totally unnecessary for Corridor Chess to be available in ChessV or for its internal engine to play against you. The purpose of this code is to allow greater support for playing against external XBoard/WinBoard engines. ChessV dynamically detects the variants that XBoard protocol version 2 engines announce support for and offers them as options for those games. But ChessV also provides an "Engine Adaptor" architecture to allow dynamic determination of engines that, while not internally configured for a given game, can be made to play that game. Corridor Chess is a good example. Almost no engines are going to announce support for it, but almost all can actually play it, simply by sending a setboard command to place the pieces correctly. This code provides a very simple example:

```
public override EngineGameAdaptor TryCreateAdaptor( EngineConfiguration config )
{
    if( config.SupportedVariants.Contains( "normal" ) &&
        config.SupportedFeatures.Contains( "setboard" ) )
    {
        EngineGameAdaptor adaptor = new EngineGameAdaptor( "normal" );
        adaptor.IssueSetboard = true;
        return adaptor;
    }
    return null;
}
```

When a user requests to play a game, ChessV will offer each engine that announces support for that game as an option. For every other engine, it will pass that engine to the Game class's TryCreateAdaptor function, and, if it returns true, it will also offer that engine as an option, passing any communication through the EngineGameAdaptor object returned. Here we're only requiring two things for an engine to support this game:

- That the engine is able to play orthodox Chess (SupportedVariants contains "normal" which is how orthodox Chess is described in the XBoard protocol)
- That it supports the "setboard" feature

If it meets these two requirements, it can play Corridor Chess. We create and return an EngineGameAdaptor that is instructed to play variant "normal" and is told to issue a setboard command. More complicated adaptor functionality is available, and the EngineGameAdaptor object is itself extensible.

B. Knightmate

Knightmate, by Bruce Zimov, is a chess variant with one royal knight where the king usually stands, and two normal "kings" where the knights usually stand. Here's the code:

```
[Game("Knightmate", typeof(Geometry.Rectangular), 8, 8,
      XBoardName = "knightmate",
      InventedBy = "Bruce Zimov",
      Invented = "1972",
      Tags = "Chess Variant,Popular",
      GameDescription1 = "Player has two Kings where the Knights usually are",
      GameDescription2 = "and a royal Knight where the King usually is")]
[Appearance(ColorScheme="Cinnamon")]
public class Knightmate: Chess
{
    // *** INITIALIZATION *** //

    public override void SetGameVariables()
    {
        base.SetGameVariables();
        Array = "rkbqnbkr/pppppppp/8/8/8/8/PPPPPPPP/RKBQNBKR";
    }

    public override void AddPieceTypes()
    {
        base.AddPieceTypes();
        castlingType = Knight;
        Knight.MidgameValue = Knight.EndgameValue = 0;
        King.MidgameValue = King.EndgameValue = 325;
    }
```

```
    public override void AddRules()
    {
        base.AddRules();
        // get rid of the Checkmate Rule
        RemoveRule( typeof(Rules.CheckmateRule) );
        // add the new Checkmate rule
        AddRule( new CheckmateRule( Knight ) );
    }
}
```

With this example, we have a new attribute that decorates the game class:

```
[Appearance(ColorScheme="Cinnamon")]
```

The Appearance attribute is optional, but can be used to specify information about the game's default appearance if desired. In this case, we specify that the default color scheme should be Cinnamon. The Appearance attribute can also specify default piece sets and the number of square colors (such as 1 for Shatranj or 3 for games with Knightriders such as Unicorn Great Chess.)

In this class, we override three virtual functions. First is SetGameVariables (nothing here we haven't seen before – we just set the Array):

```
public override void SetGameVariables()
{
    base.SetGameVariables();
    Array = "rkbqnbkr/pppppppp/8/8/8/8/PPPPPPPP/RKBQNBKR";
}
```

Then we override AddPieceTypes:

```
public override void AddPieceTypes()
{
    base.addPieceTypes();
    castlingType = Knight;
    Knight.MidgameValue = Knight.EndgameValue = 0;
    King.MidgameValue = King.EndgameValue = 325;
}
```

This virtual function is where the different types of pieces used in a game are added. In this case, we're not adding new types, just reconfiguring them. First, we call the base class's implementation of AddPieceTypes. Then we assign castlingType to the knight piece type so that the castling rule knows to use the knight for castling rather than the king. Although we are setting a game variable, we can't do this in SetGameVariables because Knight isn't defined until the call to the base class implementation of AddPieceTypes. Finally, we change the material evaluation parameters for the king and knight. Typically, the king has a value of 0 (although this value is arbitrary, since you can't lose your king anyway.) Here we give the knight midgame and endgame values of 0 and the king values of 325.

A word about piece values: pieces have both a "midgame" value and an "endgame" value. From the beginning through the midgame pieces use the midgame value. Then, as the board clears out, the values slide toward the endgame values. Steppers like the king and knight generally have the same value, but sliders get more powerful as the board clears out. Conversely, the cannon in Chinese Chess is an example of a piece that becomes less powerful as the board clears out.

The last override is AddRules:

```
public override void AddRules()
{
    base.AddRules();
    // get rid of the Checkmate Rule
    RemoveRule( typeof(Rules.CheckmateRule) );
    // add the new Checkmate rule
    AddRule( new CheckmateRule( Knight ) );
}
```

This is where we add the special rules the game requires. We call the base class implementation which sets everything up as in orthodox Chess. Then we remove the existing checkmate rule and add a new checkmate rule with the knight as the royal piece.

C. Extinction Chess and Kinglet Chess

Extinction Chess by R. Wayne Schmittberger and Kinglet Chess by V. R. Parton are very similar games. They are so similar, in fact, that we implement them both with a single class. Here it is:

```
[Game("Extinction Chess", typeof(Geometry.Rectangular), 8, 8,
      InventedBy = "R. Wayne Schmittberger",
      Invented = "1985",
      Tags = "Chess Variant,Popular",
      GameDescription1 = "A derivative of standard Chess where the objective",
      GameDescription2 = "is to capture all the pieces of any one type",
      Definitions = "ExtinctionTypes=KQRNBP;PromotionTypes=KQRNB")]
[Game("Kinglet Chess", typeof(Geometry.Rectangular), 8, 8,
      InventedBy = "V. R. Parton",
      Invented = "1953",
      Tags = "Chess Variant",
      GameDescription1 = "A derivative of standard Chess where the objective",
      GameDescription2 = "is to capture all a player's pawns",
      Definitions = "ExtinctionTypes=P;PromotionTypes=K")]
public class ExtinctionChess: Chess
{
    // *** GAME VARIABLES *** //

    [GameVariable] public string ExtinctionTypes { get; set; }


    // *** INITIALIZATION *** //

    public override void AddRules()
    {
        base.AddRules();
        //  Replace the Checkmate Rule with the Extinction Rule
        ReplaceRule( FindRule( typeof(Rules.CheckmateRule), true ),
            new Rules.Extinction.ExtinctionRule( ExtinctionTypes ) );
    }
}
```

The GameAttributes are worth mentioning because they demonstrate a couple of new things:

```
[Game("Extinction Chess", typeof(Geometry.Rectangular), 8, 8,
      Invented = "1985",
      InventedBy = "R. Wayne Schmittberger",
      Tags = "Chess Variant,Popular",
      GameDescription1 = "A derivative of standard Chess where the objective",
      GameDescription2 = "is to capture all the pieces of any one type",
      Definitions = "ExtinctionTypes=KQRNBP;PromotionTypes=KQRNB")]
[Game("Kinglet Chess", typeof(Geometry.Rectangular), 8, 8,
      Invented = "1953",
      InventedBy = "V. R. Parton",
      Tags = "Chess Variant",
      GameDescription1 = "A derivative of standard Chess where the objective",
      GameDescription2 = "is to capture all a player's pawns",
      Definitions = "ExtinctionTypes=P;PromotionTypes=K")]
```

For one thing, we have two different Game attributes on the same class. This registers the class in the game catalog with two different entries. It was possible to code these games in such a way that nothing at all in the ExtinctionChess class itself that is specific to either game. But they are not the same, so what produces the differences? That is the other new aspect to these Game attributes – the Definitions value. This is yet another place that we can set the values of game variables. This comes in handy so that we can support both games without requiring two different classes with different implementations of SetGameVariables. These two games are identical except for the types which trigger the extinction victory condition and the types to which pawns may promote, both of which are controlled with game variables.

The next thing to note is that in this class we actually define a new game variable:

```
[GameVariable] public string ExtinctionTypes { get; set; }
```

A game variable is just a public C# property decorated with the [GameVariable] attribute. This tells the ChessV architecture that it is a "game variable" – a configurable element that

affects the rules of game play. The classes that implement games have lots of member variables that they use for their own purposes and are not exposed to the architecture at large, but those marked [GameVariable] are special – this activates numerous things in the architecture that provide support for extensible game configuration. In this case, we define a string called ExtinctionTypes will store a list of the notations of the piece types which will cause a player to lose the game if they go extinct.

The only other thing we have here is the override to AddRules:

```
public override void AddRules()
{
    base.addRules();
    //  Replace the Checkmate Rule with the Extinction Rule
    ReplaceRule( FindRule( typeof(Rules.CheckmateRule), true ),
        new Rules.Extinction.ExtinctionRule( ExtinctionTypes ) );
}
```

We call the base class implementation, as usual, and then we use ReplaceRule to replace the game's CheckmateRule with an ExtinctionRule, who's constructor takes a string that lists the types to check for extinction.

That's it. This class adds support for two new games with almost no code. Of course, that is only possible because the new functionality is tucked away inside "ExtinctionRule", a new Rule class. To add support for Extinction Chess we needed to create a new Rule-derived class. But, that now having been created, we can plug it into any game henceforth to produce a piece-extinction victory condition.

Now let's take a look at how the ExtinctionRule was created. A Rule is basically a container for a collection of virtual functions that serve as "hooks" that the rule can override to respond to various events. In this example, the whole point of the rule is to override the TestForWinLossDraw function. Whenever the ChessV architecture needs to determine if a game has ended, it goes through all the Game class's Rules calling this function. The default behavior defined in the Rule base class is simply to return `MoveEventResponse.NotHandled`, which means the rule isn't doing anything. Here is the TestForWinLossDraw function for the ExtinctionRule:

```
public override MoveEventResponse TestForWinLossDraw( int currentPlayer, int ply )
{
    foreach( int typeNumber in extinctionTypeNumbers )
        if( Board.GetPieceTypeBitboard( currentPlayer, typeNumber ).BitCount == 0 )
            return MoveEventResponse.GameLost;
    return MoveEventResponse.NotHandled;
}
```

We loop through each type that can't go extinct. For each, we check to see if the current player is out of pieces of that type. If so, we return `MoveEventResponse.GameLost`.

The implementation is simple, but to understand it, we must introduce the concept of a bitboard. This is a common technique used by Chess programs. A bitboard is typically stored as an integer, but is interpreted as a collection of bits. Each bit stores one boolean value for each square of the board. Since Chess has 64 squares, and almost all modern environments efficiently handle 64-bit integers, it works out really well. Programs store bitboards for lots of things: every white piece, every bishop, etc. And logical operations are almost instantaneous. Given a bitboard of every white piece and a bitboard of every bishop, one logical AND operation and you have a bitboard of every white bishop. In a single CPU clock cycle you've basically done 64 boolean operations in parallel. Incredibly powerful stuff. But bitboards in ChessV are more complicated since boards can be of varying sizes. As such, they are much less efficient than bitboards in orthodox chess engines, but, when used judiciously, they still offer tremendous benefit. Internally, they store three 64-bit integers, allowing ChessV to support games with up to 192 squares/cells.

The Board class maintains bitboards containing the occupied squares for every player and piece type. For each piece type we need to check for extinction, we ask the board for the bitboard with the current player's pieces of the appropriate type. Then the BitCount property will tell us how many bits are set, and thus, how many pieces of that type the player has. If the answer is zero, then that type has gone extinct and the player has lost.

The rest of the class is just initialization to get the TestForWinLossDraw function what it needs. Here's the full code for the ExtinctionRule:

```
public class ExtinctionRule: Rule
{
    // *** CONSTRUCTION *** //

    public ExtinctionRule( string types )
    {
        extinctionTypesNotation = types;
    }


    // *** INITIALIZATION *** //

    public override void PostInitialize()
    {
        base.PostInitialize();
        List<PieceType> types = Game.ParseTypeListFromString( extinctionTypesNotation );
        extinctionTypeNumbers = new List<int>();
        foreach( PieceType type in types )
            extinctionTypeNumbers.Add( type.TypeNumber );
    }


    // *** OVERRIDES *** //

    public override MoveEventResponse TestForWinLossDraw( int currentPlayer, int ply )
    {
        foreach( int typeNumber in extinctionTypeNumbers )
            if( Board.GetPieceTypeBitboard( currentPlayer, typeNumber ).BitCount == 0 )
                return MoveEventResponse.GameLost;
        return MoveEventResponse.NotHandled;
    }


    // *** PROTECTED DATA MEMBERS *** //

    protected string extinctionTypesNotation;
    protected List<int> extinctionTypeNumbers;
}
```

The constructor takes a string with a list of the notations of the extinction types. `Game.ParseTypeListFromString` is a helper function provided by the Game class to turn this string into a list of PieceTypes. A PieceType is a complex object. For efficiency, every type that is used in a given game (chess variant) has a TypeNumber – an integer starting at one. Internally, the architecture uses type numbers almost exclusively. To make the TestForWinLossDraw function as efficient as possible (since it's being called a lot) we convert the PieceTypes into the type numbers during the initial setup and store those. PostInitialize is a virtual function that is called for all rules after the initialization of the Game class. At this point the PieceTypes in the game have been identified and assigned their type numbers, so we can perform the PieceType to type number translation then.

D. Chess256

Chess256 by Mats Winther is Chess with a randomized setup (similar to Fischer Random Chess, but a little easier to deal with for demonstration purposes.) In this game, every pawn may start on either the second or third rank. The game, therefore, has 256 possible starting arrays.

First, we create a game variable to store the number of the starting position being played:

```
[GameVariable] public IntVariable PositionNumber { get; set; }
```

We must use the special IntVariable type rather than C#'s internal int type because IntVariables store, in addition to the value, a range of acceptable values and information about whether it has been initialized to any actual value or is still undefined. We configure this game variable with the following line in the SetGameVariables function:

```
    PositionNumber = new IntVariable( 1, 256 );
```

This creates the IntVariable with the acceptable range of values. We do not, however, set any specific value. We normally assign all our variables in SetGameVariables, but in this case we deliberately leave it unset. When ChessV starts a game, after all the normal initialization is done, it checks to see if any game variables are unset. If so, it chooses random values, and displays a dialog box giving a preview, and offering the user the opportunity to change the values. The same thing can be seen when you start a game of Chess with Different Armies. That game has game variables that are ChoiceVariables, (pick from a list of choices), called WhiteArmy and BlackArmy. They are uninitialized so a dialog appears for army selection (with randomly armies pre-selected.) These dialogs are not specifically coded for by any Game class – it's just a feature that you get one whenever you have uninitialized game variables.

The only thing which is really changed in Chess256 is the starting array, which can be customized by setting the Array game variable. But in this case, we can't just set Array to some hard-coded value. The array needs to be determined based on the selected value of another game variable. We want to be able to set PositionNumber and have Array automatically reflect that. To see how this is done, let's first look at the implementation of SetGameVariables:

```
public override void SetGameVariables()
{
    base.SetGameVariables();
    PositionNumber = new IntVariable( 1, 256 );
    Array = "rnbqkbnr/#{BlackPawns}/8/8/#{WhitePawns}/RNBQKBNR";
}
```

We have embedded placeholders in the Array variable - #{BlackPawns} and #{WhitePawns}. These symbols will be dynamically expanded when the time comes to place the pieces. Note that BlackPawns and WhitePawns are *not* GameVariables. This is important. The user is prompted to supply values for GameVariables that have not been initialized. These symbols should be defined later in the initialization process, after the user has selected the value for the PositionNumber game variable. To define this type of symbol, there is another virtual function that can be overridden – SetOtherVariables. Here is the definition for Chess256:

```
public override void SetOtherVariables()
{
    base.SetOtherVariables();
    if( PositionNumber.Value == null )
    {
        SetCustomProperty( "BlackPawns", "8/8" );
        SetCustomProperty( "WhitePawns", "8/8" );
        return;
    }

    //  determine Black's second-row pawns
    int position = (int) PositionNumber.Value - 1;
    int bitNumber = 0;
    string notation2ndRank = "";

    while( bitNumber < 8 )
    {
        if( (position & (1 << bitNumber)) == 0 )
        {
            notation2ndRank += 'p';
            bitNumber++;
        }
        else
        {
            int emptySpaceCount = 1;
            bitNumber++;
            while( bitNumber < 8 && (position & (1 << bitNumber)) != 0 )
            {
                emptySpaceCount++;
                bitNumber++;
            }
            notation2ndRank += Convert.ToChar( '0' + emptySpaceCount );
        }
    }

    //  determine Black's third-row pawns
```

```
    bitNumber = 0;
    string notation3rdRank = "";

    while( bitNumber < 8 )
    {
        if( (position & (1 << bitNumber)) != 0 )
        {
            notation3rdRank += 'p';
            bitNumber++;
        }
        else
        {
            int emptySpaceCount = 1;
            bitNumber++;
            while( bitNumber < 8 && (position & (1 << bitNumber)) == 0 )
            {
                emptySpaceCount++;
                bitNumber++;
            }
            notation3rdRank += Convert.ToChar( '0' + emptySpaceCount );
        }
    }

    SetCustomProperty( "BlackPawns", notation2ndRank + "/" + notation3rdRank );
    SetCustomProperty( "WhitePawns", notation3rdRank.ToUpper() + "/" + notation2ndRank.ToUpper()
)
```

The SetCustomProperty function allows the creation of new variables without even the need to declare them ahead of time. This code constructs the strings for BlackPawns and WhitePawns that should be substituted into the Array. This is not the only way the dynamic array could be accomplished however. We could also have overridden the LookupGameVariable function, which is called whenever the architecture needs to get the value of a game variable, and used this to intercept the lookup of the "Array" game variable and compute the value on the fly. ChessV is extremely extensible with access points all over the place that can be hooked onto.

With these examples we have seen the most important elements, but we've covered less than half. To continue exploring ChessV extensibility, look at the source code for the implementations of other variants.

VI. Defining New Rules

A. Overview

As previously described, Rules are containers of virtual functions that can be used to hook into the architecture to handle events and change behavior. This section describes the various virtual functions that can be handled with examples of where they are used.

B. Overridable Functions

1. MoveBeingMade

This function is called whenever a move is being played immediately after the board is updated. Rules can use this to declare a move illegal, or simply to update internal information for other purposes. Of course, they can simply choose to do nothing.

The CheckmateRule uses this function to determine if the king of the side that is moving is now attacked. If so, it returns `MoveEventResponse.IllegalMove` to signal that this move is illegal (because it moves the king into check or leaves it in check.) Before allowing the user to move through the GUI, all generated moves are tested by being made and unmade to ensure they are legal.

For an example of a Rule that uses this function to update internal information, the EnPassantRule uses this to detect multi-space pawn moves and internally record the fact that they may be captured En Passant on the following move. The CastlingRule uses it to track loss of castling privileges when kings or rooks are moved.

2. MoveBeingGenerated

ChessV understands basic step and slide movements for piece types. When it is time to generate moves, it loops through all piece types, then through all move capabilities for those types, stepping in the given directions generating moves for

each step. For each move it is being generated, it calls this function in all rules. A handler may declare the move illegal by returning `MoveEventResponse.IllegalMove`, or it may generate alternate move(s) and return `MoveEventResponse.Handled`, in which case the original move will not be generated. Or it can return `MoveEventResponse.NotHandled` to do nothing.

The BasicPromotionRule uses this to handle promotions. The internal move generator would happily generate the push of a pawn to the last rank, but when it notifies BasicPromotionRule, it receives `MoveEventResponse.Handled`, so it cancels generation of the normal pawn move. The rule handler function manually generates the appropriate moves where the pawn changes types.

The PieceLocationRestrictionRule uses this to limit the locations a piece may occupy in a configurable way. In Eurasian Chess, for example, it declares illegal any move where the king would be crossing the river. This could also be accomplished by overriding MoveBeingMade and declaring the moves illegal at that time. The results would be the same, but it is more efficient to do it here, saving the need to make and unmake the move later.

3. MoveBeingUnmade

   The compliment to MoveBeingMade, this function is called when a move is undone. This happens not only when the user takes back a move on the board, but also during engine thinking as moves are made and unmade during brute-force analysis. No rules currently make use of this, but some may conceivably need to use this to update internal state information it is provided for completeness.

4. TestForWinLossDraw

   This is called whenever the architecture needs to check to see if a game has ended. A handler can return `MoveEventResult.GameWon`, `MoveEventResult.GameLost`, `MoveEventResult.GameDrawn`, or `MoveEventResult.NoResult`.

   The Move50Rule and RepetitionDrawRule use this to declare the game drawn by 50-move rule and three-fold repetition respectively. We've already seen how the ExtinctionRule uses this to implement the victory condition in Extinction Chess.

5. NoMovesResult

   This function is called when a position results in no legal moves at all. It is similar to TestForWinLossDraw, but is called under specific circumstances, and one rule *must* return a game result.

   The CheckmateRule implements it, checking to see if the king of the side to move is attacked. The result is a loss if it is, and a draw if it is not (thereby handling stalemate properly.)

6. GenerateSpecialMoves

   This function is called whenever moves are being generated to allow ad hoc generation of moves that aren't related to the normal movement capabilities of the pieces. Whereas MoveBeingGenerated is called for every move being generated by the internal move generator, this is called only once each turn (for each rule.)

   The CastlingRule, EnPassantRule, and KingsLeapRule are examples of rules that generate moves in this way. The function receives a MoveList object as a parameter and generates moves manually by calling member functions of the MoveList to add the moves. Adding a move is accomplished by calling `BeginMoveAdd`, followed by one or more calls to `AddPickup` and/or `AddDrop` to lift pieces from the board and (optionally) set them back down, and finally a call to `EndMoveAdd`.

7. DescribeMove

   This function can alter the way moves are described when displaying to the user or how they are recorded in the move notation.

8. SetDefaultsInFEN

This is used to initialize parts of the FEN related to the rule. The CastlingRule, for example, uses it to set the {castling} component of the FEN appropriately. The FEN is itself an object that supports reading and writing to named components so that the traditional FEN representation for Chess can be preserved while allowing extensibility for games with new rules that need to store information about a position in order to be able to load games from an FEN string.

9. SavePositionToFEN

   The complement to SetDefaultsInFEN, this function is called when we need to convert the current game position back into an FEN string. Rules update the components of the FEN string for which they are responsible.

10. PositionLoaded

    The compliment to SetDefaultsInFEN, this is called whenever a position is loaded from an FEN. The FEN object is passed in as a parameter and the rule can ask this object for the value of the FEN component(s) appropriate to the given rule and initialize its internal state appropriately – e.g., initializing castling rights or determining whether a pawn is currently capable of being captured En Passant.

11. PositionalSearchExtension

    This is used by the internal engine to allow a rule to deepen the brute-force search under specific conditions. Presently it's only used by the CheckmateRule to deepen the search one ply when the king is in check.

12. GetPositionHashCode

    This function is called for each rule whenever the Zobrist hash code for the current position is needed, and any value returned is XOr'd with the existing hash. The CastlingRule and EnPassantRule, for example, overload this function so that the castling rights and en passant square are included in the hash.

C. MoveCompletionRule

The MoveCompletionRule is a special kind of rule derived from the Rule class. Every Game must have exactly one MoveCompletionRule – no more, no less. The Game class automatically sets this to an instance of MoveCompletionDefaultRule, and for most games there is no need to replace it.

What the MoveCompletionRule handles is the switching of the current player after each move. The default handler simply changes the current player after each move – white, black, white, black … The purpose of placing this functionality into a rule that can be replaced is to allow games with alternate move orders, such as double-move variants like Marseillais Chess.