“可支持人类和 AI 博弈的国际象棋棋盘软件”软件开发课程项目——软件设计报告说明书

王继成 《软件开发方法》课程 第 6 小组

1 引言

1.1 编写目的

本文档旨在对“可支持人类和 AI 博弈的国际象棋棋盘软件”进行概要设计，描绘出该软件的大致体系结构；以及对于详细设计进行说明，以供用户了解本软件的系统结构和详细设计方式。

1.2 定义、首字母缩写和缩略语

名词	解释说明
本软件	指本项目的预期开发成果之一，“可支持人类和 AI 博弈的国际象棋棋盘软件”，英文名为 The Chess Board 或 Board。
AI	国际象棋人工智能（Artificial Intelligence）程序的缩略语。 AI 是《人工智能原理》课程的各参赛小组独立设计编写的、能够有效思考并模拟人类作为国际象棋比赛任意一方进行走子博弈的人工智能程序。
比赛	指《人工智能原理》课程的“国际象棋人工智能程序比赛”。
参赛者	指参加比赛的各小组成员（人）。
参赛 AI	指参加比赛的各小组编写完成的 AI，将模拟人进行国际象棋博弈。
组织者	比赛的组织者，也是《人工智能原理》课程的三位课程负责人。
SAN	标准代数记谱法（Standard Algebraic Notation）的缩写。 是一种国际象棋走子的记谱法，采用数字、字母和符号构成的短字符串记录走子的每一步。其详细定义见参考资料 3)与附录 5.1。

PGN	便携式游戏记号（Portable Game Notation）的缩写。 一种在电子计算机上存储国际象棋棋谱的文件格式，其主要部分为 SAN 记谱法的字符串。后文将不对 PGN 和 SAN 作明显的区分。 其详细定义见参考资料 4)。
FEN	福斯夫-爱德华兹记谱法（Forsyth-Edwards Notation）的缩写。 是一种国际象棋局面的记谱法，采用数字、字母、符号构成的长字符串记录一个瞬间的棋盘局面。其详细定义省略，见参考资料 5)。

1.3 参考文献

1) IEEE 标准文档。

2) 《“可支持人类和 AI 博弈的国际象棋棋盘软件”软件开发课程项目——目标规模说明书》。

3) “代数记谱法”的 Wikipedia 词条页面：https://zh.wikipedia.org/wiki/%E4%BB%A3%E6%95%B8%E8%A8%98%E8%AD%9C%E6%B3%95 。（可能需要翻墙，下略）

4) “Portable Game Notation”的 Wikipedia 词条页面：https://en.wikipedia.org/wiki/Portable_Game_Notation 。

5) “福斯夫-爱德华兹记谱法”的 Wikipedia 词条页面：https://zh.wikipedia.org/wiki/%E7%A6%8F%E6%96%AF%E5%A4%AB%EF%BC%8D%E6%84%9B%E5%BE%B7%E8%8F%AF%E8%8C%B2%E8%A8%98%E8%99%9F%E6%B3%95。

6) 《“可支持人类和 AI 博弈的国际象棋棋盘软件”软件开发课程项目——软件需求规格说明书》

2 项目总体描述

本部分略去，请参照《“可支持人类和 AI 博弈的国际象棋棋盘软件”软件开发课程项目——软件需求规格说明书》中有关项目总体描述的部分。

第一部分 体系结构（总体、概要）设计

1 整体架构描述

本节将描述本软件的整体架构，从逻辑视角和组合视角来描述，采用 UML 包图、构件图。

1.1 逻辑视角

1.1.1 体系结构设计风格

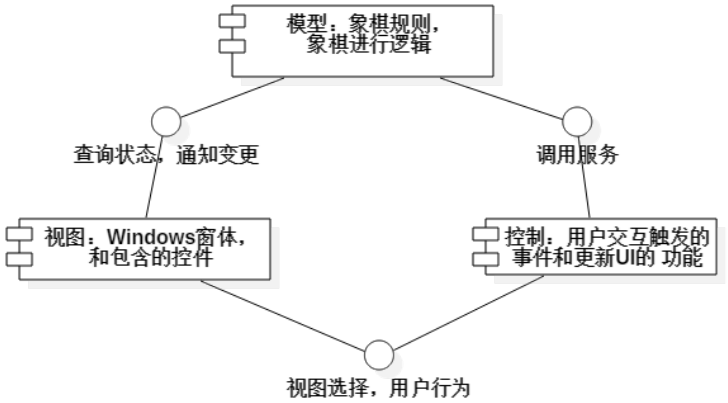
本软件体系结构设计的风格采用模型-视图-控制器（Model-View-Control，MVC）风格。采用该风格的方案明显较好，因为：

- 实际开发时使用 C#结合.NET Framework，非常适合于实现 MVC 风格的体系结构；
- 能够促进并行开发，缩短开发时间；
- 该风格的部件区分明朗，便于进行体系结构设计和详细设计。

MVC 风格将整个系统分为三个部件（子系统）：

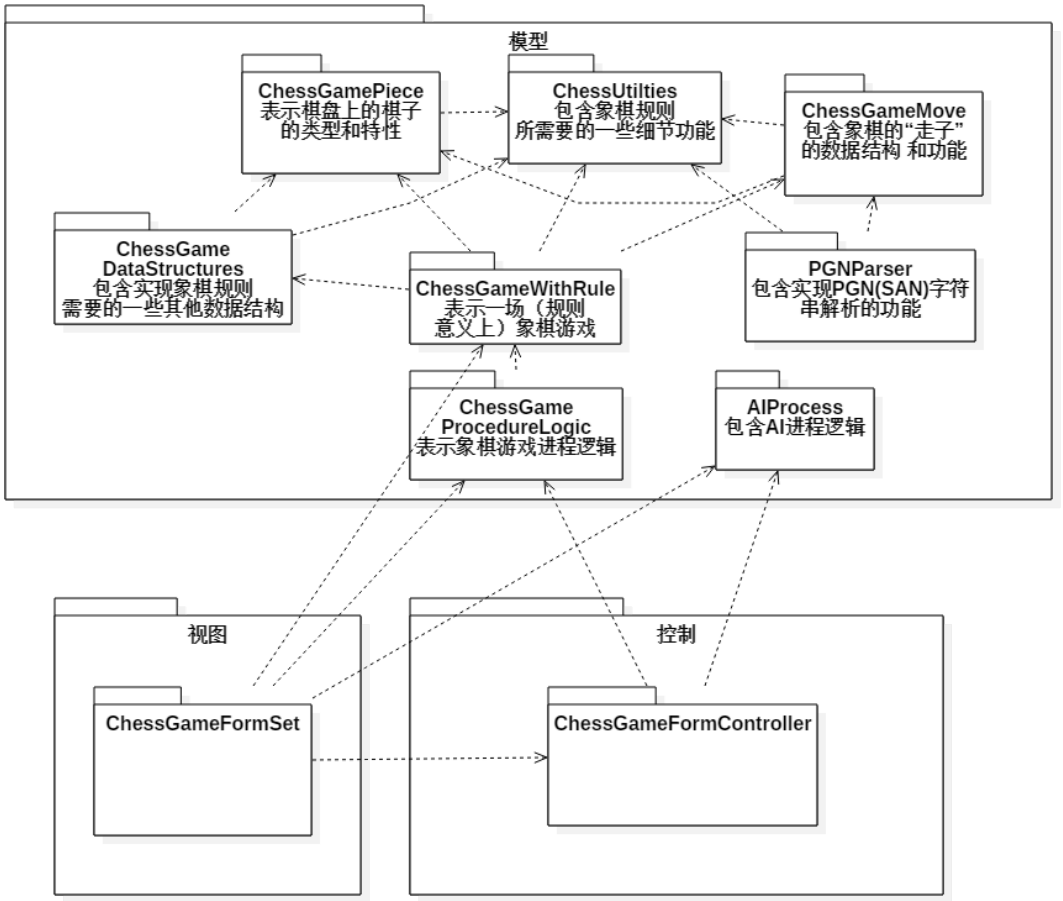
- 模型（Model）：封装系统的数据和状态信息，也包含业务逻辑。在本软件中，模型包含国际象棋的规则部分、国际象棋游戏进程的逻辑。
- 视图（View）：提供业务展现，接受用户行为。在本软件中，视图包含程序的显示窗体、控件。控件既可向用户展示信息，也可以被用户以点击的形式交互。
- 控制（Controller）：封装系统的控制逻辑。在本软件中，控制包含用户点击控件后触发的事件函数，以及刷新 UI 所需调用的事件函数。

构件图如下：



1.1.2 概要功能设计和逻辑包图

由于本软件的界面和控制不复杂，实现较为简单，视图和控制部件包均可只用一个逻辑包实现；模型涉及到功能较多，用多个逻辑包实现。用包图表达的最终软件体系结构逻辑设计方案如下：



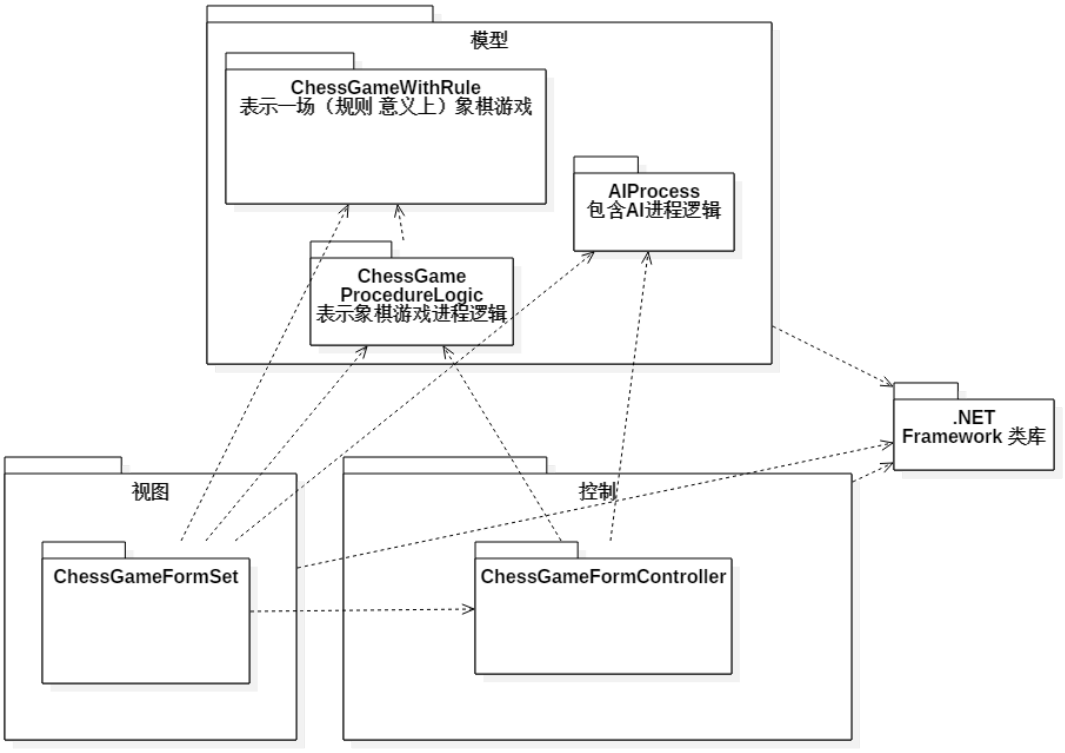
1.2 组合视角

1.2.1 开发包设计

在逻辑视角的基础上，可以用组合视角对于体系结构进行开发包的设计。由于我们的项目较为简单，故采用以下的开发包设计：

- 每一个组合包最多转化为一个开发包
- 模型部件中依赖关系较多的包组合为一个开发包
- 逻辑包中没有循环依赖需要解决，故无需再增加开发包
- 为简洁，不再另设不同部件之间的接口包。

在引入.NET Framework 框架提供的类库之后，整个软件的开发包图如下：



各包的名称、功能和依赖关系均已在图中呈现，故不另外列开发包表。

1.2.2 运行时进程

略。由于软件简单，故运行时排除 AI 进程外，只有一个主进程。

1.2.3 物理部署

略。由于软件简单，只需要一个可执行文件部署在本地计算机。

2 模块分解

对于诸模块的分解设计采用结构视角和接口视角来说明。

2.1 模块的职责

按照 MVC 的部件划分，可以直接将每个部件转换为一个大的模块：Model 模块、View 模块、Control 模块。其职责如下：

模块	职责
Model	记录软件运行的状态、象棋的局面信息，处理象棋的走子、规则。
View	作为 Windows 窗体的模块，处理图形界面的输出和用户交互的输入。包括显示象棋棋盘、控制象棋游戏流程的按钮等控件。
Control	包含与 View 有关的控制逻辑，根据用户交互调用 Model 内的相关功能，并根据功能调用之后的系统状态刷新 View。

不同模块之间通过简单的函数调用完成连接。

2.2 MODEL 的分解

Model 模块包含与象棋的状态和信息有关的对象类，如 ChessGame（以象棋规则为中心的象棋棋局类）、ChessGameLogic（以象棋游戏进程逻辑为中心的象棋棋局类）、ChessPieces（象棋的棋子类）等。

2.2.1 Model 分解后的职责

Model 模块包含三个开发包，其职责如下表所示：

开发包模块	职责
ChessGameWithRule	负责以象棋规则为中心实现象棋游戏
ChessGameLogic	负责以象棋游戏进程逻辑为中心实现象棋游戏，依赖 ChessGameWithRule
AIProcess	负责实现 AI 进程的载入、启动、停止、读写等逻辑

2.2.2 Model 分解后的接口规范

注：只列出对于本软件有关键作用的接口，重要性较小的接口如计时、用户设置有关的在此不列出。

2.2.2.1 ChessGameWithRule 的接口规范

提供的服务（供接口）		
ChessGameWithRule.WhoseTurn	语法	public Player WhoseTurn()
	前置条件	游戏局面按规则进行下来
	后置条件	返回当前局面执子方，以便针对每一方进行判断
ChessGameWithRule.Moves	语法	public ReadOnlyCollection<MoreDetailedMove> Moves
	前置条件	游戏局面按规则进行下来
	后置条件	返回游戏局面中已经有的历史走子，以便对于走子进行分析处理或输出
ChessGameWithRule.ApplyMove	语法	public virtual MoveType ApplyMove(Move move, bool alreadyValidated)
	前置条件	游戏局面按规则进行下来
	后置条件	对于游戏局面应用传入的 Move 对象，使得局面进行改变
ChessGameWithRule.GetValidMoves	语法	public virtual ReadOnlyCollection<MoreDetailedMove> GetValidMoves(Player player, bool returnIfAny, bool careAboutWhoseTurnItIs)
	前置条件	游戏局面按规则进行下来
	后置条件	在游戏局面中，给出当前行动方可以做出的走子，返回的是一个 MoreDetailedMove 的集合，方便从中选择并给出合法走子。
ChessPieces.GetFenCharacter	语法	public abstract char GetFenCharacter()
	前置条件	初始化了一个具体的棋子对象
	后置条件	返回该具体的棋子对象对应的 FEN 字符，方便输出。
ChessUtilities.GetOpponentOf	语法	public static Player GetOpponentOf(Player player)

	前置条件	已知象棋游戏中某方 Player
	后置条件	返回该方 Player 的对方
PgnMoveReader. ParseMove	语法	public static Move ParseMove(string moveInStr, Player player, ChessGame game)
	前置条件	已知一个 SAN 字符串
	后置条件	返回该 SAN 字符串分析后的走子对象 Move
需要的服务（需接口）		
除了.NET Framework 类库中的接口（较多且调用较频繁，故省略），没有需要调用的外部接口。		

2.2.2.2 AIProcess 的接口规范

提供的服务（供接口）		
AIProcess. Start	语法	public void Start()
	前置条件	AI 程序已经载入, AIProcess 对象已经初始化完毕
	后置条件	AI 程序启动，可以开始与其通信
AIProcess.Kill	语法	public void Kill()
	前置条件	AI 程序载入并正常启动运行
	后置条件	AI 程序终止
AIProcess.Wait	语法	public void Wait()
	前置条件	AI 程序正在思考，即将给出走子输出
	后置条件	等待直到 AI 程序给出走子输出
AIProcess.LineProcess	语法	public event LineProcessorHandler LineProcess
	前置条件	AI 程序正在思考，即将给出走子输出
	后置条件	这是一个异步事件，当 AI 进程给出走子输出时，处理该输出，并使得 Wait 能够继续运行直到返回
AIProcess.WriteLine	语法	public void WriteLine(string str)

	前置条件	AI 程序正在接受输入
	后置条件	将走子字符串输入给 AI 进程，使得 AI 进程能够了解对方的走子，进而轮到己方走子
需要的服务（需接口）		
除了.NET Framework 类库中的接口（较多且调用较频繁，故省略），没有需要调用的外部接口。		

2.2.2.3 ChessGameLogic 的接口规范

提供的服务（供接口）		
ChessGameLogic.Init	语法	void Init()
	前置条件	主窗体载入完毕, ChessGameLogic 对象完成构造
	后置条件	ChessGameLogic 对象完成初始化, 可以进行游戏进程
ChessGameLogic.Start	语法	public void Start(GameMode mode)
	前置条件	主窗体载入完毕, ChessGameLogic 对象完成构造和初始化
	后置条件	以某一 GameMode 游戏模式开始游戏
ChessGameLogic.Stop	语法	public void Stop()
	前置条件	ChessGameLogic 对象正在进行游戏
	后置条件	停止游戏
ChessGameLogic.ResetGame	语法	public void ResetGame()
	前置条件	任何时候
	后置条件	重置游戏进程逻辑的所有数据
ChessGameLogic.*StatusUpdated	语法	public event *StatusUpdatedEventHandler *StatusUpdated
	前置条件	当游戏逻辑的状态更新时
	后置条件	这是一个异步事件，当游戏逻辑状态更新时，就调起订阅在其上的代理函数。通常情况下包括更新 UI 控件的操作。

ChessGameLogic.Set*Status	语法	public void Set*Status(ChessBoardGame*State state, bool updateImportant, string reason = null)
	前置条件	外部需要改变游戏逻辑的状态时
	后置条件	更新游戏逻辑的状态。
ChessGameLogic.ApplyMove	语法	public MoveType ApplyMove(Move move, bool alreadyValidated, out Piece captured)
	前置条件	游戏进程已经开始，需要应用一个走子
	后置条件	应用该走子，触发 AI 更新，并视情况触发接受下一步走子的新线程
ChessGameLogic.LoadAIExec	语法	public bool LoadAIExec(Player player, string execPath, string execArguments)
	前置条件	游戏进程尚未开始，需要载入 AI
	后置条件	载入该 AI，并验证其是否正常使用
需要的服务（需接口）		
.NET Framework 类库中的接口（较多且调用较频繁，故省略）以及上表中列出的所有 ChessGameWithRule、AIProcess 的接口。		

2.3 View 的分解、Control 的分解

从《需求规格说明书》中给出的 UI 图中，可以看到使用到的所有 WinForm 控件，这些构成 View 模块。由于其接口只有一个 Initialize()初始化方法，只有主界面，并无其他界面的跳转，以及 Control 的内容亦大多为 View 中控件的点击等交互事件，故 View 和 Control 的分解与接口规范此处省略。可以参考编码后的 Form 部分代码。

3 信息视角

本软件所需要处理的数据为用户设置数据。用户设置数据的数据定义、数据类型见《需求规格说明书》中的相关内容。其具体实现是 XML 格式的文件存储：

```
<userSettings>
        <TheChessBoard.Properties.Settings>
            <setting name="WatchChessTime" serializeAs="String">
                <value>100</value>
```

```
            </setting>
            <setting name="WhiteDefaultExecPath" serializeAs="String">

<value>..\..\..\ChessArtificialRetard\bin\Debug\ChessArtificialRetard.exe</value>
            </setting>
            <setting name="WhiteDefaultExecArguments" serializeAs="String">
                <value />
            </setting>
            <setting name="BlackDefaultExecPath" serializeAs="String">

<value>..\..\..\ChessArtificialRetard\bin\Debug\ChessArtificialRetard.exe</value>
            </setting>
            <setting name="BlackDefaultExecArguments" serializeAs="String">
                <value />
            </setting>
            <setting name="AutoSaveAIConfig" serializeAs="String">
                <value>True</value>
            </setting>
            <setting name="HideAIWindow" serializeAs="String">
                <value>False</value>
            </setting>
            <setting name="AIExecPathHistory" serializeAs="Xml">
                <value>
                    <ArrayOfString xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema" />
                </value>
            </setting>
            <setting name="AIExecArgumentsHistory" serializeAs="Xml">
                <value>
                    <ArrayOfString xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema" />
                </value>
            </setting>
        </TheChessBoard.Properties.Settings>
    </userSettings>
```

第二部分 详细设计

本软件详细设计的基本方法为面向对象设计方法（Object-oriented Design Method），意在将各个构件实现时，用抽象为一系列对象的方式看待。

1 结构视角

1.1 MODEL 的分解

1.1.1 模块概述和整体结构

Model 模块的职责为记录软件运行的状态、象棋的局面信息，处理象棋的走子、规则。在软件的体系结构设计中，其下分为 ChessGameWithRule、ChessGameLogic、AIProcess 三个包，分别包含象棋规则、象棋游戏进程逻辑、AI 进程逻辑三个方面的逻辑。后两个包可各用一个类实现，而前一个包由于构成与所实现的功能更复杂一些，故可以用多个类来实现。

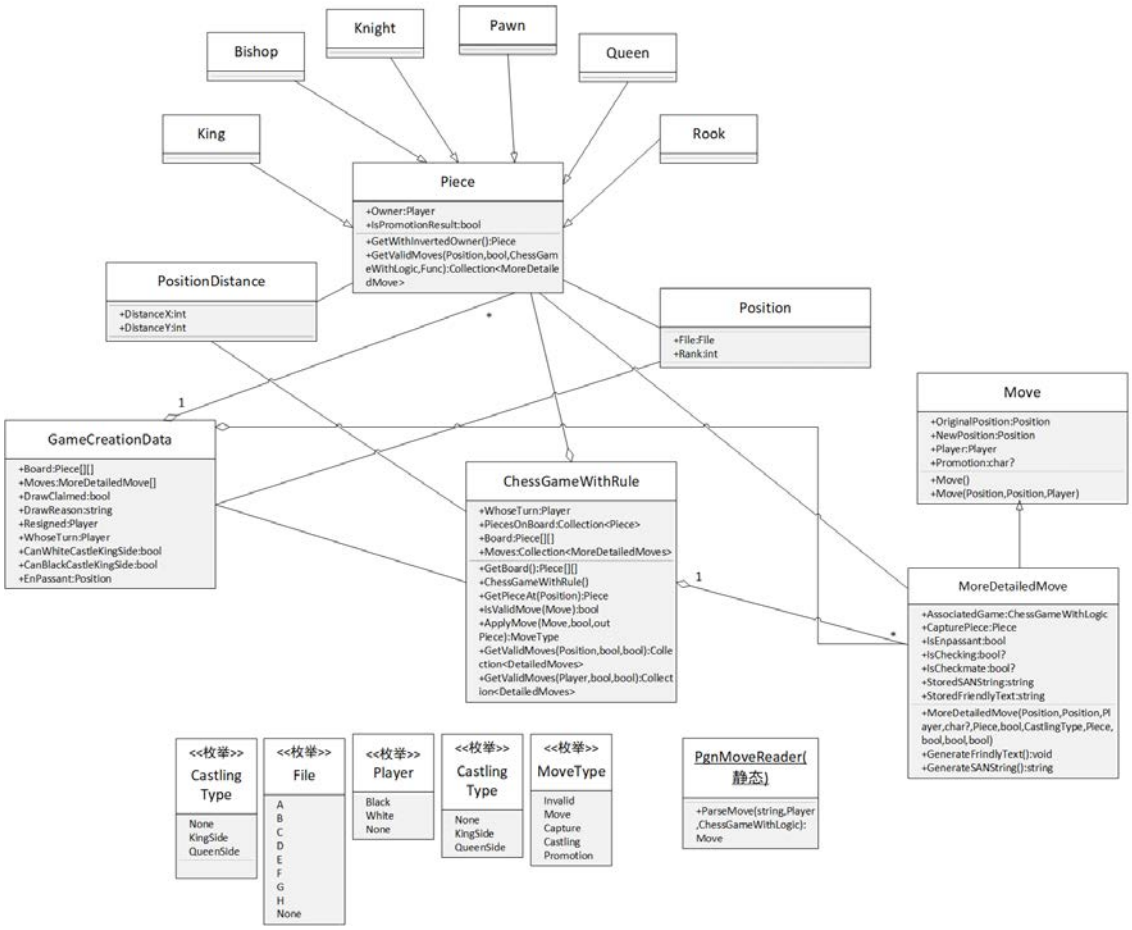
这三个开发包的内部构造和职责、相互协作描述如下：

- ChessGameWithRule 包中的核心实现类是同名类 ChessGameWithRule（在实际代码编写中，更名为 ChessGame）。它除了对.NET Framework 框架，以及同一包内的一些数据结构类有依赖之外，是一个自成一体的象棋规则实现类。一个 ChessGameWithRule 对象可以完备地从规则角度上实现一局象棋游戏的过程。其内有包含棋盘（棋子对象构成的数组 Piece [][]）、历史行棋（Move 对象构成的列表 List<Move>）等，也有 ApplyMove（实现走子）、GetValidMoves（获得当前所有合法走子）等具体功能方法。除此之外，ChessGameWithRule 包还包含与象棋规则有关的数据结构，如 Piece（棋子）、Move->MoreDetailedMove（走子，MoreDetailedMove 继承自 Move，包含更多信息）、Position（位置）等等，被 ChessGameWithRule 核心类所聚合。ChessGameWithRule 还包含了一个用于处理 SAN 字符串为 Move 对象的分析函数 PgnMoveParser，便于游戏进程逻辑层面的使用。
- AIProcess 由一个同名类实现，实现一个 AI 进程的逻辑，如启动、停止、标准输入输出的读写。它调用.NET Framework 系统类，可以控制启动、停止系统进程，并操作标准输入输出。可以说它是系统进程与本软件的接洽。
- ChessGameLogic 由一个同名类实现，注重于象棋游戏的进程（开始、循环走棋、何时结束游戏）来实现一局象棋游戏。由于象棋游戏的进程取决于规则，故它依赖并使用 ChessGameWithRule 作为规则的实现。同时，它也使用 AIProcess 类，向其发送有关于黑白双方 AI 的命令，以实现机器博弈。至于人工博弈，人类的行棋是通过用户界面，从 View 模块传导到 Control 模块，再

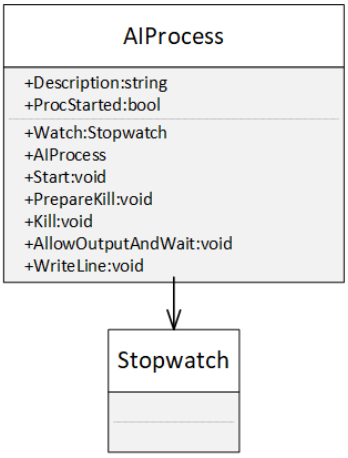
调用 ChessGameLogic 里的 ApplyMove 实现的，不全部由 ChessGameLogic 实现。

基于以上的设计，可以画出类图如下：（不涉及到核心功能的类或方法，有所省略）

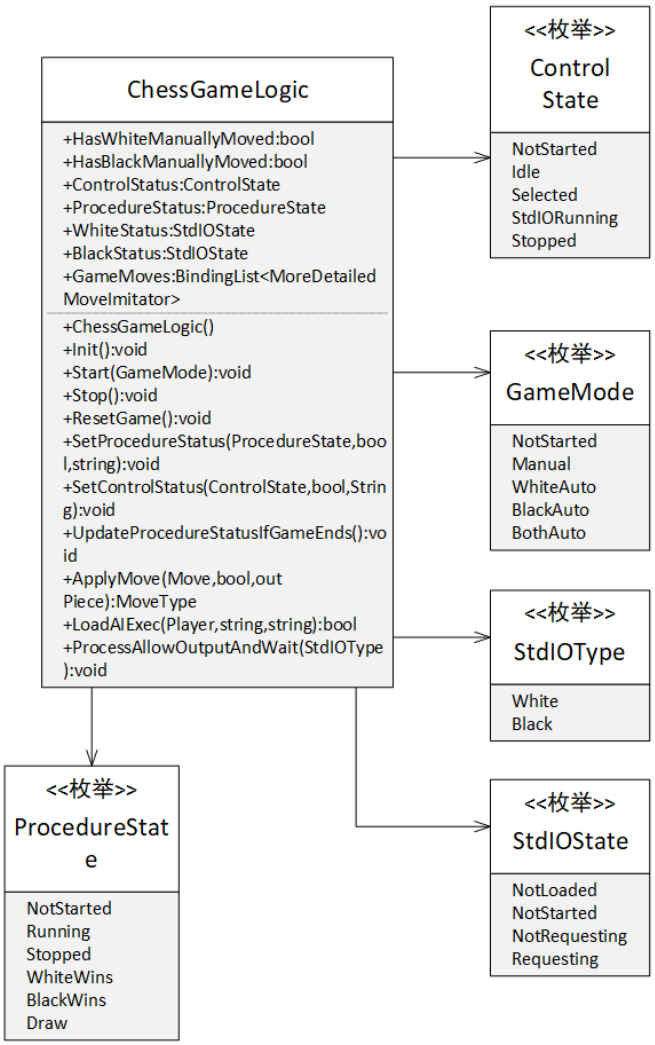
1.1.1.1 ChessGameWithRule 内的类图



1.1.1.2 AIProcess 内的类图



1.1.1.3 ChessGameLogic 内的类图（实际实现中，ChessGameLogic 命名为 ChessBoardGameFormLogic）



以下将不再分每一个小包进行接口规范的描述，而是直接对 Model 内的类进行接口规范的描述。

1.1.2 内部类的接口规范

与“体系结构设计”中的 Model 接口有所重合的类接口，这里有些就省略不列出。现将“体系结构设计”中细化的 Model 内部类接口规范描述如下：

ChessGameWithRule 类的接口规范

提供的服务（供接口）		
	语法	`public Piece[][] GetBoard()`

ChessGameWithRule.GetBoard	前置条件	对局已经开始
	后置条件	返回当前棋盘信息
ChessGameWithRule.GetPieceAt	语法	public Piece GetPieceAt(File file, int rank)
	前置条件	当前位置有棋子
	后置条件	返回该位置棋子信息
ChessGameWithRule.GetValidMoves	语法	Public virtual ReadOnlyCollection<MoreDetailedMove> GetValidMoves(Position from, bool returnIfAny, bool careAboutWhoseTurnItIs)
	前置条件	对局已经开始，且走子合法
	后置条件	返回所有从 Position from 开始的合法走子，返回 MoreDetailedMove 的只读集合。
ChessGameWithRule.IsCheckmated	语法	public virtual bool IsCheckmated(Player player, bool useCache = true)
	前置条件	对局已经开始，在一方走子结束后
	后置条件	返回某方 Player 是否被将死。
ChessGameWithRule.IsInCheck	语法	public virtual bool IsInCheck(Player player, bool useCache = true)
	前置条件	对局已经开始，在一方走子结束后
	后置条件	返回某方 Player 是否被将军。
ChessGameWithRule.IsValidMove	语法	public virtual bool IsValidMove(Move move, bool validateCheck, bool careAboutWhoseTurnItIs)
	前置条件	对局已经开始，且输入的走子信息合法
	后置条件	判断某个 Move 在该游戏中是不是合法的 Move。
ChessGameWithRule.SetPieceAt	语法	public virtual void SetPieceAt(File file, int rank, Piece piece)
	前置条件	对局已经开始

	后置条件	在某行某列设置或移除一个棋子。
需要的服务（需接口）		
ChessDotNet.Piece	表示棋子的抽象类	
ChessDotNet.MoreDetailedMove	是 Move 的派生类，现在已经全面使用，记录了最详细的这个 Move 的信息。	
ChessDotNet.ChessUtilities	是一个静态类，里面列了一些 ChessGame 用到的功能。	
…	…	

MoreDetailedMove 类的接口规范

提供的服务（供接口）		
MoreDetailedMove.GenerateFriendlyText	语法	`public void GenerateFriendlyText()`
	前置条件	对局已经开始
	后置条件	生成文字描述
MoreDetailedMove.GenerateSANString	语法	`public string GenerateSANString(ChessGame gameBeforeTheMove)`
	前置条件	对局已经开始
	后置条件	生成 SAN 字符串
需要的服务（需接口）		
ChessDotNet.Piece	表示棋子的抽象类	
ChessDotNet.Move	最基础的 Move 类，记录一个走子的基础信息。	

Piece 类（抽象类，具体棋子类的基类）的接口规范

提供的服务（供接口）		
Piece.Equals	语法	`public override bool Equals(object obj)`
	前置条件	对局已经开始，获得两个 Piece 对象信息
	后置条件	返回两个 Piece 对象是否相等的结果。
Piece.GetHashCode	语法	`public override int GetHashCode()`

	前置条件	对局已经开始
	后置条件	返回该棋子的哈希值，用于查询。
Piece.IsValidMove	语法	public abstract bool IsValidMove(Move move, ChessGame game)
	前置条件	对局已经开始
	后置条件	得到这个 Piece 对象在 ChessGame 对象中作 Move 是否合法
Piece.GetValidMoves	语法	public abstract ReadOnlyCollection<MoreDetailedMove> GetValidMoves(Position from, bool returnIfAny, ChessGame game, Func<Move, bool> gameMoveValidator)
	前置条件	对局已经开始
	后置条件	得到这个 Piece 对象在 ChessGame 对象中从 Position 出发对应的所有合法走子

AIProcess 类的接口规范

提供的服务（供接口）		
AIProcess. Start	语法	public void Start()
	前置条件	AI 程序已经载入，AIProcess 对象已经初始化完毕
	后置条件	AI 程序启动，可以开始与其通信
AIProcess.Kill	语法	public void Kill()
	前置条件	AI 程序载入并正常启动运行
	后置条件	AI 程序终止
AIProcess.Wait	语法	public void Wait()
	前置条件	AI 程序正在思考，即将给出走子输出
	后置条件	等待直到 AI 程序给出走子输出
AIProcess.LineProcess	语法	public event LineProcessorHandler LineProcess

	前置条件	AI 程序正在思考，即将给出走子输出
	后置条件	这是一个异步事件，当 AI 进程给出走子输出时，处理该输出，并使得 Wait 能够继续运行直到返回
AIProcess.WriteLine	语法	public void WriteLine(string str)
	前置条件	AI 程序正在接受输入
	后置条件	将走子字符串输入给 AI 进程，使得 AI 进程能够了解对方的走子，进而轮到己方走子
需要的服务（需接口）		
除了.NET Framework 类库中的接口（较多且调用较频繁，故省略），没有需要调用的外部接口。		

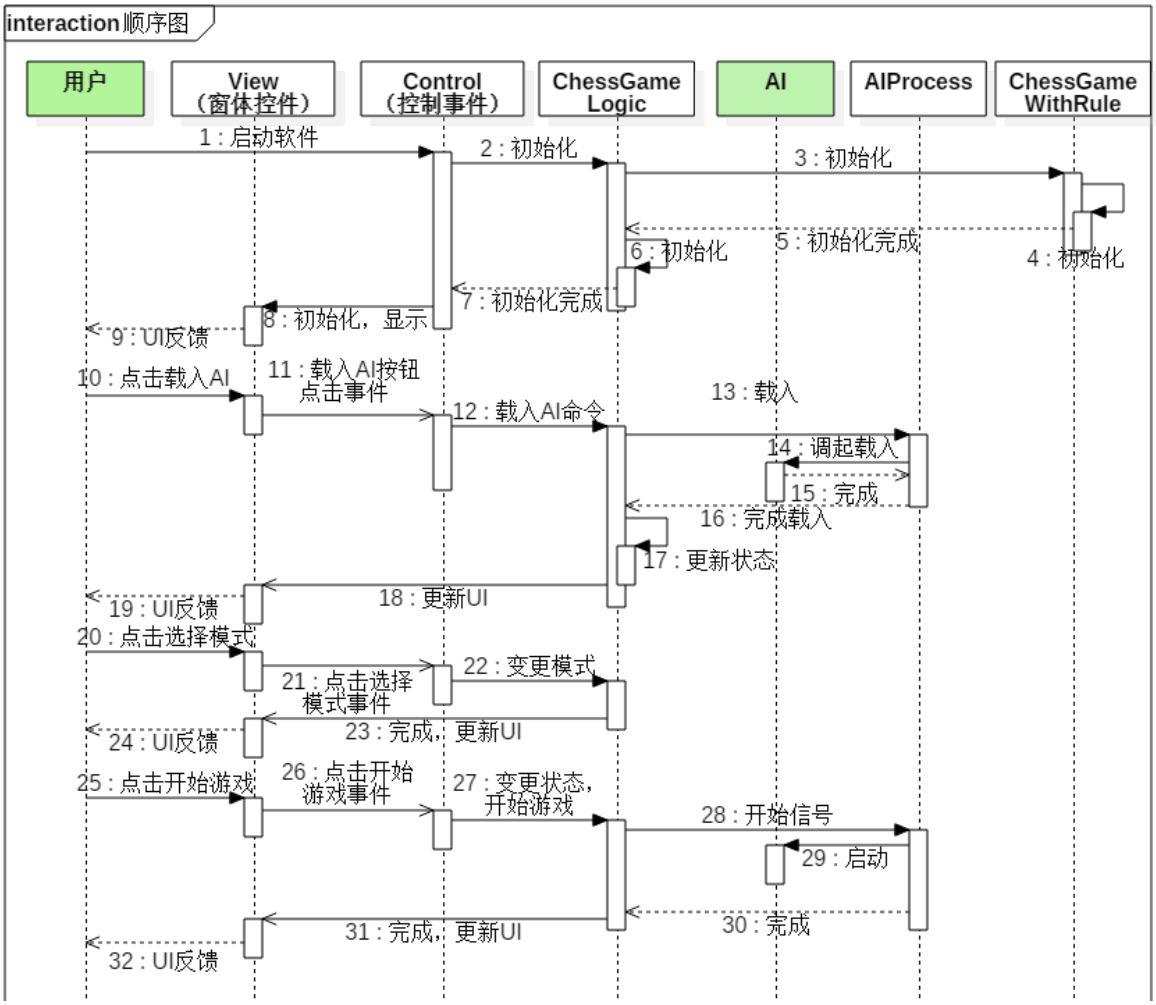
ChessGameLogic 类的接口规范

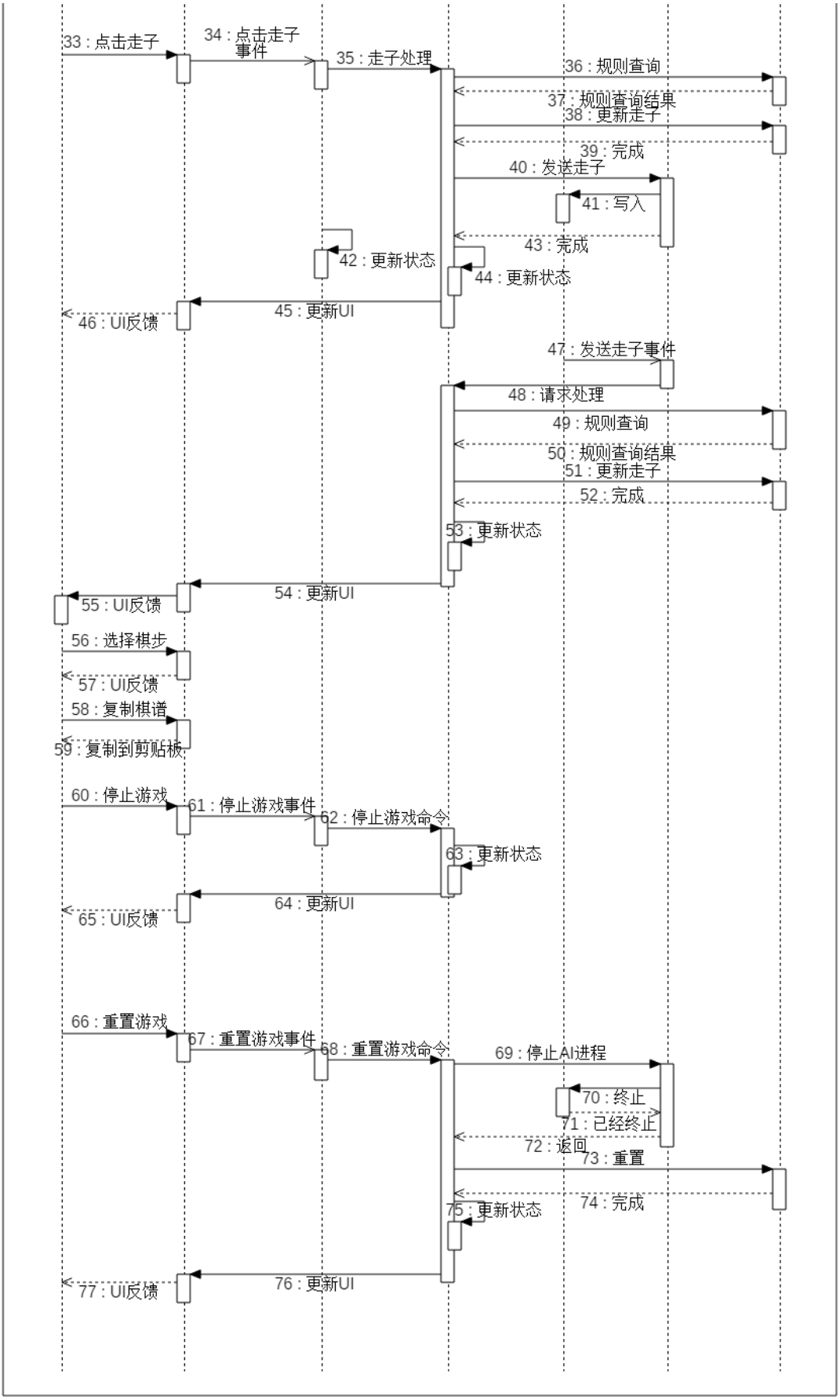
提供的服务（供接口）		
ChessGameLogic.Init	语法	void Init()
	前置条件	主窗体载入完毕，ChessGameLogic 对象完成构造
	后置条件	ChessGameLogic 对象完成初始化，可以进行游戏进程
ChessGameLogic.Start	语法	public void Start(GameMode mode)
	前置条件	主窗体载入完毕，ChessGameLogic 对象完成构造和初始化
	后置条件	以某一 GameMode 游戏模式开始游戏
ChessGameLogic.Stop	语法	public void Stop()
	前置条件	ChessGameLogic 对象正在进行游戏
	后置条件	停止游戏
ChessGameLogic.ResetGame	语法	public void ResetGame()
	前置条件	任何时候
	后置条件	重置游戏进程逻辑的所有数据

ChessGameLogic.*StatusUpdated	语法	public event *StatusUpdatedEventHandler *StatusUpdated
	前置条件	当游戏逻辑的状态更新时
	后置条件	这是一个异步事件，当游戏逻辑状态更新时，就调起订阅在其上的代理函数。通常情况下包括更新 UI 控件的操作。
ChessGameLogic.Set*Status	语法	public void Set*Status(ChessBoardGame*State state, bool updateImportant, string reason = null)
	前置条件	外部需要改变游戏逻辑的状态时
	后置条件	更新游戏逻辑的状态。
ChessGameLogic.ApplyMove	语法	public MoveType ApplyMove(Move move, bool alreadyValidated, out Piece captured)
	前置条件	游戏进程已经开始，需要应用一个走子
	后置条件	应用该走子，触发 AI 更新，并视情况触发接受下一步走子的新线程
ChessGameLogic.LoadAIExec	语法	public bool LoadAIExec(Player player, string execPath, string execArguments)
	前置条件	游戏进程尚未开始，需要载入 AI
	后置条件	载入该 AI，并验证其是否正常使用
需要的服务（需接口）		
.NET Framework 类库中的接口（较多且调用较频繁，故省略）以及上表中列出的所有 ChessGameWithRule、AIProcess 的接口。		

1.1.3 Model 的动态模型

由于本软件详细设计中的动态模型中，状态图的设计与软件结构是否分解为类关系不大，故动态模型中的状态图省略不画，请参考《需求规格说明书》中的状态图。现将《需求规格说明书》中进行一盘游戏的系统顺序图扩展为详细顺序图，描绘如下：

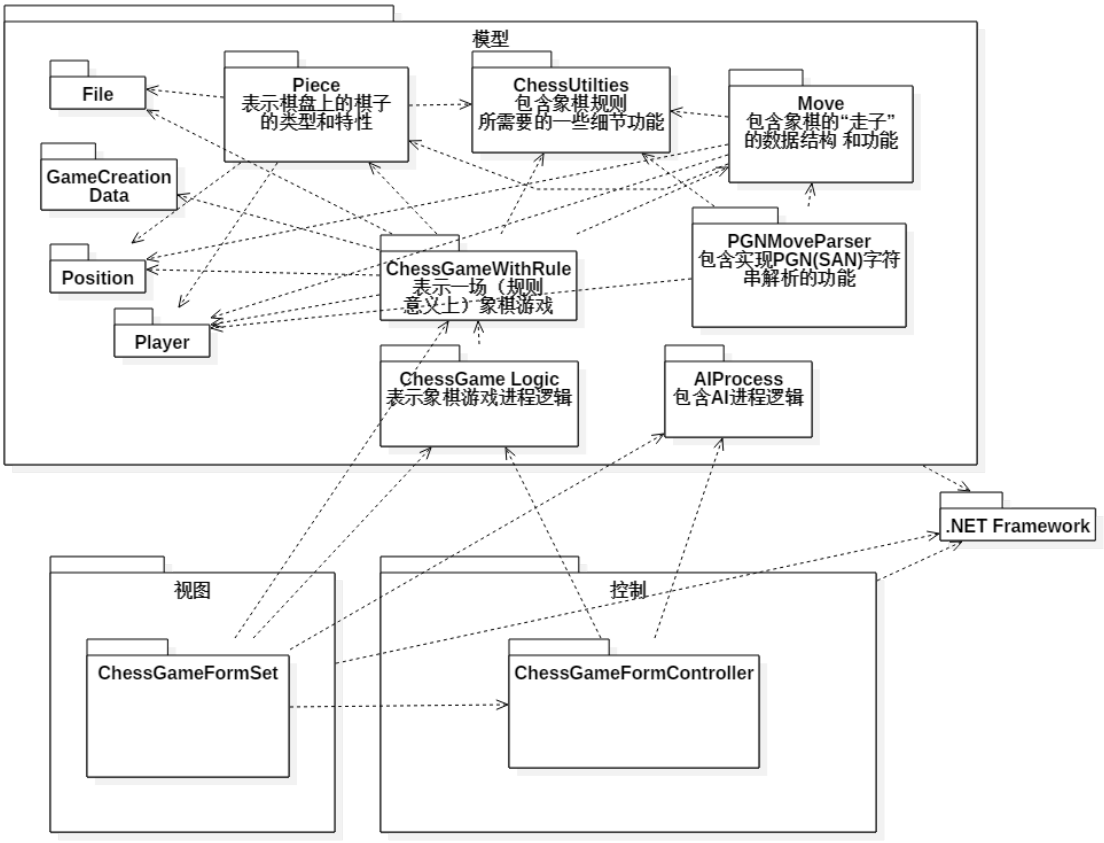

interaction顺序图
用户
View
（窗体控件）
Control
（控制事件）
ChessGame
Logic
AI
AIProcess
ChessGame
WithRule
1 : 启动软件
2 : 初始化
3 : 初始化
4 : 初始化
5 : 初始化完成
6 : 初始化
7 : 初始化完成
8 : 初始化，显示
9 : UI反馈
10 : 点击载入AI
11 : 载入AI按钮
点击事件
12 : 载入AI命令
13 : 载入
14 : 调起载入
15 : 完成
16 : 完成载入
17 : 更新状态
18 : 更新UI
19 : UI反馈
20 : 点击选择模式
21 : 点击选择
模式事件
22 : 变更模式
23 : 完成，更新UI
24 : UI反馈
25 : 点击开始游戏
26 : 点击开始
游戏事件
27 : 变更状态，
开始游戏
28 : 开始信号
29 : 启动
30 : 完成
31 : 完成，更新UI
32 : UI反馈


33 : 点击走子
34 : 点击走子事件
35 : 走子处理
36 : 规则查询
37 : 规则查询结果
38 : 更新走子
39 : 完成
40 : 发送走子
41 : 写入
42 : 更新状态
43 : 完成
44 : 更新状态
45 : 更新UI
46 : UI反馈
47 : 发送走子事件
48 : 请求处理
49 : 规则查询
50 : 规则查询结果
51 : 更新走子
52 : 完成
53 : 更新状态
54 : 更新UI
55 : UI反馈
56 : 选择棋步
57 : UI反馈
58 : 复制棋谱
59 : 复制到剪贴板
60 : 停止游戏
61 : 停止游戏事件
62 : 停止游戏命令
63 : 更新状态
64 : 更新UI
65 : UI反馈
66 : 重置游戏
67 : 重置游戏事件
68 : 重置游戏命令
69 : 停止AI进程
70 : 终止
71 : 已经终止
72 : 返回
73 : 重置
74 : 完成
75 : 更新状态
76 : 更新UI
77 : UI反馈

1.2 VIEW 的分解、CONTROL 的分解（同“体系结构设计”中原因，略）

2 依赖视角

详细设计中的包和依赖与体系结构设计中的逻辑包图类似，展示如下：



3 核心算法描述

3.1 GETVALIDMOVES 算法描述

ChessGameWithRule.GetValidMoves 是 Model 里象棋规则实现库 ChessGameWithRule 中的一重要函数，可以获得某一方的所有允许的走子。

它由两个同名重载函数实现。

3.1.1 public virtual ReadOnlyCollection<MoreDetailedMove> GetValidMoves(Position from, bool returnIfAny, bool careAboutWhoseTurnItIs)

这个函数可以将该行动方从 Position from 出发的所有可行走子返回。returnIfAny 和 careAboutWhoseTurnItIs 是内部使用和调试用参数，可以忽略。

其具体实现如下：

```
public virtual ReadOnlyCollection<MoreDetailedMove> GetValidMoves(Position from, bool returnIfAny, bool careAboutWhoseTurnItIs)
        {
            ChessUtilities.ThrowIfNull(from, "from");
            Piece piece = GetPieceAt(from);
            if (piece == null || (careAboutWhoseTurnItIs && piece.Owner != WhoseTurn)) return new ReadOnlyCollection<MoreDetailedMove>(new List<MoreDetailedMove>());
            return piece.GetValidMoves(from, returnIfAny, this, IsValidMove);
        }
```

描述为自然语言：

- 获取当前棋盘 from 位置的棋子为 piece
- 如果那个位置没有棋子
  - 直接返回空集合
- 否则
  - 调用这个具体棋子的多态函数 GetValidMoves，给出从 from 出发的所有可行走子。同时，还传入当前 ChessGameWithRule 对象下的，对于本局游戏特化的走子验证函数，方便 Piece.GetValidMoves 调用用于二次验证走子的合法性。

3.1.2 public virtual ReadOnlyCollection<MoreDetailedMove> GetValidMoves(Player player, bool returnIfAny, bool careAboutWhoseTurnItIs)

这个函数可以将该行动方（不受 Position 制约）的所有合法走子返回。

```
public virtual ReadOnlyCollection<MoreDetailedMove> GetValidMoves(Player player, bool returnIfAny, bool careAboutWhoseTurnItIs)
        {
            if (careAboutWhoseTurnItIs && player != WhoseTurn) return new ReadOnlyCollection<MoreDetailedMove>(new List<MoreDetailedMove>());
            List<MoreDetailedMove> validMoves = new List<MoreDetailedMove>();
            for (int r = 1; r <= Board.Length; r++)
            {
                for (int f = 0; f < Board[8 - r].Length; f++)
                {
                    Piece p = GetPieceAt((File)f, r);
                    if (p != null && p.Owner == player)
                    {
                        validMoves.AddRange(GetValidMoves(new Position((File)f, r),
```

```
returnIfAny));
                        if (returnIfAny && validMoves.Count > 0)
                        {
                            return new
ReadOnlyCollection<MoreDetailedMove>(validMoves);
                        }
                    }
                }
            }
            return new ReadOnlyCollection<MoreDetailedMove>(validMoves);
        }
```

描述为自然语言：

- 初始化 validMoves 为空集
- 对于棋盘上的所有格子（位置 Position）作一遍历：
  - 获得该格子上的棋子为 p
  - 若格子上有棋子（p 非空）且 p 的拥有者是 player：
    - 调用 3.1.1 中同名函数 GetValidMoves，获取从该位置出发的所有合法走子
    - 将 validMoves 并上刚刚返回的走子集合
- 返回 validMoves

3.2 IsMoveValid 算法描述

这个函数同样属于象棋规则库。它判断某一个走子在当前游戏的进行情况中是不是合法的，这对于合法走子的生成和判定非常重要。

```
public virtual bool IsValidMove(Move move, bool validateCheck, bool
careAboutWhoseTurnItIs)
        {
            ChessUtilities.ThrowIfNull(move, "move");
            if (move.OriginalPosition.Equals(move.NewPosition))
                return false;
            Piece piece = GetPieceAt(move.OriginalPosition.File,
move.OriginalPosition.Rank);
            if (careAboutWhoseTurnItIs && move.Player != WhoseTurn) return false;
            if (piece == null || piece.Owner != move.Player) return false;
            Piece pieceAtDestination = GetPieceAt(move.NewPosition);
            bool isCastle = pieceAtDestination is Rook && piece is King &&
pieceAtDestination.Owner == piece.Owner;
```

```
            if (pieceAtDestination != null && pieceAtDestination.Owner == move.Player
&& !isCastle)
            {
                return false;
            }
            else if(move is MoreDetailedMove m)
                if (pieceAtDestination != null && pieceAtDestination.Owner ==
ChessUtilities.GetOpponentOf(move.Player))
                {
                    m.IsCapture = true;
                    m.CapturedPiece = pieceAtDestination;
                }
            if (!piece.IsValidMove(move, this))
            {
                return false;
            }
            if (validateCheck)
            {
                if (!isCastle && WouldBeInCheckAfter(move, move.Player))
                {
                    return false;
                }
            }

            return true;
        }
```

自然语言描述如下：

- 若 move 为空
  - 返回 false
- 获取 move 的源位置的行、列，并以此获取该位置的棋子到 piece
- 若 piece 为空或不是本方棋子
  - 返回 false
- 获取 move 的目标位置的，并以此获取目标位置的棋子到 pieceAtDestination
- 对于 move 是王车易位的情况作特殊判断，若王车易位不合法：
  - 返回 false
- 调用该 piece 的 IsValidMove 函数，获知该棋子能否如此移动。若不能：
  - 返回 false

- 若该走子非王车易位，且走子后会陷入将军：
  - 返回 false
- 返回 true。

第三部分 编码与测试

1 软件编码

根据“软件详细设计”中设计的各种职责的类，可以将其实现为 C#代码：

- 编写各个 Model 类的属性
- 编写各个 Model 类的方法
- 绘制 Windows 窗体，实现 View
- 用 Visual Studio 的可视化方法直接从 View 的控件中双击，自动生成 Control 函数
- 在 Control 函数中操作 Model 中的类
- 将 Model 类中的属性绑定到 View 的相应控件中

代码见随附的 C#源码文件和工程文件。

2 软件测试

由于本软件体量较小，可以随时编译并运行，故可以在每次修改代码后进行一次系统、功能测试。该测试可以注重正在编写的代码的行为，也可以注重整个系统运行的正确。其测试目标为观察到的行为要符合软件规格说明文档中的说明。

本软件在开发过程中进行测试，每当一个类方法通过测试后，才会继续开发。在这同时进行的过程中，整个系统编写完成后，基本上初步通过了开发人员测试。

开发出来的软件分发给《人工智能原理》班级的参赛者进行公共测试。经公共测试，该软件的功能无异常，并且符合《需求规格说明书》中的要求。在 13-14 周，《人工智能原理》课程如期举行国际象棋 AI 比赛，比赛顺利进行。因此，本软件的测试和维护工作便已经告一段落。