Unnatural Gaming

Checkmate

Design Specifications

Revision #: 02
Revision Date: October 2nd, 2003

Table of Contents

1. Architecture

1.1 Deployment

To review from the requirements document[1], the distribution of components is as follows:

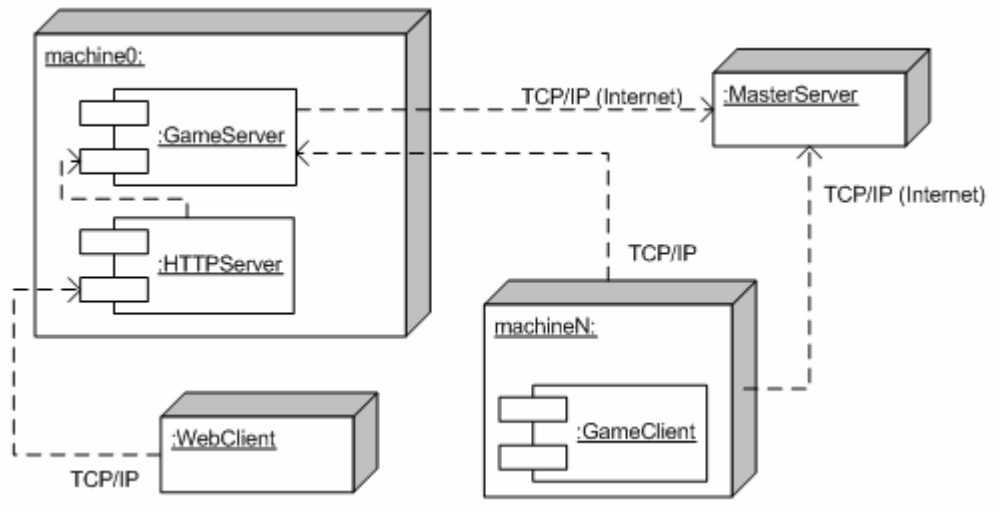


Figure 2: Deployment of Checkmate

Interesting components

The two components that we are modifying are the GameServer and GameClient.

GameServer: The authoritive component that maintains the true instance of the game and serves the clients.

GameClient: Runs an interpretation of the game on its own process. The client runs a proxy version of the game which attempts to simulate and predict what is actually happening on the server.

The GameServer and GameClient both run their own instance of the same executable, but as different roles. (authoritive vs. proxy). The authoritive instance regularly updates the proxies with the game state. The server must also maintain and prioritize what is most relevant to the clients, so that when bandwidth is limited it can send what is most important.

[1] Unnatural Gaming, "Requirement Document", March 3rd, 2003.

1.2. Packages

We are producing two types of packages:

- Multimedia package
- Unreal module[2]

The first type of package contains media content such as textures, 3-D models, or sounds. Multimedia content was defined to be outside the scope of for the project, so this type of package will not be discussed further.

The second type of package, an Unreal module, is where the Checkmate game is located. A single module (Checkmate.u) is built and deployed to both server and clients[3]. **Figure 1** shows the packages that Checkmate depends on.

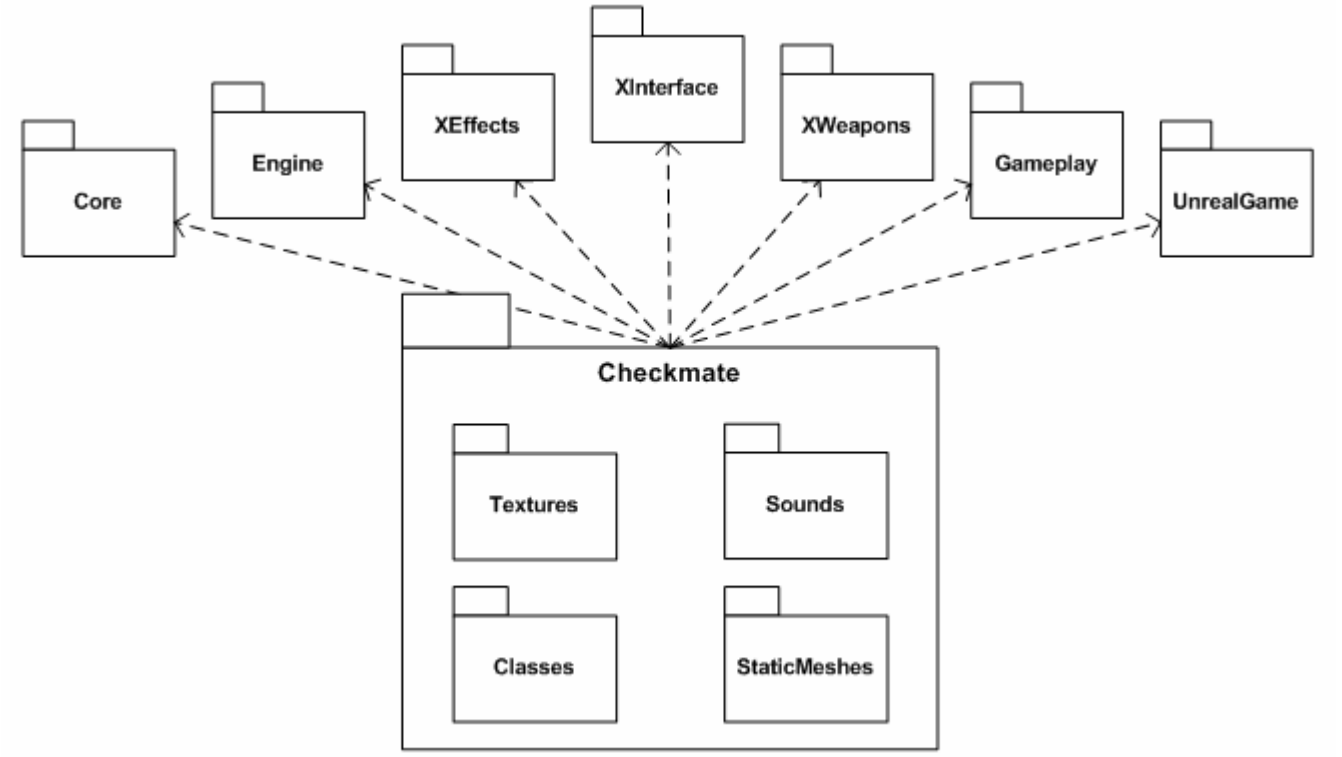


Figure 1: Package dependencies of Checkmate

All these packages are published by Epic and are shipped with UT2003, so Checkmate's only dependency is the game itself (and the latest patch).

A description of what we are using from these packages is described below.

Core

- *Object* – the base class for all objects; defines the ability for classes to change states; provides native utility methods (i.e. Math functions)

[2] An unreal module is library file containing UScript byte-code, it is valid for any platform that can run UT2003 (Windows and Linux).

[3] By deploying the same module to both client and server, the server is able to compute a checksum to verify with reasonable accuracy that the client has not modified the module.

Engine

- *Actor* – the base class for all gameplay objects that can be spawned into a game; some operations implemented by Actor include animation, physics, producing sounds, actor creation and destruction, triggers and timer functions, message broadcasting.

- *Pawn* – the base class for all actors that can be controlled by a player or bot. They are the physical representation of players in a level, they have a mesh, collision, and physics.

- *Controller* – the base class for players and bots (AI); a non-physical actor that can be attached to a pawn to control its actions. *PlayerController*s are used by human players while *AIControllers* are used by the server to implement artificial intelligence controlling the pawns.

- *Info* – The base class for all classes holding information about a specific game element. Information in these classes are usually replicated to all the clients.

- *GameInfo* – defines the game being played including the game rules, scoring, what pawns are created for the players. Game specifics are implemented in the subclasses. A GameInfo actor is instantiated when a level is loaded based on the desired gametype specified by the server administrator.

- *Hud* – the base class for a "heads-up-display". The hud contains the visual components seen by the player as they play the game indicating their current status in the game (such as health, ammo).

- *PlayerReplicationInfo* – contains information about a given player that should be replicated to all other clients. Examples include score and name.

- *TeamInfo* – contains information about a particular team in the game. Although Checkmate has two teams, any number of teams could theoretically exist.

- *Weapon* – the base class for a weapon usable by a player. Every weapon has one ore more *WeaponFires* which describe the result of firing the weapon. A *WeaponFire* usually produce hit-scan[4] damage or fires a projectile which inflicts damage when a collision occurs.

XEffects

This package contains many visual effects which are used in Checkmate.

XInterface

This package contains the UI library which we use to build UI components such as the heads-up-display, in-game menus, and the scoreboard.

XWeapons

The chaingun and the sniper rifle are both derived from this package.

Gameplay

The Gameplay package contains the classes that define player interactions and environmental effects.

[4] Hit-scan refers to an operation that occurs instantly using a tracer with infinite velocity.

UnrealGame
UnrealGame contains concrete implementations of the game elements defined in the Engine package. To avoid duplicating work, most of our implementations extend the classes defined in this package. For example, *Checkmate* extends *TeamGame* which is a concrete subclass of *GameInfo* which already handles separating players on two separate teams.

1.3 Purpose of the Checkmate Module

The Checkmate module is responsible for providing the complete functionality of the game for both the server component and the client component. This includes server side operations such as team scoring and client side operations such as displaying the user interface.

A single module is used for both server and client because of the interdependent nature of playing a game on client-server architecture. A server may spawn numerous game objects, all of which are implicitly replicated to the clients. A brief overview of how this replication works in unreal script is provided below.

1.3.1 Overview of Unreal Script Replication

Unreal Script has the mechanisms built into the language which allow code to be written once, and duplicated on both the server and client.

Types of Replication

- **Actor Replication** – This occurs implicitly whenever an actor is spawned. The server tells the client to spawn an actor of a given class and replicates the default properties. Actors are also replicated when they become relevant to the client (i.e. when the object comes in view).

  Example: the server spawns a rocket projectile, it is implicitly replicated to clients.

  Code Snippet:
  ```
  // when this code runs on the server, the rocket is automatically created
  // on the clients.
  Projectile p = spawn(class'RocketProj', , , initialLocation, direction);
  ```

- **Variable Replication** – An actor's variables can be replicated when their values change. Variables must be explicitly replicated by specifying a replication condition on the variable. Every time the variable changes value, the replication condition is checked and the client is instructed to change the variable to the new value.

  Variables can be marked for reliable or unreliable replication. Reliable requires more bandwidth and is guaranteed to update whereas unreliable uses less bandwidth but is not guaranteed to update (this is more appropriate for frequent updates such as the location of an actor).

  Example: the server updates a player's credits to all clients when it is updated.

  Code Snippet:
  ```
  // class variable definition:
  var int credits;
  ```

```
…
// replication block
replication {
       reliable if (role == Role_Authority) // this conditions means that the
              credits;                      // current role is the server.
}
…
// variable update
credits += reward; // when this code runs on the server, the value is
                   // automatically replicated to all clients
```

- **Function Replication** – The server can instruct a client to run a function locally. Likewise, a client can instruct the server to run a function server-side. These functions must be marked explicitly for remote calling in the replication block similar to variable replication.

  Unlike variable replication, function replication can only be called to the owner of the actor containing the function. For a function to run on the client side it must also be marked as *simulated*; functions not marked as *simulated* will not run even if replicated. Function replication is also always reliable and is always guaranteed to run remotely.

Example: a client calls a function to change player class, it is invoked by the client but runs on the server.

Code Snippet:

```
// the function is defined like any other function, in this case 'exec'
// indicates that a player can explicitly call the function.
exec function selectClass(string className) {
       //implementation is not important
}

replication {

       // these functions can be called by a client and run on the server
       reliable if (role < ROLE_Authority) // this condition means that the current
              selectClass;                 // role is not the server
}
```

1.4 Checkmate API

Although Checkmate is not intended to be extended by other Unreal modules, the API for the core components is available in Appendix A for internal documentation purposes.

2. Design

2.1 Design of the Game Component

The core classes used to implement the game are:

- **Checkmate –** the game rules.
- **CMPlayer** – a player playing Checkmate, provides commands for selecting classes and using abilities.
- **CMBot** – a computer controlled player playing Checkmate.
- **CMPlayerReplicationInfo** – holds information made available to other players.
- **CMTeamInfo** – tracks the current state of a team, handles assigning the King
- **CMPawn** – the base class for all player controlled classes (PonPawn[5], KnightPawn, BishopPawn, RookPawn, QueenPawn, KingPawn).

To see how these core classes fit into the existing class hierarchy, see **Appendix B.1**. Additional class information is provided for the player class system in **Appendix B.2**. This hierarchy includes the special abilities, the weapons, and the player classes included in the game.

2.2 Design of the UI Component

HUD (Heads Up Display): The widgets on the players screen viewed when playing the game. These can include player's health, weapon, ammo, score, etc.

The in-game UI is the main component in which the user is given information about their class. For example the interface tells the user:

- Amount of health
- Amount of ammo
- Radar
- Their next class
- Number of frags
- Number of credits
- Score

Also the user must interact using the UI when selecting a class. This is done through a menu that pops up upon death or on key press.

The classes used to implement the in-game UI are:

- CMClassSelectionMenu – menu that allows players to purchase new classes based on how many credits they have.
- CMHud – the generic features of the HUD which include the player's health.
- CMKingHud – the King specific HUD features (ex: King's Radar).
- CMGeneralHUD – the HUD for all player classes other than the King.
- CMScoreboard – shows the standings of the current game. Other than the King (who is always on top of the scoreboard) the standings are ranked by score.

[5] Since the term 'Pawn' is used in the context of Unreal to refer to any player-controlled object, 'Pon' is used to refer to the chess piece 'Pawn'. Although the distinction between the two is still somewhat confusing, using a different name for the Chess instance helps reduce this confusion.

Check the status report for screenshots. To see how these GUI classes fit into the existing class hierarchy, see **Appendix B.3**.

2.3 Interactions of Classes

The following interaction Diabgarms (found in Appendix C), correspond with Use Cases previously defined in the Checkmate Analysis Report

Appendix C.1 Interaction – Player Connection
Use Cases Addressed: Use Case 2 – Players Join a Server, Use Case 4 – Select Player Class
Requirements Addressed: R8, R9, R10, R11, R12, R13, R14, R21, R23

Appendix C.2 Interaction – Player Killing
Use Cases Addressed: Use Case 3 – Players Accumulate Points
Requirements Addressed: R13, R21

Appendix C.3 Interaction – Killing the King
Use Cases Addressed: Use Case 5 – Fragging Opposing Team's King
Requirements Addressed: R10, R13, R15, R18, R21

Appendix C.4 Interaction – Selecting a New Player Class
Use Cases Addressed: Use Case 6 – The Match Ends
Requirements Addressed: R10, R13, R15, R17, R21

Appendix A: API of Core Classes

Class: Checkmate.Checkmate

```
//=============================================================================
// Written by: Unnatural Gaming, 2003
// Description: Starting point for the Checkmate gametype. Contains rules for
// team scoring based on killing a king.
//=============================================================================
class Checkmate extends TeamGame;
```

Functions:

```
/**
 * Determine the value of awarded credits based on killer's and victim's player
 * class.
 *
 * killer - The controller who committed the kill.
 * victim - The controller who was killed.
 *
 * return - the value to award for the kill.
 */
function int awardCredit(Controller killer, Controller victim);

/**
 * Team scoring, reward team for killing the other team's king.
 *
 * checkmatedTeam - the team who's king was killed.
 */
function scoreCheckmate(CMTeamInfo checkmatedTeam);

/**
 * Updates the health of each king based on the number of players on each team.
 * A team with more players would be handicapped for outnumbering a team with
 * less players.
 */
function updateKingHealth();

/**
 * Assign a chess class to a player. Defaults to the Pawn class if the player
 * does not have sufficient credits to purchase the class.
 *
 * aPlayer - the player to set this new class for.
 * className - the fully qualified class name to set to, must be a subclass of
 *             Checkmate.CMPawn.
 */
function setPlayerClass(Controller aPlayer, string className);
```

Additional States:

```
/**
 * Intermission state, players observe the assassinated king and wait for the
 * next round.
 */
State Intermission;
```

Class: Checkmate.CMPlayer

```
//-------------------------------------------------------------
// Written by: Unnatural Gaming, 2003
// Description: Additional attributes and operations for
//              Checkmate player controllers.
//-------------------------------------------------------------
class CMPlayer extends xPlayer;
```

Replication:

```
replication {
    // Functions the client calls on the server.
    reliable if(Role<ROLE_Authority)
        special, selectClass;

    // Functions the server calls on the client side.
    reliable if(remoteRole == ROLE_AutonomousProxy)
        clientGotoIntermission;
}
```

Functions:

```
/**
 * Called by a client indicating to use their special ability
 */
exec function special(optional float f);

/**
 * Allows a player to select a class from the console. This method runs on the
 * server.
 *
 * className - the shortname of the class to select (i.e. Pawn, Knight, etc...)
 */
exec function selectClass (string className);

/**
 * Displays a menu allowing the client to select a class.
 */
exec function showClassSelectMenu();

/**
 * Change this player's state to 'Intermission'.
 */
function gotoIntermission();

/**
 * Instruct the client to change state to 'Intermission'.
 */
function clientGotoIntermission();
```

Class: Checkmate.CMPawn

```
//=============================================================================
// Written by: Unnatural Gaming, 2003
// Description: abstract Pawn for Checkmate Pawns (King, Queen, Rook, Bishop,
//              Knight, Pawn).
//=============================================================================
class CMPawn extends xPawn Abstract;
```

Attributes:

```
// the heads-up-display for the player class
var const string hudType;

// the weighted point value for the player class
var const float  cmValue;

// the player record specifying the species and model of the pawn
var const xUtil.playerRecord cmPlayerRecord;

// the class to spawn for the special ability
var const class<SpecialAbility> specialAbilityClass;
```

Attribute Defaults:

```
defaultproperties {
    hudType = "Checkmate.CMGeneralHud"
    requiredEquipment(0) = "Checkmate.PulseRifle"
    requiredEquipment(1) = ""
    controllerClass = Class'Checkmate.CMBot'
    cmValue = 1
    shieldStrengthMax = 0
    specialAbilityClass = none;
}
```

Replication:

```
replication {
    reliable if(Role == ROLE_Authority && bNetOwner)
        specialAbility;
}
```

Class: Checkmate.SpecialAbility

```
//=============================================================================
//Written by: Unnatural Gaming, 2003
//Description: Abstract class for special abilities. Provides operations for
//             activating and aborting the ability, and a client function to
//             check the cooldown on the ability.
//
//             Subclasses should overload the methods startSpecial,
//             cancelSpecial, and endspecial making sure to call the super
//             implementations.
//
//             An ability has two states and three triggers.
//             (STATE 1)  : The skill is inactive.
//             (TRIGGER 1): The skill is activated if the cooldown has expired.
//             (STATE 2)  : The skill is active and delaying to terminate, some
//                          skills are 'consumed' during this state while others
//                          can be fired off at the end of this state.
//             (TRIGGER 2): The skill has terminated, either consumption ends
//                          or the skill is finally triggered.
//=============================================================================class
SpecialAbility extends Actor abstract;
```

Attributes:

```
// time between uses (in seconds)
var const float COOLDOWN;

// delay before special finishes or triggers
var const float DELAY;

// the initial charge percent of this ability (0.0-1.0)
var const float INITIAL_CHARGE;
```

Attribute Defaults:

```
defaultproperties {
    COOLDOWN = 60.0
    DELAY = 10.0
    INITIAL_CHARGE = 1.0
}
```

Functions:

```
/**
 * Try to activate the special ability. By default, the skill can be used if
 * the cooldown delay has elapsed (which is true initially by default).
 */
function useSpecial();

/**
 * Abort the ability. In the initial state this does nothing since the skill
 * is inactive.
 */
function abortSpecial();

/**
 * Answers the percentage of the cooldown or delay. When the skill is cooling
 * down (waiting to activate),
 *
 * return - when the skill is cooling down (waiting to activate), the percent
 *          of the cooldown percent is returned; when the skill is delaying to
 *          finish (could mean the skill is active or delaying to activate),
```

```
 *          the delay percent is returned.
 */
simulated function float getChargePercent();
```

Additional States:

```
/**
 * The ability is active. This could mean the skill is being delayed before it
 * actually triggers, or it could mean the skill is being used up currently.
 * Interpretation of the meaning 'active' varies amoung subclasses.
 */
state AbilityActive {

    /**
     * This implementation will abort the skill and return back to the initial
     * state.
     */
    function abortSpecial();
}
```

Class: Checkmate.CMPlayerReplicationInfo

```
//-----------------------------------------------------------
// Written by: Unnatural Gaming, 2003
// Description: Additional Checkmate replication info for
//              players.
//-----------------------------------------------------------
class CMPlayerReplicationInfo extends xPlayerReplicationInfo;
```

Data Definitions:

```
// enumeration specifying the different player classes
enum EPlayerClass {
    CM_PAWN,
    CM_KNIGHT,
    CM_BISHOP,
    CM_ROOK,
    CM_QUEEN,
    CM_KING
};
```

Attributes:

```
// player's current credits
var int credits;

// player’s kills, same as defined in PlayerReplicationInfo,
// but this value is replicated.
var int cmKills;

// player's current player class
var EPlayerClass playerClass;
```

Replication:

```
replication {
    //things the server should send to the client.
    reliable if ( bNetDirty && (role == Role_Authority) )
        credits, cmKills, playerClass;
}
```

Class: Checkmate.CMTeamInfo

```
//=============================================================================
// Written by: Unnatural Gaming, 2003
// Description: Team info for Checkmate. Also contains operations that deal
//              with a specific team.
//=============================================================================
class CMTeamInfo extends xTeamRoster;
```

Attributes:

```
// a king exists on this team
var bool        bKingExists;

// a king has disconnected in the current round
var bool        bKingDisconnected;

// the controller controlling the team's king
var Controller  kingController;

// the KingPawn instance of the team's king
var KingPawn    teamKing;
```

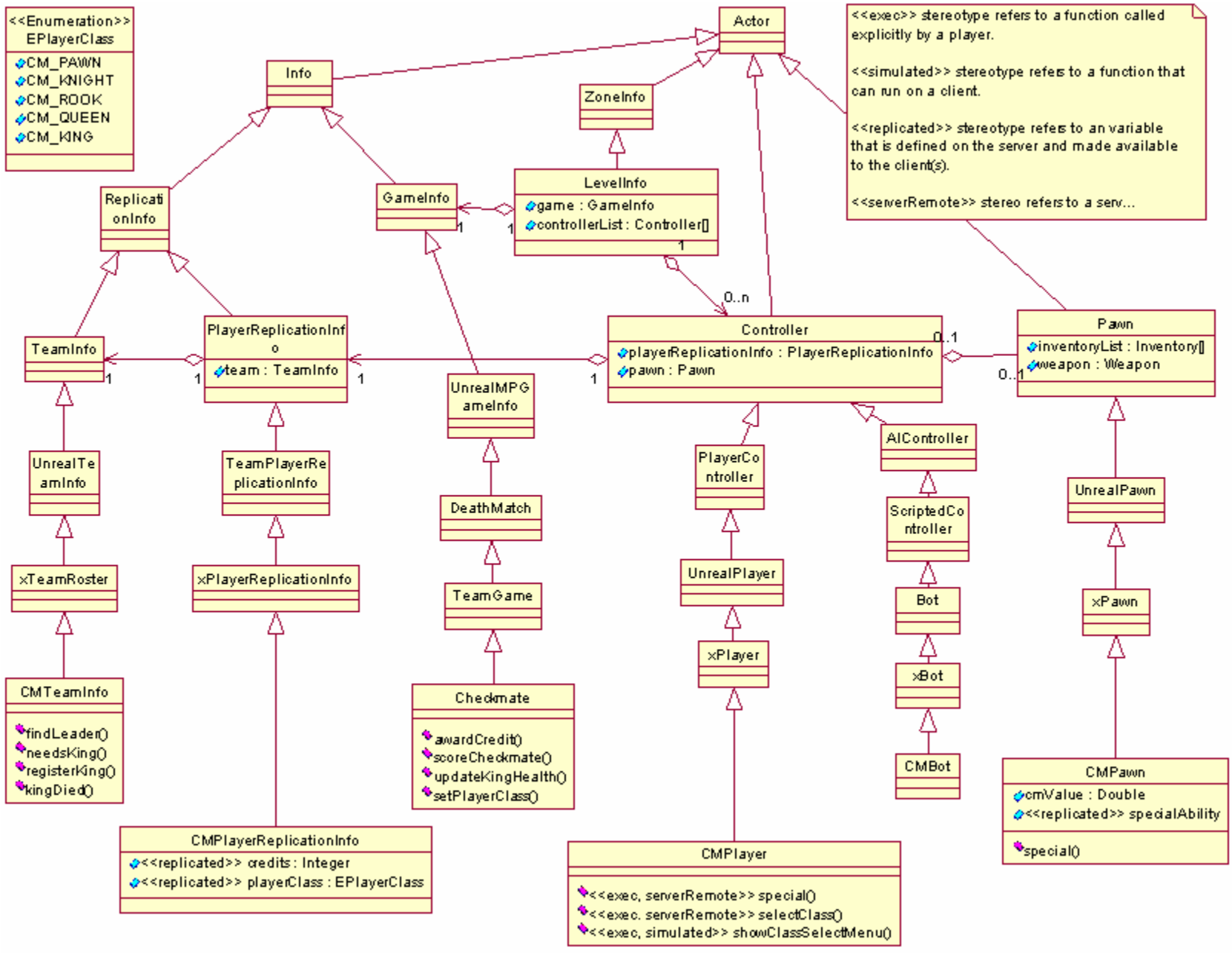
Attribute Defaults:

```
defaultproperties {
    bKingExists = false;
}
```

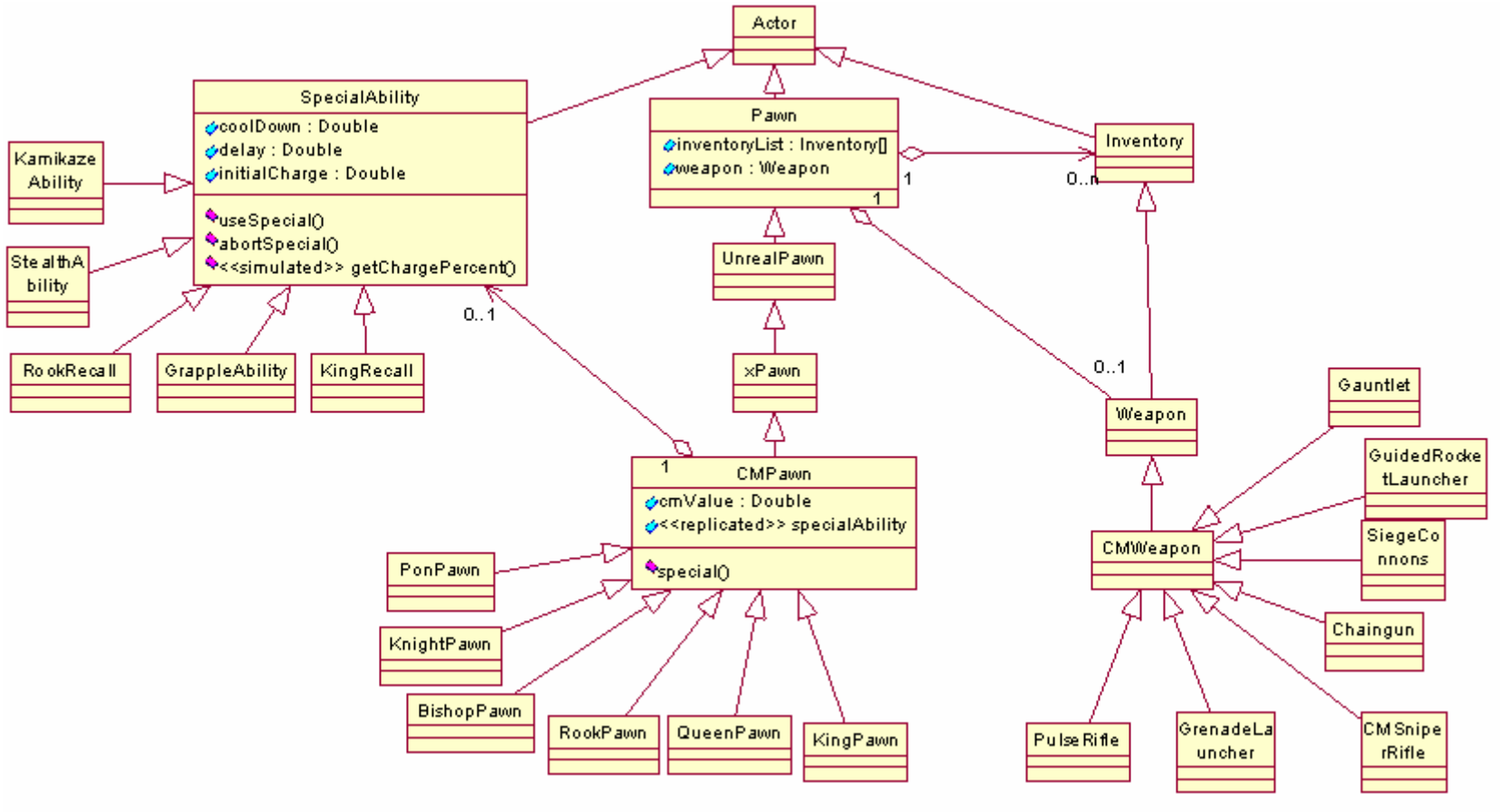
Functions:

```
/**
 * Find the player on the team with the highest score ignoring a given player.
 *
 * ignorePlayer - this player will be ignored when looking for the leader, can
 *                be 'none' to ignore no players.
 *
 * return - the Controller with the highest score.
 */
function Controller findLeader(Controller ignorePlayer);

/**
 * Answers whether this team needs a king.
 *
 * return - true if the team needs a new king, false otherwise.
 */
function bool needsKing();

/**
 * Registers a player's KingPawn to the team. Should only be called after a new
 * KingPawn has been spawned (on player restart).
 *
 * aPlayer - the player to register as this team's new king.
 */
function registerKing(Controller aPlayer);

/**
 * Notifies the team that their king has died. Does nothing if the dead king
 * does not match up with the team's registered king.
 *
 * deadKing - the King that has died.
 */
function kingDied(KingPawn deadKing);
```

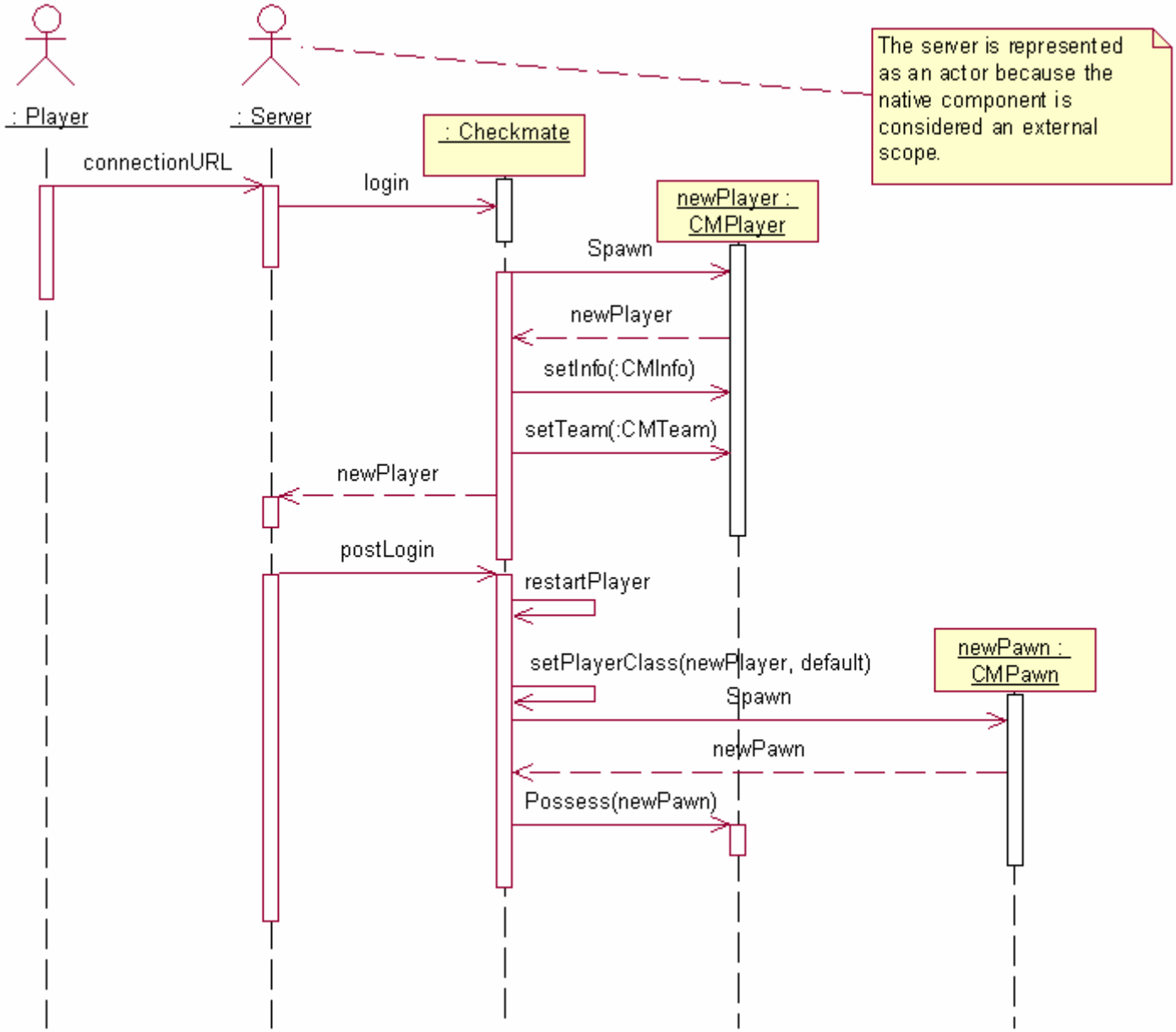
Appendix B.1: Core Class Hierarchy



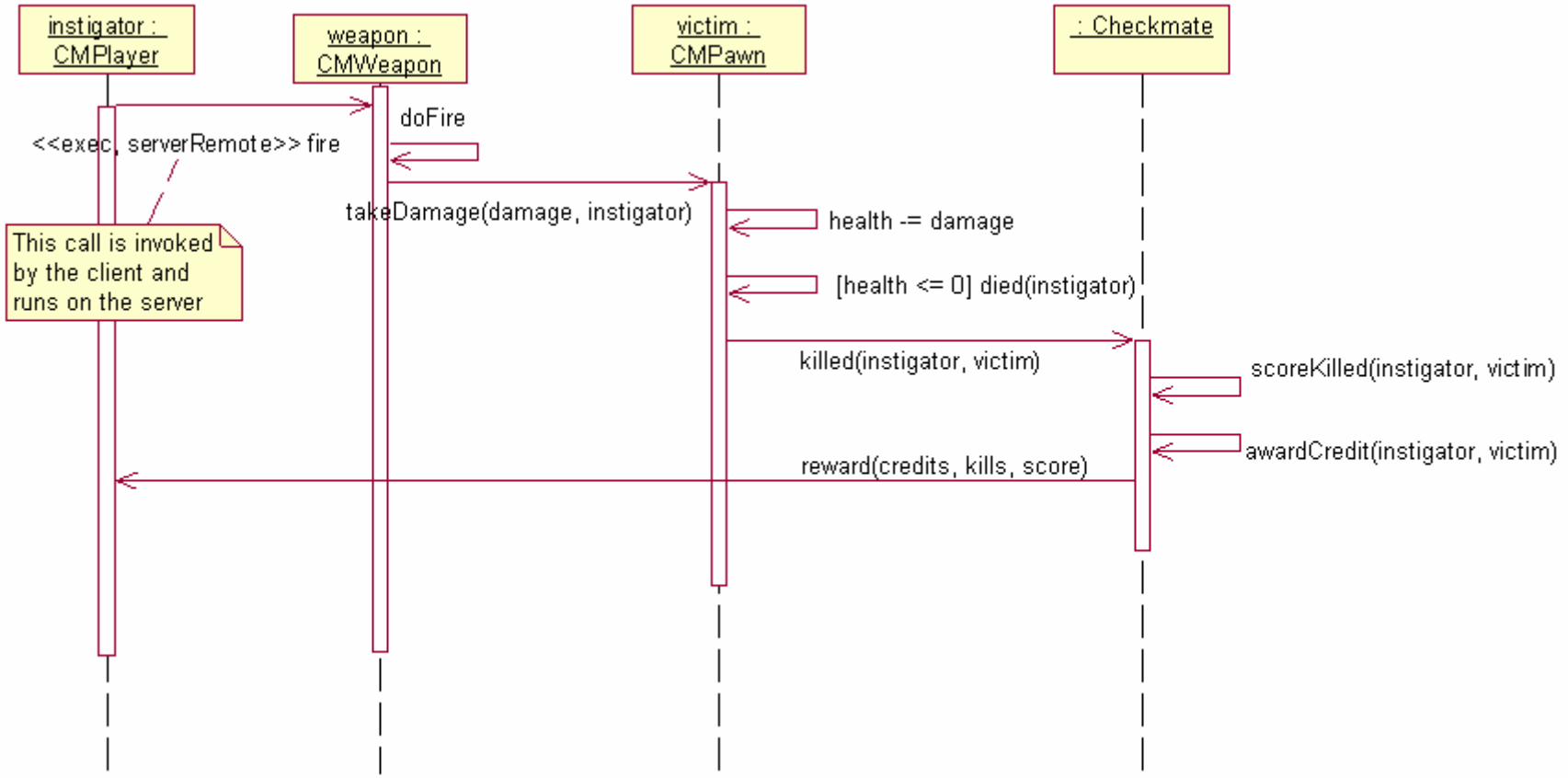
Appendix B.2: Chess-Class, Classes



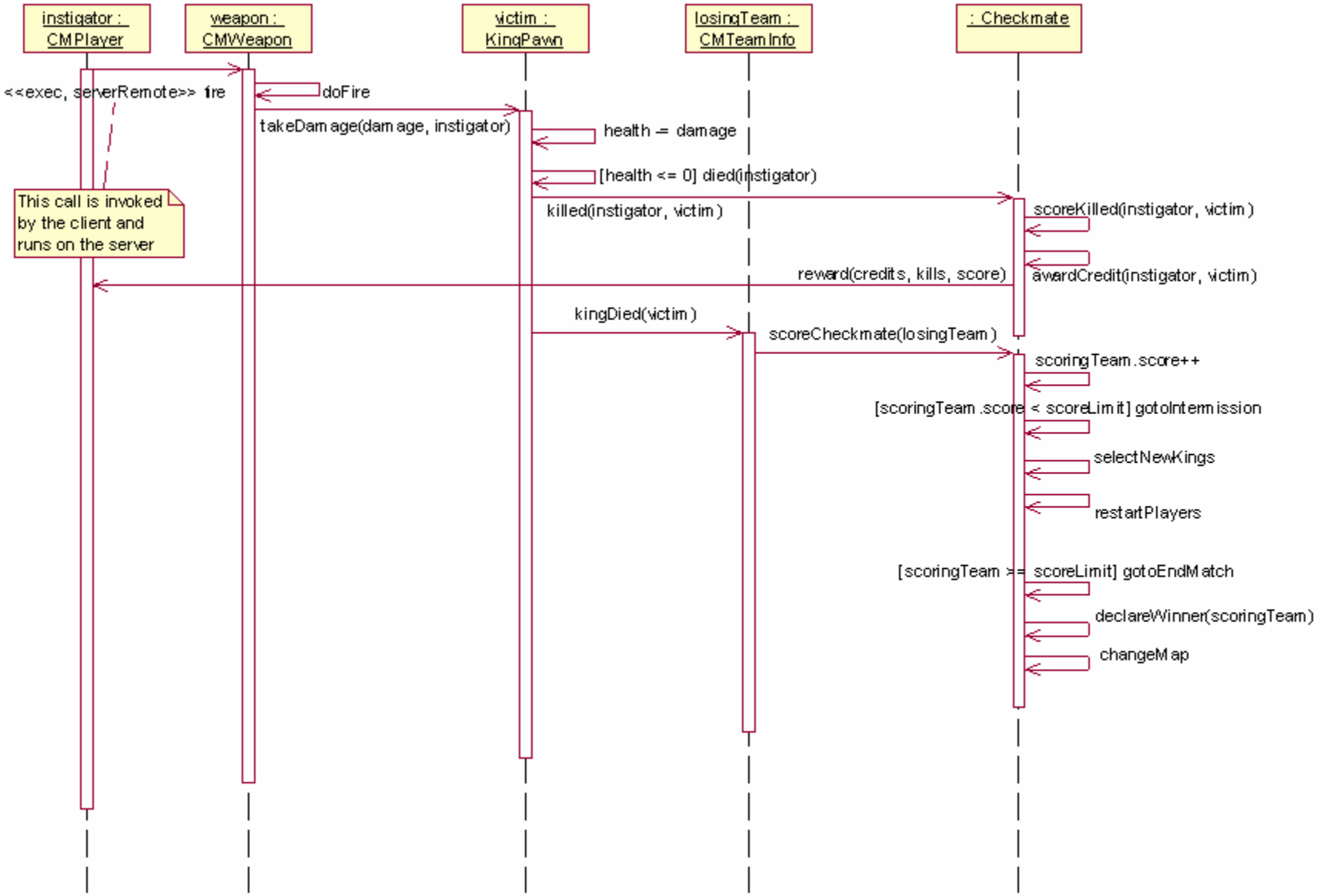
Appendix C.1: Interaction – Player Connection



Appendix C.2: Interaction – Player Killing



Appendix C.3: Interaction – Killing the King



Appendix C.4: Interaction – Selecting a New Class Upon Death