Univerisità degli Studi di Padova
Facoltà di Scienze MM. FF. NN.
Sistemi Concorrenti e Distribuiti

Software per Simulare Match Calcistici

Autore: dott. Paolo Carletto

Matricola: 622276

Docente: dott. Tullio Vardanega

A.A. 2011/12

03 gennaio 2011

Indice

1 Introduzione

1.1 Tema

Il tema del progetto è un simulatore software di una partita di calcio.

Il sistema software deve supportare almeno le seguenti caratteristiche:

- Gli utenti devono essere in grado di giocare un singolo match; il supporto per la giocabilità di una serie di partite con calendari e regole associate è facoltativo e può essere omesso;
- La variante scelta del gioco deve essere configurabile in tutti i parametri rilevanti, consentendo uno dei formati classici del calcetto a 5, 7 e il canonico 11 gicatori;
- I componenti delle squadre saranno caratterizzati da caratteristiche configurabili individualmente almeno per capacità tecniche e tattiche, velocità, parametri fisici tra cui anche la stanchezza; alcuni di questi parametri devono essere dinamici ed evolvere con la partita secondo una logica programmata;
- Ogni squadra deve avere un manager (software) in grado di configurare la formazione iniziale dei giocatori, la tattica iniziale e di impartire comandi per cambiamenti tattici e sostituzioni, tutti soggetti alle regole del gioco come definito nello standard corrispondente;
- Ogni squadra deve giocare secondo la tattica comandata dal manager; variazioni sono ammesse nella misura in cui risultano dalle caratteristiche programmabili dei giocatori;
- Ogni partita ha un arbitro (software) indipendente e 2 o 3 assistenti (software) subordinati che controllano il gioco e garantiscono che le norme vengano rispettate; il comportamento e le prestazioni dell'arbitro e degli assistenti non necessitano di mostrare i limiti fisici tipiche degli esseri umani;

Il sistema software deve includere almeno:

- Un software di base, centralizzato o distribuito, che implementi tutta la logica della simulazione;
- Un pannello grafico di sola lettura per la visualizzazione del campo di calcio, i giocatori, la palla, l'arbitro e gli assistenti; per le figure umane in campo è sufficiente a rappresentarle come puntini numerati che si muovono sul display senza ricorrere a sofisticati rendering grafici, come in una visione di un tavolo Subbuteo visto dall'alto;
- Due distinti pannelli grafici di lettura-scrittura per permettere all'utente di influenzare l'azione altrimenti indipendente del team manager; il pannello deve visualizzare i parametri attuali per ogni giocatore; la frequenza di aggiornamento dei display saranno configurabili dall'utente;
- Un pannello grafico di sola lettura per la visualizzazione di statistiche configurabili dall'utente; la frequenza di aggiornamento di questo display sarà configurabile dall'utente;
- Il nucleo del software deve essere sviluppato in Ada. La progettazione del software deve permettere la sostituzione a piacimento degli algoritmi principali: in altre parole, il sistema software deve essere il più modulare, configurabile e scalabile possibile. Queste qualità contribuiranno alla valutazione;
- I pannelli grafici possono essere programmati in qualsiasi linguaggio che i partecipanti considereranno idonei allo scopo. La bellezza grafica dei pannelli sarà tuttavia solo un fattore secondario nella valutazione. Ciò che importa invece è che l'interazione e il flusso di dati e di controllo tra il nucleo e il software

1.2 Requisiti

Possiamo riassumere brevemente i requisiti principali del sistema, ricapitolandoli in una tabella riassuntiva esplicativa. Non vengono proposti, nel presente elaborato, nessun tipo di diagramma Use-Case UML in quanto lo scopo del documento non è quello di effettuare un'analisi approfondita dei requisiti, ma di individuare le necessità principali che il software deve soddisfare. Si tratta principalmente di riepilogare quanto proposto nel capitolo precedente al fine di schematizzare gli obiettivi principali.

Requisito	**Descrizione**	**Obbligatorio**
R01	Possibilità da parte dell'utente di giocare un singolo match	Sì
R02	Configurabilità dei parametri principali	Sì
R03	Configurabilità dei parametri fisici dei giocatori	Sì
R04	Dinamicità delle caratteristiche ed evoluzione durante il match (Es: stanchezza)	Sì
R05	Presenza per ogni squadra di un manager	Sì
R06	Il manager gestisce le configurazioni della squadra in campo	Sì
R07	I giocatori giocano sulla base delle decisioni del manager	Sì
R08	Presenza di un arbitro che controlla il rispetto delle regole	Sì
R09	Utilizzo di un processo per ogni singolo giocatore in campo	Sì
R10	Distribuzione del sistema su diversi calcolatori	Sì

Tabella 1: La lista dei requisiti obbligatori del sistema.

La colonna obbligatorio indica la necessità o meno del requisito al fine del funzionamento del sistema. Tutti i requisiti opzionali devono essere intesi come caratteristiche non necessarie all'implementazione di base del sistema, ma forniscono migliorie non trascurabili al sistema.

2 Analisi

2.1 La Struttura di Gioco

Il progetto prevede una serie di classi che compongono la struttura del gioco. Il riepilogo presenta gli elementi con una breve descrizione. Ogni singolo elemento verrà esaminato nelle sezioni successive entrando nel dettaglio delle varie componenti e dei vari metodi necessari per il corretto funzionamento del sistema.

Elemento	**Descrizione**
Timer	Gestione del tempo di gioco.
Campo	Dimensioni della superficie di gioco e gestione dei posizionamenti.
Arbitro	Controlla la regolarità del gioco.
Manager	Effettua le scelte per la squadra in base agli eventi.
Giocatore	Gioca la partita nella squadra.
Pallone	Oggetto condiviso dai giocatori

Tabella 2: La lista degli elementi che compongono il sistema.

2.1.1 Timer

Rappresenta la classe "Tempo" del nostro gioco. Si occupa di ricevere l'impostazione iniziale relativa alla durata della partita e di riproporzionare i 90 minuti regolari di un match rispetto ai minuti reali di gioco. Questo permette la gestione univoca del tempo di gioco, dall'inizio alla fine della partita attraverso il risveglio dei processi attivi e lo stop degli stessi al termine della partita.

2.1.2 Campo

Le dimesioni del campo dipendono dal numero dei giocatori. Indichiamo le dimensioni ufficiali LNP[1] sulla base del numero di componenti per ogni singola squadra:

Num. Giocatori	**Lunghezza**	**Larghezza**
5	42	22
7	55	32
8	65	38
11	105	68

Tabella 3: Le misure standard dei campi da calcio e calcetto.

Tutte le misure sopra indicate sono da considerarsi espresse in metri[2].

Le dimensioni del campo da gioco sono fondamentali rispetto al numero dei giocatori in quanto determinano la dimensione del movimento ovvero quanto il giocatore possa procedere in una direzione prima di finire fuori dal bordo.

[1] Lega Nazionale Professionisti

[2] Un metro è definito come la distanza percorsa dalla luce nel vuoto in un intervallo di tempo pari a 1/299.792.458 di secondo.

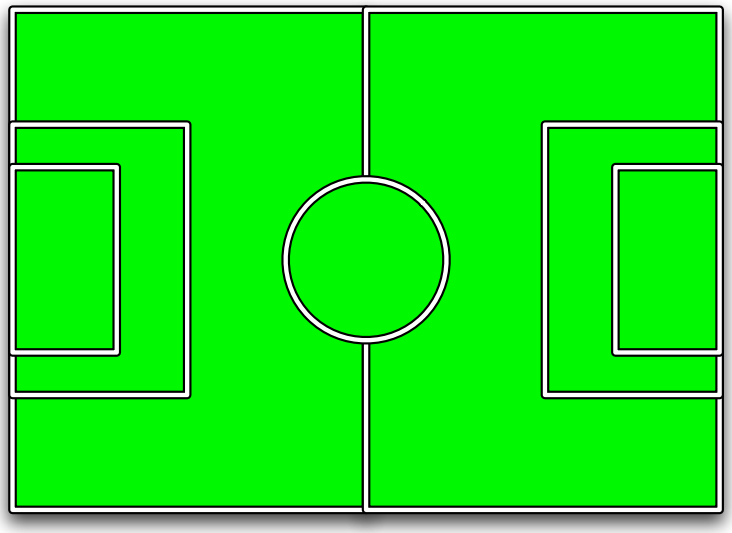

Figura 1: Il campo da gioco

Il campo è rappresentato attraverso un rettangolo in cui vengono definite le dimensioni dell'area complessiva di gioco e le aree in cui, esclusivamente per i portieri, è consentita presa con le mani. La rappresentazione del campo per l'elaboratore è una griglia composta da $2M \times 2N$ righe di separazione, dove M è la lunghezza del campo misurata in metri, mentre N è la larghezza. Potenzialmente, suddividendo il reticolato in maglie più strette, è possibile nascondere all'occhio umano il reticolato simulando un movimento fluido piuttosto che scattoso.

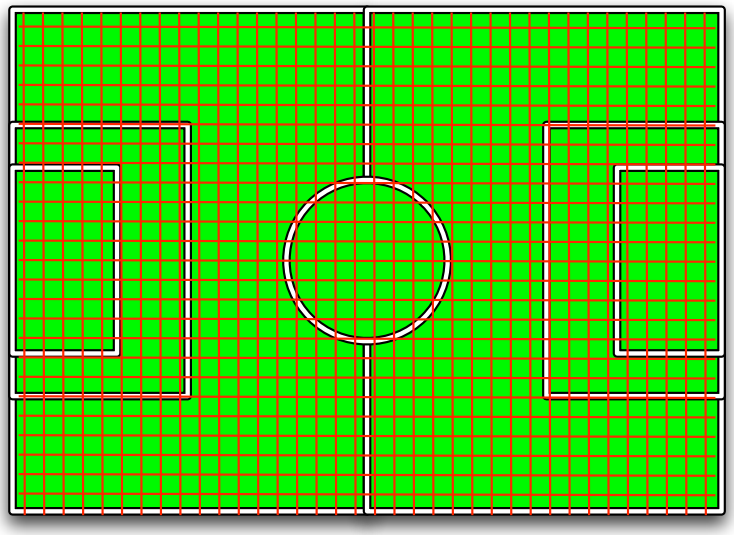

Figura 2: Il campo da gioco con il reticolato su cui avvengono i movimenti

2.1.3 Arbitro

Definiamo l'arbitro come un'entità che assolve il compito di giudicare alcuni eventi che sono segnalati come critici. Variabili aleatorie permettono di simulare il margine di errore decisionale rispetto a tali eventi.

L'arbitro viene assistito da due collaboratori, i guardalinee, che influenzano la decisione attraverso altre 2 variabili aleatorie, spostando il peso della decisione dell'arbitro a favore o contro il fischio.

Non vi sono elementi nel campo da gioco che rappresentano l'arbitro e i suoi collaboratori. Potenzialmente è possibile implementare solamente il fischio nel momento in cui viene segnalata una irregolarità di gioco.

2.1.4 Manager

Anche il manager della squadra, come per gli arbitri, non è rappresentato nel campo da gioco. Non interviene attivamente al gioco sul campo, ma si occupa di seguire la logica con la quale la squadra gioca la partita. Interviene nella scelta della strategia di gioco, nella disposizione iniziale della formazione, nelle eventuali variazioni necessarie a seguito dei risultati in campo e nelle sostituzioni dei vari giocatori all'interno del campo di gioco o tra giocatori in panchina.

Possiamo immaginare facilmente il manager come la mente della squadra che si occupa di interpretare la partita e di mettere la squadra in condizione di vincere.

2.1.5 Giocatore

Il giocatore rappresenta la forza in campo che gioca attivamente la partita, cerca di segnare il maggior numero di reti e di subirne il minimo al fine di vincere il match.

Ogni squadra dispone di n giocatori in campo di cui necessariamente:

- 1 portiere;
- 2 difensori;
- 2 centrocampisti;
- 1 attaccante;

Questa disposizione minima obbligatoria evita degli sbilanci eccessivi in favore di un reparto piuttosto che un altro, rendendo i match relativamente equilibrati e giocabili. In caso di calcetto a 5, la disposizione minima è leggermente differente per effetto del minor numero di giocatori in campo.

Ogni giocatore è contraddistinto da un pallino colorato. La squadra A è contraddistinta dal colore blu, mentre la squadra B dal colore rosso.

(a) Giocatore A (b) Giocatore B

Figura 3: I giocatori delle 2 squadre.

Il colore della squadra viene assegnato automaticamente in modo casuale durante il processo di inizializzazione della partita.

2.1.6 Palla

La palla è uno degli elemeni fondamentali del gioco. È l'oggetto condiviso dai giocatori e può essere controllato in ogni istante di tempo da al massimo un giocatore.

Viene mostrato nel campo da gioco come un pallino di colore nero, per contraddistinguerlo dai giocatori.

Figura 4: Il pallone da gioco

Nel momento in cui un giocatore entra in possesso del pallone, per distinguerlo dai compagni, il margine esterno viene rappresentato in grassetto. Questo permette di comprendere a colpo d'occhio quale giocatore detenga il pallone e come si muova con esso.

(a) Giocatore A con palla

(b) Giocatore B con palla

Figura 5: I giocatori delle 2 squadre in possesso del pallone.

2.2 I Dettagli di Gioco

2.2.1 Movimento

Ogni giocatore è in grado di muoversi all'interno del campo da gioco per interagire con gli altri giocatori della squadra, per contrastare gli avversari ed impossessarsi del pallone, per avanzare verso la porta con i compagni o per tirare il pallone nel momento in cui la distanza dalla rete è sufficiente per poter compiere un buon tiro e segnare un gol.

È stato deciso di semplificare il movimento nei 4 assi fondamentali:

- su;
- giù;
- destra;
- sinistra;

In questa maniera il giocatore è in grado di muoversi negli incroci del reticolato e dichiarare a compagni ed avversari la propria posizione al fine del corretto svolgimento del match. Inoltre, attraverso una serie di piccoli movimenti di questo tipo ogni giocatore è in grado di percorrere potenzialmente tutto il campo di gioco.

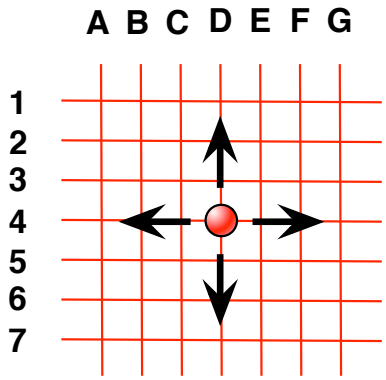

Figura 6: Gli assi del movimento dei giocatori.

2.2.2 Zone di Movimento

Per evitare che i giocatori sbilancino la squadra in modo esagerato è stato deciso di creare, per ogni area di gioco, delle "zone di movimento" che permettono di mantenere i difensori all'interno della zona di difesa, i centrocampisti entro dei margini di gioco centrali e gli attaccanti al limite del centrocampo.

In questo modo, in caso di contropiede avversario, non rimangono zone troppo scoperte e che gli avversari possono sfruttare a loro favore per effettuare azioni indesiderate.
La difesa muove sempre dalla porta fino al limite del centrocampo, evitando di entrare nella zona di campo avversaria.

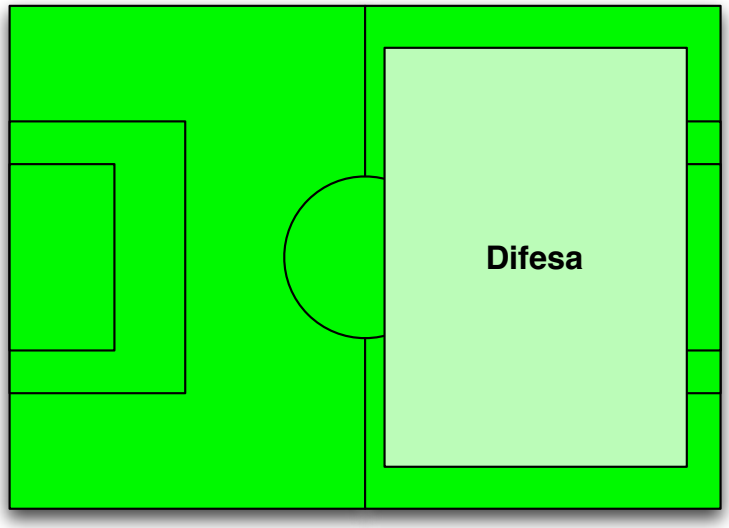

Figura 7: La zona di movimento dei difensori.

Il centrocampo può muoversi in tutta la parte centrale del campo e nelle fascie laterali per assistere gli attaccanti e i difensori.

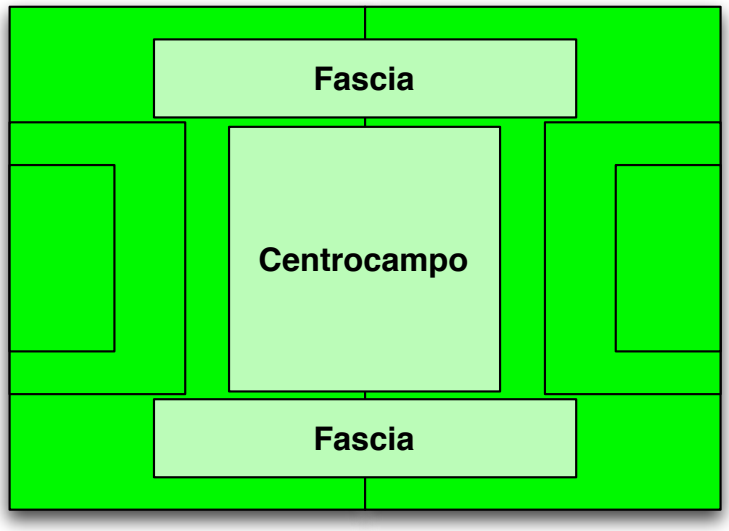

Figura 8: La zona di movimento dei centrocampisti.

Gli attaccanti invece muovono in tutta la zona del campo avversario senza mai retrocedere oltre il limite del centrocampo, per poter intervenire in caso di passaggi da parte dei compagni in caso di contropiede.

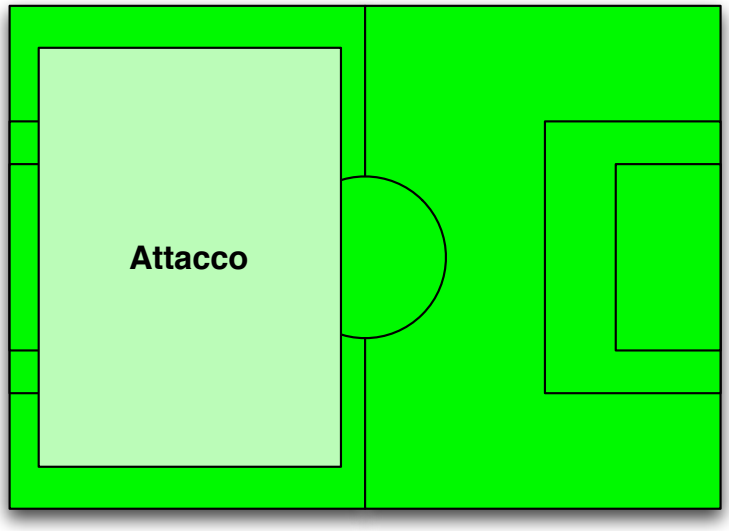

Figura 9: La zona di movimento degli attaccanti.

2.2.3 I Giocatori

Ogni giocatore, oltre al movimento all'interno del campo è in grado di effettuare alcune azioni di base ed in particolar modo legate ad alcune condizioni fondamentali:

- possesso del pallone da parte della propria squadra:
  - del giocatore stesso
  - di un compagno
- possesso del pallone da parte degli avversari

Definiamo con il termine *logica* una modalità di gioco relativa ad ogni singolo giocatore e legata alle condizioni di cui sopra.

Elenchiamo brevemente le possibilità che un giocatore può trovarsi a dover affrontare, analizzandole poi dettagliatamente nei paragrafi successivi:

1. il giocatore ha la palla:
   - logica di passaggio
   - logica di tiro
   - logica di avanzamento
   - logica di posizione
2. la squadra ha la palla:
   - logica di smarcatura
   - logica di avanzamento
   - logica di posizione
3. gli avversari hanno la palla:
   - logica di indietreggiamento
   - logica di contrasto
   - logica di posizione
4. il giocatore in porta:
   - logica di contrasto (sull'attaccante)
   - logica di parata (solo su tiro avversario)
   - logica di posizione

3 Applicazioni delle Logiche

Nella presente sezione analiziamo le formule ed i coefficenti che permettono di determinare, rispetto all'elenco visto precedentemente, le varie azioni che i giocatori possono compiere ed i risultati a cui possono portare. Come abbiamo detto, utilizzeremo il termine "logica" per definire una modalità con cui il giocatore determina le possibilità di gioco, sceglie l'azione migliore e la effettua.

Cominciamo definendo un particolare fondamentale: la "visibilità" del giocatore.

Con visibilità intendiamo la cognizione dello stato di gioco nell'istante t da parte del giocatore in questione. Per semplicità è stato deciso di definire con "visibilità" un quadrato di lato x (con x dispari) e il giocatore oggetto della funzione risiede nell'intersezione centrale. Questo permette al giocatore di conoscere la situazione di compagni ed avversari per tutte le celle contenute nel quadrato.

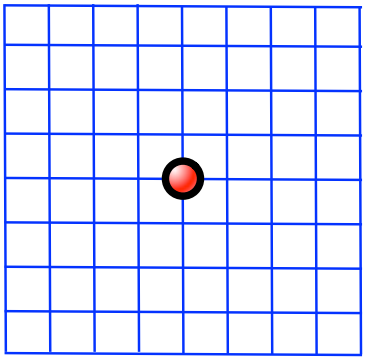

Figura 10: La "visibilità" dei giocatori in gioco.

Inoltre è possibile rendere il parametro x soggettivo rispetto ad ogni giocatore, il che permette a giocatori come il regista di avere una grande visione di gioco rispetto a difensori ed attaccanti.

Tutti i movimenti e i posizionamenti dei giocatori che verranno affrontati nei prossimi capitoli sono sempre da intendersi legati ai ruoli che i giocatori assumono nel campo e alle aree di movimento a cui sono vincolati. Si faccia riferimento alla sezione 2.2.2 per i dettagli delle zone in cui i giocatori sono liberi di muovere.

In tutto il capitolo trascureremo ogni aspetto legato alla concorrenza, tematica che verrà affrontata nel dettaglio al capitolo 5.

3.1 Giocatore con Palla

Il caso più complesso che possiamo affrontare è quello legato al possesso di palla da parte del giocatore. In questa casistica dobbiamo risolvere la maggior parte dei problemi legati alle scelte dei giocatori e determinare le regole con cui avanzano, tirano o passano il pallone ai compagni di squadra.

La priorità con cui un giocatore effettua un'azione è:

1. tiro in porta
2. passaggio ad un compagno
3. avanzamento con palla

4. mantenimento della posizione

Questo perchè se il giocatore vede la porta, effettua un tiro per tentare di segnare una rete. Altrimenti, il passaggio è il metodo più rapido per muovere velocemente la palla ed avanzare verso la porta. In caso non ci sia visione della porte e non ci siano compagni liberi a portata, si processa un avanzamento di posizione. Il mantenimento della posizione è una casistica limite che cercheremo sempre di evitare, ma che ci permetterà di risolvere situazioni complicate.

3.1.1 Logica di Tiro

La prima logiche che andiamo ad analizzare è quella legata al tiro in porta.

Un giocatore che inizia la logica di tiro ha sicuramente visibilità sulla porta e pertanto non ne è relativamente distante.

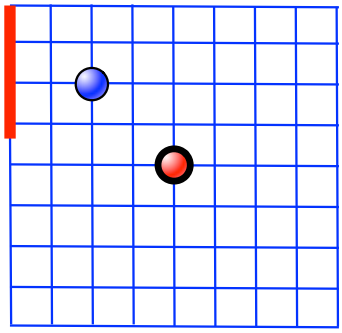

Figura 11: Giocatore con palla e visione della porta (linea rossa).

Il tiro può essere un buon tiro o un tiro mediocre. Inoltre può essere troppo potente o troppo debole e può venir parato dal portiere avversario. Per simulare un comportamento realistico del tiro, andiamo a considerare alcuni fattori fondamentali:

- potenza
- precisione
- distanza

Questi 3 coefficienti fortemente soggettivi rispetto al giocatore e alla posizione, vengono combinati assieme in un'equazione per produrre un coefficiente che verrà valutato assieme al coefficiente di parata del portiere (che vedremo in seguito nella sezione 3.4.2) per determinare il successo o il fallimento del tiro e la conseguente rete della squadra.

L'equazione è composta da

$$T_1 = \frac{potenza}{100} \cdot \frac{precisione}{100} \cdot \left(1 - \frac{distanza}{diagonale}\right)$$

dove potenza è il valore di potenza del giocatore espresso con un valore da 1 a 100, precisione è la precisione di tiro, anch'essa espressa con un valore da 1 a 100, mentre l'ultimo coeffienciente è il risultato del rapporto inverso tra la distanza del giocatore dalla porta e la diagonale del campo.

Per dare un'idea di come varino i vari coefficienti ne esprimeremo alcuni campioni nella tabella sottostante ed andremo a definire che cosa rappresentano in termini di gioco.

Potenza	**Precisione**	**Distanza**	**Coefficiente**
0.3	0.3	0.3	0.027
0.3	0.3	0.6	0.054
0.3	0.3	0.9	0.081
0.3	0.6	0.3	0.054
0.3	0.6	0.6	0.108
0.3	0.6	0.9	0.162
0.3	0.9	0.3	0.081
0.3	0.9	0.6	0.162
0.3	0.9	0.9	0.243
0.6	0.3	0.3	0.054
0.6	0.3	0.6	0.108
0.6	0.3	0.9	0.162
0.6	0.6	0.3	0.108
0.6	0.6	0.6	0.216
0.6	0.6	0.9	0.324
0.6	0.9	0.3	0.162
0.6	0.9	0.6	0.324
0.6	0.9	0.9	0.486
0.9	0.3	0.3	0.081
0.9	0.3	0.6	0.162
0.9	0.3	0.9	0.243
0.9	0.6	0.3	0.162
0.9	0.6	0.6	0.324
0.9	0.6	0.9	0.486
0.9	0.9	0.3	0.243
0.9	0.9	0.6	0.486
0.9	0.9	0.9	0.729

Tabella 4: Alcuni valori campione per il coefficiente di tiro (media 0.216).

E' facile comprendere osservando la tabella che al crescere delle carratteristiche ci siano differenti effetti sul tiro: un tiro può essere molto potente (ultimi risultati in basso) ma non efficace perchè poco preciso ed effettuato da distanze troppo elevate. Inoltre può essere molto preciso e puntare all'incrocio dei pali, ma non sufficientemente potente o magari effettuato da troppo distante e facilmente parabile dal portiere.

Chiameremo il valore ottenuto come T_1 e dovrà essere confrontato con il risultato del calcolo relativo alla parata per determinare il successo del tiro.

Il fattore aleatorio è stato escluso dal calcolo in quanto potrebbe complicare ulteriormente i conteggi e rendere difficile segnare una rete. Non sarebbe però complesso introdurre il classico fattore c generando un numero casuale e incrementando o moltiplicando il coefficiente ottenuto precendentemente per c ottenendo un coefficiente T_2 maggiormente influenzato da leggi pseudo-casuali.

3.1.2 Logica di Passaggio

Un giocatore che ha la palla può decidere di passare il pallone ad un compagno libero. La logica con cui avviene il passaggio è legato all'avanzamento verso la porta avversaria. Non è però sempre possibile avanzare

rispetto alla porta per effetto delle distanze dei giocatori avversari all'uomo a cui si vorrebbe passare il pallone.

Anche in questo caso entra in gioco la "visibilità" del giocatore e la sua capacità di vedere compagni liberi nelle vicinanze.

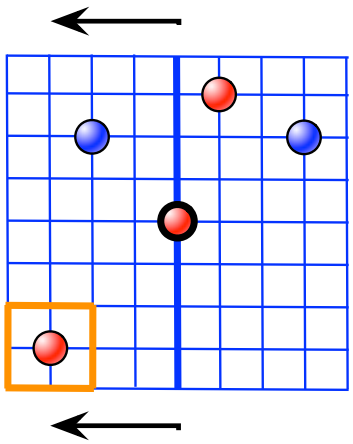

Figura 12: Giocatore con palla e visione di compagni ed avversari.

Il giocatore in possesso di palla, valuta la posizione del giocatore più vicino e più avanzato rispetto a se, controlla che sia libero entro un certo raggio ed eventualmente passa il pallone. L'equazione che regola il passaggio, in questo caso è più semplice della precedente e garantisce quasi sempre il buon fine del passaggio al giocatore destiantario entro un intervallo di tempo t. In particolare l'equazione è influenzata dalla distanza tra chi passa la palla e chi la deve ricevere e dalla precisione del passaggio:

$$P_1 = \left(1 - \frac{distanza}{diagonale}\right) \cdot \frac{precisione}{100}$$

Nel caso non si raggiunga la soglia, il pallone viene lanciato casualmente in una delle 4 direzioni per un numero variabile di intersezioni. Questo permette potenzialmente alla squadra avversaria di impossessarsi della palla in quanto ne prenderà il possesso il primo giocatore a venirne in contatto indipendentemente dallo schieramento.

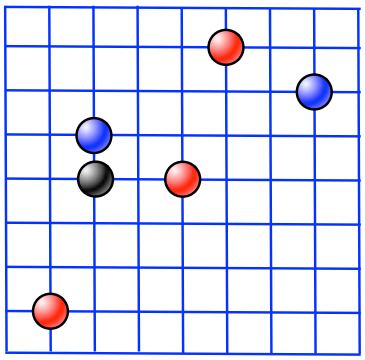

Figura 13: Giocatore avversario si avvicina al pallone prima degli altri.

Il buon fine del passaggio è soggetto al superamento della soglia minima per il passaggio pari a 0.3. Il valore di soglia è stato approssimato secondo il criterio per cui un passaggio non avviene mai a distanze elevate e la precisione dei giocatori non dovrebbe essere mai troppo bassa.

Precisione	**Distanza**	**Coefficiente**
0.3	0.6	0.18
0.3	0.8	0.24
0.6	0.6	0.36
0.6	0.8	0.48
0.9	0.6	0.54
0.9	0.8	0.72

Tabella 5: Alcuni valori campione per il coefficiente di passaggio (soglia 0.3).

Come si può facilmente vedere dalla tabella, per passaggi molto ravvicinati ($\frac{1}{4}$ della diagonale del campo), anche con una precisione molto bassa (intorno a 40 punti) il passaggio avviene con successo. Chiaramente, giocatori più precisi non sbaglieranno passaggi facili, ma potrebbero comunque avere qualche difficoltà in passaggi lunghi a giocatori distanti.

Anche qui, come nel tiro in porta, può essere introdotto un fattore aleatorio che può scombinare gli eventi: il fattore c può entrare in calcoleo nell'equazione che assumerebbe la forma:

$$P_1 = \left(1 - \frac{distanza}{diagonale}\right) \cdot \frac{precisione}{100} \cdot c$$

dove c è un numero casuale tra 0,5 e 1,5. Questo potrebbe portare al fallimento di ottimi passaggi e al successo di passaggi inizialmente fallimentari.

Nella logica di passaggio non è stato affrontato il passaggio all'indietro ad un compagno di squadra. Questo per un semplice motivo. Se avessimo considerato anche i passaggi all'indietro, avremmo scatenato un meccanismo infinito di passaggi tra giocatori con una piccolissima percentuale d'errore. Potenzialmente gli interi 90 minuti di gioco potrebbero essere spesi in passaggi tra giocatori senza nemmeno un tiro in porta. Forzando il passaggio in avanti costringiamo i giocatori ad avanzare, passare o tirare in porta. In questo modo evitiamo diverse situazioni che possono causare un stallo del gioco inteso come assenza di azioni a causa di una politica non consistente.

3.1.3 Logica di Avanzamento

Un'altra azione a disposizione di un giocatore con la palla che non può effettuare un tiro e non può passare la palla è quella di avanzare con il pallone verso la porta. Un giocatore può anche muovere sulla linea verticale verso l'alto o verso il basso, specialmente verso un compagno di squadra che si sta smarcando per ricevere il passaggio del compagno.

Un giocatore potrebbe però continuare con un movimento ripetitivo dall'altro verso il basso e viceversa per tutta la durata della partita. Per evitare una situazione di questo genere, è stato scelto di limitare a y il numero di movimenti in orizzontale fino al successivo avanzamento verso la porta avversaria.

Per risolvere il problema è sufficiente inserire nella classe *Giocatore* una variabile y inizializzata al numero di movimenti verticali massimi. Un contatore decrementa il valore di y per ogni movimento consecutivo sull'asse verticale. Quando y raggiunge il valore 0, l'avanzamento è obbligatorio e y viene reimpostata al valore iniziale.

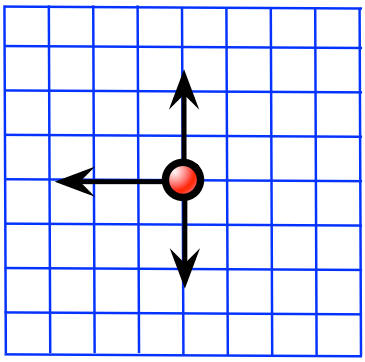

Figura 14: Giocatore in fase di movimento con palla.

3.1.4 Logica di Posizione

Il giocatore ha come azione di default in caso di emergenza un'ultima logica che viene utilizzata nel momento in cui nessuna delle logiche precedenti porti ad un risultato: il mantenimento della propria posizione in campo senza alcun tipo di movimento.

È una logica che non determina alcuna variazione del gioco, ma che permette di compiere in ogni caso un'azione e di restituire un parametro di "azione compiuta" alla classe "Timer" per poter procedere nell'invio del successivo istante temporale *t+1*.

3.2 Squadra con Palla

Un tipo di logica differente si applica quando il giocatore non è direttamente in possesso della palla, ma lo è un compagno di squadra. La strategia di movimento del giocatore deve essere finalizzata ad assistere il giocatore con la palla al fine di poter ricevere un passaggio e procedere verso la porta.

3.2.1 Logica di Smarcatura

Un giocatore senza palla deve garantire ai compagni di essere libero nel momento in cui sia necessario passare il pallone. È fondamentale infatti che assista la squadra nell'avanzamento verso la porta e si proponga per ricevere il pallone ed effettuare a sua volta un'azione.

In questo modo l'azione prioritaria è il movimento verso una intersezione distante almeno n intersezioni da uno o più avversari, dando prevalenza ad azioni di avanzamento rispetto a quelle di arretramento in modo da favorire l'attacco verso la porta avversaria.
L'azione di spostarsi verso un punto distante almeno n intersezioni consecutive da tutti gli avversari è detta "smarcatura".

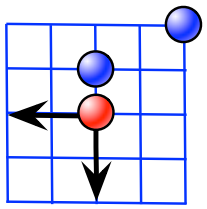

Figura 15: Giocatore in fase di smarcatura senza palla.

3.2.2 Logica di Avanzamento

Nel momento in cui il giocatore risulti già smarcato rispetto agli avversari si applica la stessa logica di avanzamento che si applica ad un giocatore in possesso di palla: il movimento deve essere direzionato in avanti verso la porta avversaria. Valgono le stesse regole viste nella sezione 3.1.3.

3.2.3 Logica di Posizione

Come gia visto nella sezione 3.1.4, anche per il giocatore senza palla esiste un'azione di "emergenza" che permette di evitare situazioni di stallo determinate da impossibilità di compiere un movimento nel campo.

3.3 Avversari con Palla

Finora abbiamo visto tutte le logiche al possesso del pallone da parte della squadra o del giocatore, ma grossomodo per il 50% della durata della partita la palla sarà in mano ai giocatori avversari e la squadra dovrà assumere un atteggiamento difensivo al fine di evitare che gli avversari possano segnare una rete. Analiziamo le logiche di gioco che permettono di difendere la porta.

3.3.1 Logica di Contrasto

Nel momento in cui gli avversari prendono possesso del pallone, è necessario un meccanismo che permetta al giocatore di recuperare il pallone al fine di invertire le parti e portare la propria squadra in fase di attacco. Questa regola è definita come logica di contrasto ed è rappresentata nel campo di gioco da un contatto fisico tra giocatori. Nel contrasto, i due giocatori si scontrano per potenza e il più forte prende il controllo del pallone.

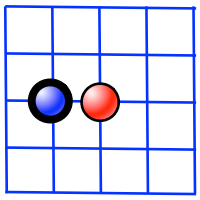

Figura 16: Giocatore in fase di contrasto contro un avversario con palla.

Il contatto fisico avviene tra due giocatori in posizione ravvicinata, a distanza di un'intersezione l'uno dall'altro. La logica con cui avviene il contrasto viene applicata ad entrambi i giocatori e rispetta la formula:

$$C_1 = \frac{potenza}{100} + c$$

dove potenza è la potenza del giocatore e c un fattore casuale con valore compreso tra -0.2 e 0.3 che rappresenta un rimpallo a favore o contro il giocatore.

Nel momento in cui avviene il contrasto l'oggetto condiviso viene liberato per l'istante t in cui viene valutato il contrasto e poi preso dal giocatore che ottene il valore superiore rispetto al valore di contrasto.

In caso la formula produca due valori uguali, il giocatore in difesa vince il contrasto e prende il controllo della risorsa e il gioco ricomincia.

I dettagli legati alla liberazione della risorsa e alla ripresa della stessa a seguito del contrasto vengono discussi nel capitolo 3.4.1.

3.3.2 Logica di Indietreggiamento

Nel momento in cui un giocatore non ha visibilità sul giocatore con la palla può effettuare una sola azione: indietreggiare a supporto della fase difensiva.

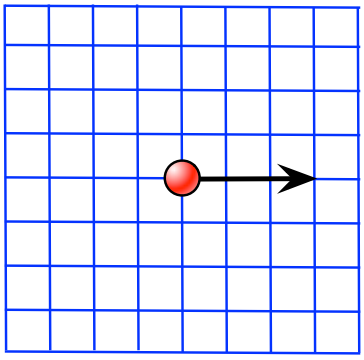

Figura 17: Giocatore in fase di indietreggiamento.

In questa maniera, il giocatore si avvicina ai giocatori in difesa, limita le aree di movimento degli avversari in fase di attacco e, in caso di recupero del pallone, si propone per il passaggio da parte dei compagni.

3.3.3 Logica di Posizione

Anche nel caso in cui la propria squadra non sia in possesso del pallone, come gia visto nella sezione 3.1.4, esiste la solita azione di "emergenza" ovvero il mantenimento della propria posizione in campo.

3.4 Parata del Portiere

L'unico giocatore eccezionale nel campo di gioco è il portiere. Questo particolare giocatore non segue le regole valide per tutti gli altri giocatori nel campo di gioco ma è soggetto a delle proprie logiche che lo differenziano nel modo di giocare, di muoversi e di reagire agli eventi come avanzamenti dell'attaccante avversario e al tiro in porta.

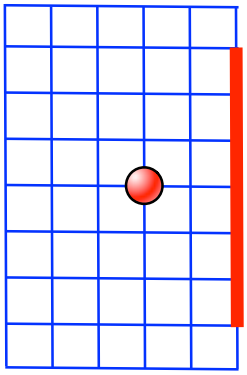

Figura 18: Portiere in posizione difensiva.

Vediamo nel dettaglio le logiche che ne regolano i movimenti e la posizione.

3.4.1 Logica di Contrasto

La logica primaria che permette il movimento del portiere è il contrasto con il giocatore avversario. Limitatamente all'area di gioco entro cui il portiere ha la possibilità di muoversi, il portiere cerca il contatto fisico con l'avversario che controlla il pallone. In questo modo può tentare uno scontro fisico e prendere il controllo del pallone evitando che l'attaccante possa segnare una rete.

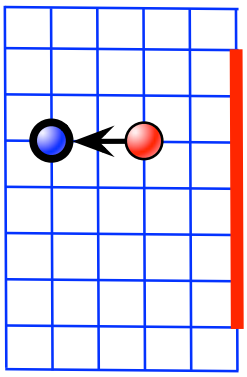

Figura 19: Portiere in fase di contrasto.

Le regole per la determinazione del contrasto sono uguali a quelle viste nella sezione 3.3.1. L'unica differenza è relativa al peso del fattore c. In questo caso, poichè il portiere è l'unico giocatore a cui è permesso l'intervento con le mani, il valore di c nella formula:

$$C_2 = \frac{potenza}{100} + c$$

varia tra 0 e 0.2, rendendo la potenza minima del portiere pari alla propria e permettendo solo incrementi su di essa.

3.4.2 Logica di Parata

Nel momento in cui un giocatore effettua un tiro in porta, il portiere effettua l'unica azione disponibile: tenta una parata del tiro.

L'azione di parare un tiro è indipendente dalla posizione attuale del portiere in quanto un tuffo gli permette di muoversi velocemente da una parte all'altra della porta.

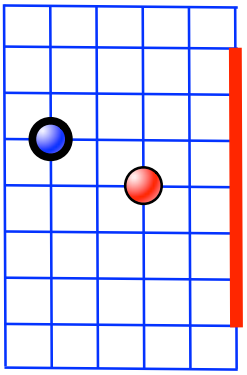

Figura 20: Portiere in fase di parata del tiro avversario.

L'equazione per determinare il valore della parata è uguale a:

$$P_1 = \frac{agilità}{100} \cdot \frac{reattività}{100} \cdot \left(1 - \frac{distanza}{diagonale}\right)$$

dove agilità è il valore di agilità del portiere espresso con un valore da 1 a 100, reattività è la velocità di reazione, anch'essa espressa con un valore da 1 a 100, mentre l'ultimo coeffienciente è il risultato del rapporto inverso tra la distanza del portiere dal giocatore che sta effettuando il tiro e la diagonale del campo.

La parata è in contrasto con il tiro visto nella sezione 3.1.1. Il tiro genera un valore che viene confrontato con quello generato dalla parata del portiere. In caso il valore P_1 sia inferiore a T_1, il portiere non è riuscito a parare il tiro (per merito dell'attaccante o demerito del portiere) e la palla è finita in rete. Nel caso in cui il valore P_1 sia uguale o maggiore di T_1 il portiere ha compiuto una parata e ha salvato la porta.

L'azione ricomincia dal portiere con un passaggio verso un compagno per tentare un'azione d'attacco.

3.4.3 Logica di Posizione

Anche il portiere, in qualità di giocatore come tutti gli altri, ha la possibilità di gestire una situazione di "emergenza" rimanendo in posizione e non generando movimenti. Si faccia riferimento alla sezione 3.1.4 per i dettagli della logica.

4 Modello

Analizziamo ora il modello del sistema e la composizione delle classi e dei metodi principali necessari al corretto funzionamento del software come descritto nella tabella sottostante.

Classe	**Identifica**	**Descrizione**
Timer	Tempo	Regola la durata del match e ne determina la fine.
Field	Campo	Gestisce le dimensioni della superficie di gioco e il posizionamento di giocatori e palla su di esso.
Referee	Arbitro	Controlla la regolarità del gioco e fischia eventuali scorrettezze.
Manager	Manager	Gestisce il modulo di gioco, effettua i cambi tra giocatori in campo ed in panchina e schiera la formazione d'attacco o di difesa.
Team	Squadra	Associa i vari giocatori alla squadra di cui fanno parte.
Player	Giocatore	Gioca la partita assieme ai compagni di squadra.
Ball	Pallone	Oggetto condiviso dai giocatori

Tabella 6: La lista degli elementi principali che compongono il sistema.

Le classi principali sono le 6 classi fondamentali appena presentate. Se mancasse anche solo una di esse, non si potrebbe realizzare il gioco nella nostra realtà virtuale. Ognuna di esse rappresenta un elemento presente e visibile nel campo di gioco ed attivamente partecipe allo svolgimento della partita. La sola eccezione vale per la classe *Team* che si occupa di raggruppare i giocatori in un'unica squadra.

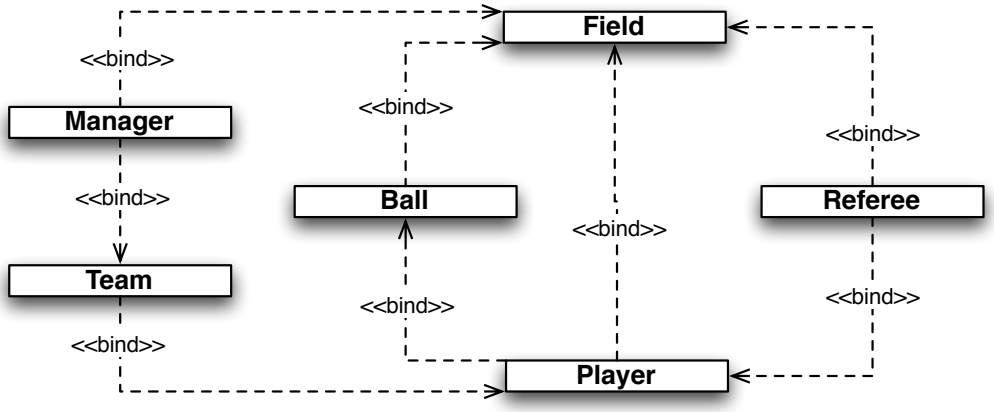

Figura 21: Diagramma generale delle classi del sistema.

Lo schema esposto presenta le relazioni che intercorrono tra le classi principali del nostro gioco oltre all'indicazione del verso in cui la comunicazione è direzionata.

Approfondiamo nelle prossime sezioni le singole classi entrando nel dettaglio per comprendere la loro tipologia e le funzioni principali che permettono il corretto svolgimento del gioco.

4.1 Timer

Il timer è la classe che si occupa della gestione del tempo. Ad intervalli regolari variabili incrementa un contatore che scandisce il trascorrere del tempo durante la partita.

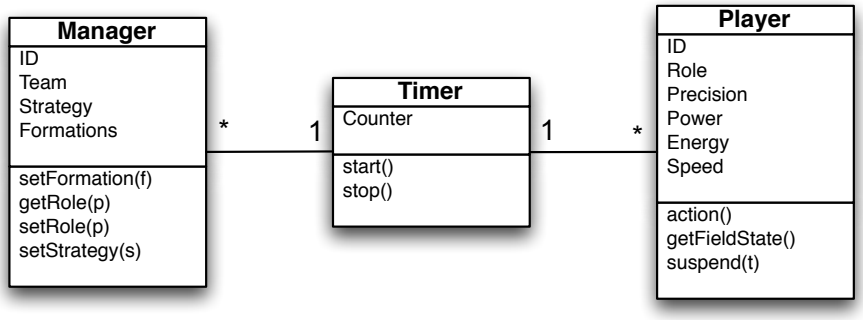

Figura 22: Diagramma delle classi per il timer di gioco.

La classe tempo ha metodi di accesso diretto a tutti i processi attivi in campo che necessitanto una inizializzazione simultanea ovvero:

- giocatori
- manager
- arbitro

In questa maniera il gioco inizia simultaneamente per tutte le parti coinvolte che cominciano ad interagire tra di loro e con gli oggetti passivi. Al termine della partita, per lo stesso principio, un metodo della classe si occupa di mettere in stop i thread decretando il termine della partita e l'eventuale vincitore se non si verifica una situazione di parità.

4.2 Campo

Il campo deve essere la classe di riferimento necessariamente accessibile a tutti i giocatori in campo. Si occupa di salvare i posizionamenti dei giocatori nell'area di gioco a seguito dei movimenti effettuati e di fornire alcune informazioni legate al campo di gioco come ad esempio la posizione del pallone e la dislocazione delle porte.

Ad ogni giocatore che effettua l'interrogazione di stato restituisce:

- la posizione nel campo di gioco;
- le direzioni percorribili;
- la posizione dei compagni;
- la posizione degli avversari;
- la posizione del pallone;

Ogni giocatore deve poter accedere alla classe *Field*, effettuare la richiesta del vettore delle posizioni ed attendere una risposta dalla classe. Essa restituirà una lista contenente le informazioni relative alla posizione dei compagni, degli avversari e del pallone necessarie al giocatore per poter compiere il proprio movimento in gioco coerentemente con gli spazi disponibili e con la logica applicata.

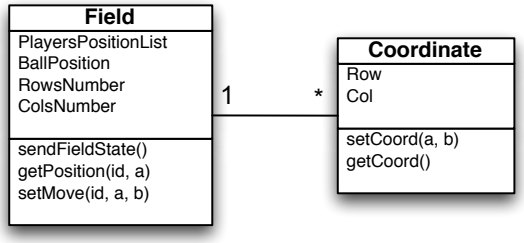

Figura 23: Diagramma delle classi per il campo di gioco.

I valori delle dimensioni di gioco e delle aree di movimento sono inizializzate non appena vengono definite la tipologia di partita (calcio/calcetto) e il numero di giocatori in campo.

Un vettore di posizioni contiene i vari posizionamenti dei giocatori e lo schieramento a cui appartengono. Una variabile ad-hoc contiene invece il posizionamento del pallone e viene trattato in modo similare al vettore precedente.

Le funzioni che permettono di accedere ai vettori di stato sono:

function GetPositions return PositionList che ritorna un listato di posizioni dei vari giocatori in campo;

function GetBallPosition return Position che restituisce la posizione della palla in campo;

Nel caso si trovi al limite della propria area di movimento, le uniche azioni possibili sono: rimanere nella propria posizione e non compiere alcun movmento o, in alternativa, muovere lateralmente per avvicinarsi al giocatore con la palla (se visibile).

Il campo gestisce anche alcuni eventi particolari legati all'uscita del pallone dall'area di gioco. Mentre possiamo ammettere che un giocatore rimanga tutta la partita in campo e quando raggiunga il bordo del campo non possa attraversarlo, è lecito pensare che il pallone fuoriesca dall'area di gioco. In questo caso viene lanciata una eccezione e la palla passa alla squadra avversaria. Si possono verificare 3 casi principali:

Fallo laterale avviene quando un giocatore lancia la palla oltre la linea laterale del campo. La palla passa ad un giocatore avversario e il gioco riprende dal bordo in cui è avvenuta l'uscita, punto in cui vengono riposizionati giocatore e pallone;

Rimessa dal fondo è causata da un giocatore avversario che scaglia la palla oltre la linea della porta aversaria. Il pallone passa al portiere e il gioco ricomincia dal fondo del campo;

Calcio d'angolo viene fischiato quando un giocatore lancia la palla oltre la linea della propria porta. Il gioco riprende dall'angolo con il riposizionamento di palla e giocatore;

La funzione che permette all'arbitro di resettare le posizioni in base all'evento è:

procedure ResetPosition(ID : in Integer; P : Position) nella quale viene passata l'ID del giocatore che andrà a battere la rimessa e la posizione in cui verranno posizionati giocatore e pallone;

4.3 Arbitro

L'arbitro è di per se un'entità molto semplice: non è presente direttamente nel campo di gioco e viene richiamato in casi chiaramente definiti per determinare il corretto svolgimento del gioco.

Le casistiche in cui l'arbitro viene chiamato in causa sono:

- contrasto tra due giocatori (possibile fallo);
- palla oltre il limite del campo:
  - fallo laterale;
  - rimessa dal fondo;
  - calcio d'angolo;
- fuorigioco (non trattato);

In tutti i casi, il fischio dell'arbitro comporta il fermo del gioco, la riassegnazione della palla e l'inizio di una nuova azione.

Definiamo cosa fa l'arbitro nei casi sopra esposti:

- fallo tra giocatori:
  1. mantiene la posizione della palla sul punto del fallo;
  2. mantiene la posizione del giocatore che ha subito il fallo vicino alla palla;
  3. riposiziona posiziona l'altro giocatore a distanza definita;
  4. fa riprendere il gioco;
- palla fuori dal campo:
  1. posiziona la palla sul bordo del campo nel punto di uscita;
  2. posiziona il giocatore sulla palla;
  3. ne assegna il possesso;
  4. fa riprendere il gioco;

Nello schema sottostante si comprende facilmente come l'arbitro intervenga nelle varie fasi di gioco ed interagisca con le altre classi.

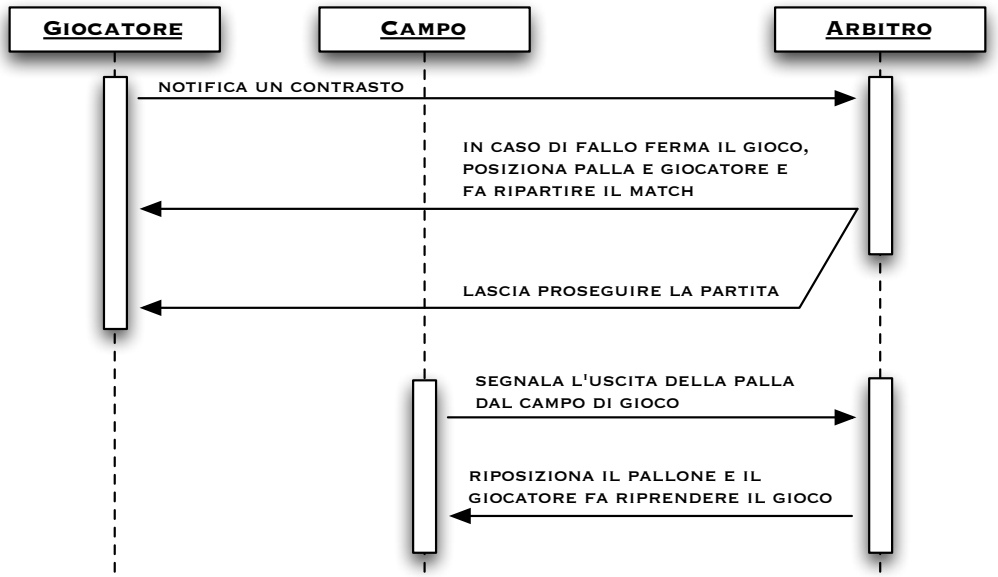

Figura 24: Diagramma di sequenza dell'arbitro di gioco.

L'unico aspetto determinante rispetto a quanto visto sotto è legato esclusivamente alla priorità con cui l'arbitro interviene nel gioco: per un'analisi dettagliata delle priorità in campo, rimandiamo alla sezione 5.3 in cui vengono trattate nello specifico.

4.4 Manager

Il Manager deve essere considerato la mente del gioco in quanto non partecipa attivamente alla partita ma si occupa di definire le strategie di gioco e i cambi tra giocatori in campo o in panchina. Le scelte legate alla posizione dei giocatori e alle strategie sono legate ad alcuni fattori oggettivi che permettono al manager di prendere le decisioni opportune per modificare la squadra in campo:

- risultato parziale della partita;
- numero di giocatori in campo (a seguito di potenziali espulsioni o infortuni);
- sbilanciamento attuale della formazione (attacco o difesa);
- formazione in campo;
- modulo di gioco;

Manager è un thread che viene risvegliato ad intervalli regolari per visionare e comprendere l'andamento del match ed apportare correzioni alla squadra oppure in casi particolari in cui viene risvegliato. Gli eventi che scatenano il Manager possono essere identificati da:

- gol della propria squadra;
- gol della squadra avversaria;
- energia di un giocatore che scende sotto una soglia predefinita (e necessita un cambio);

- giocatore che subisce un infortunio durante la partita;
- fischio dell'arbitro per contrasto tra giocatori;
- espulsione di un giocatore in campo;

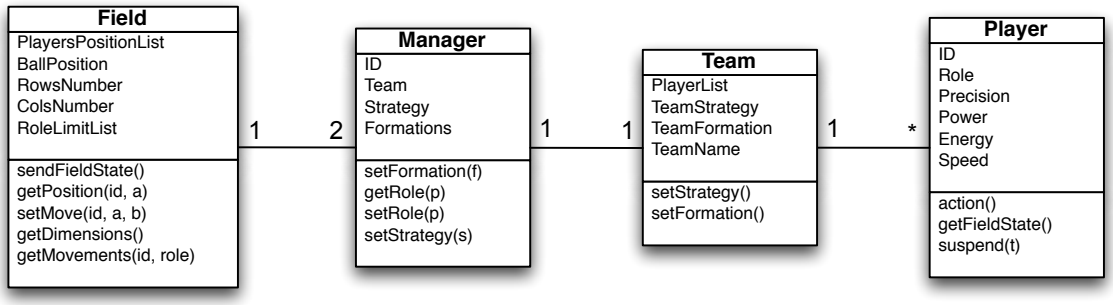

Figura 25: Diagramma delle classi per il manager della squadra.

Come descritto nel diagramma sopra esposto, il manager ha una relazione diretta con i giocatori attraverso la classe *Team*" con la quale può visionare lo stato dell'affaticamento ed eventuali ulteriori caratteristiche proprie di ogni singolo elemento. Inoltre è direttamente collegato con la classe squadra che raggruppa tutti i giocatori per apportare eventuali variazioni a cui è soggetta tutta la squadra piuttosto che un singolo giocatore specifico.

Può inoltre accedere alla classe *Field* che gli permette di conoscere il posizionamento dei giocatori in campo nonchè quello della palla attraverso la funzione *getFieldState()*, uguale a quella utilizzata dai giocatori. Anche questi fattori sono determinanti al fine di prendere decisioni per la propria squadra.

A seguito dei risultati della propria elaborazione, possono essere effettuati interventi come:

- cambio di formazione;
- sostituzione di un giocatore;
- cambio di posizione tra giocatori in campo;
- cambio della propensione di gioco (attacco/difesa);

Nel diagramma sottostante invece viene analizzato lo schema di comunicazione degli stati da parte del Manager che, come già visto, interroga il campo e i giocatori per ottenere il quadro della situazione per effettuare le variazioni necessarie.

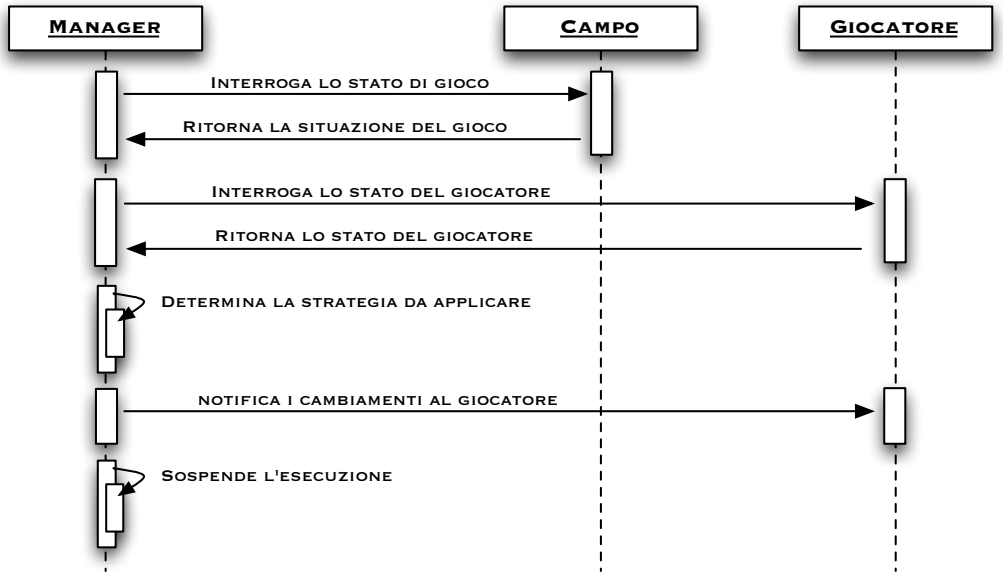

Figura 26: Diagramma di sequenza del manager.

Anche il Manager deve avere una priorità differente rispetto a quella dei giocatori in campo. Rimandiamo alla sezione 5.3 per ulteriori chiarimenti.

4.5 Squadra

La classe squadra è una classe molto semplice nella nostra realtà di gioco. È l'elemento che funge da contenitore per i giocatori e che permette di raggrupparli in due insiemi differenti.

La classe squadra permette di facilitare il coordinamento dei giocatori, la diffusione di alcune specifiche impartite dal Manager a tutti i giocatori della squadra in campo e non soltanto a qualche istanza di giocatore particolare. Il modulo in campo e lo sbilanciamento della squadra in attacco o in difesa sono caratteristiche generali e l'oggetto di riferimento è la classe squadra, piuttosto che tutti i singoli giocatori che le appartengono.

I metodi principali appartenenti alla classe *Team* sono:

procedure SetStrategy(S : in Strategy) che imposta la strategia di gioco;

procedure SetModule(M : in Module) che setta il modulo di gioco;

function GetStrategy return Strategy restituisce ai giocatori ed al manager la strategia applicata;

function GetModule return Module restituisce ai giocatori ed al manager il modulo di gioco in uso;

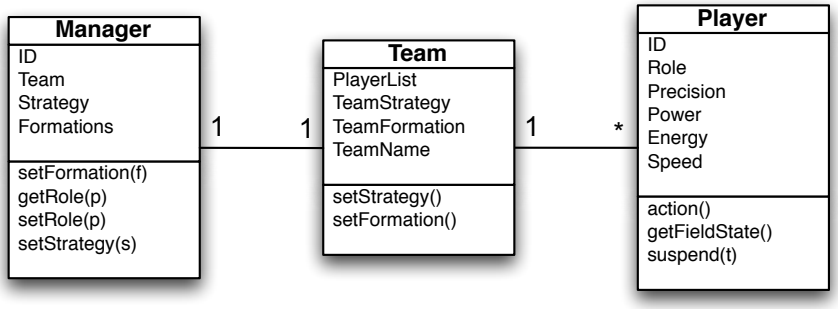

Figura 27: Diagramma delle classi per la classe squadra.

Manager è l'unica entità in grado di accedere direttamente alla squadra ed effettuare variazione. I giocatori subiscono tali modifiche in riferimento alla classe squadra di cui fanno parte e leggono le modifiche apportate attraverso i comandi *Get*.

4.6 Giocatore

I giocatori sono i task di base necessari per lo svolgimento del gioco e vengono intesi come multiple istanze dello stesso task.

Ognuno di essi è contraddistinto da una serie di caratteristiche proprie e legate al ruolo, alla posizione e alle abilità:

- **ID:** ogni giocatore possiede un ID univoco generale che permette di identificare il processo rispetto tra tutti gli altri processi giocatore in campo al fine di determinare il possessore della risorsa condivisa o identificarne nel vettore delle posizioni in campo;
- **Ruolo:** determina il posizionamento in campo e le possibilità di movimento all'interno dell'area specifica di gioco.
- **Visibilità:** è il parametro che determina l'abilità di visione del gioco di ogni singolo giocatore.
- **Potenza:** è il parametro che determina la potenza di tiro e di contrasto del giocatore. È un valore tra 0 e 100.
- **Precisione:** contraddistingue l'abilità del giocatore in questione nel giostrare il pallone.
- **Velocità:** identifica la velocità del giocatore ovvero la quantità di movimenti che può compiere rispetto ad altri.
- **Energia:** rappresenta la resistenza del giocatore durante la partita. È un parametro che decrementa (in modo lineare o esponenziale) con il proseguire delle azioni di gioco fino alla fine della partita.

Ogni giocatore è definito da un processo che compie in maniera ciclica tre azioni base:

- **getFieldState():** il giocatore apprende lo stato del gioco ed in particolare il posizionamento dei compagni, degli avversari e della palla;
- **action():** sulla base dello stato del gioco e limitatamente alla propria visibilità, ogni giocatore effettua una mossa tra quelle definite nelle logiche di gioco a disposizione;

- **suspend():** al termine dell'azione, ogni giocatore sospende la propria esecuzione per un tempo definito. Al termine dell'attesa procede con una nuova azione;

Il diagramma esposto identifica quali sia la sequenza delle azioni che ogni giocatore compie durante la propria fare di gioco. Inizialmente procede con una interrogazione del campo al fine di ottenere lo stato del gioco. Il campo risponde con un vettore contenente i posizionamenti dei vari giocatori in campo nonchè quella del pallone. Segue immediatamente una fase di analisi dell'area "visibile" al fine di determinare quale sia l'azione ottimale da compiere. L'azione viene portata a termine e viene inviato al campo il posizionamento del giocatore a seguito della manovra effettuata. Dopo la notifica, il processo si sospende per un tempo t noto al fine di lasciare agli altri processi gicatore la possibilità di operare a loro volta.

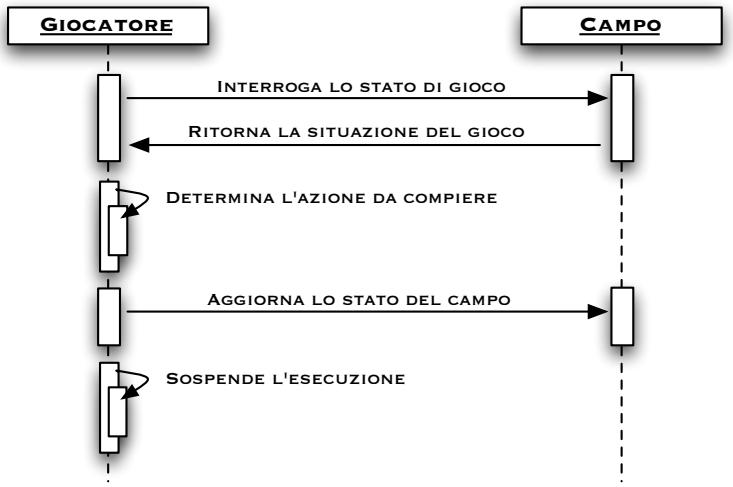

Figura 28: Diagramma di sequenza del giocatore.

Il diagramma degli stati presenta in forma semplificata i vari stadi in cui il giocatore si può trovare in ogni istante di tempo.

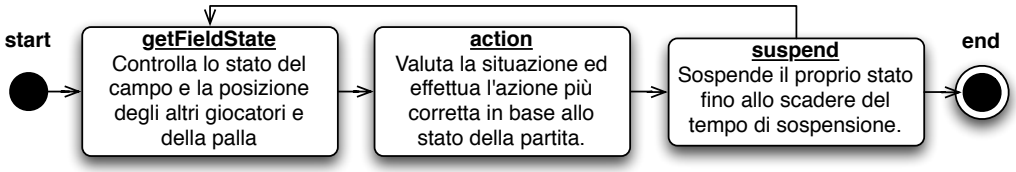

Figura 29: Diagramma di stato dei giocatori in campo.

La parte più consistente del processo è tutta contenuta nello stato *action*, il quale si occupa di valutare le posizioni ed applicare la logica opportuna. Lo schema che segue indica dettagliatamente, passo per passo, tutte le funzioni che vengono applicate per creare il quadrato visibile al giocatore al fine di consentirgli il passaggio ad altri compagni o movimenti in una delle quattro direzioni possibili. All'interno dello schema esistono due diramazioni, legate alla possibilità che il giocatore detenga o meno il possesso di palla oppure che la squadra sia in possesso di palla attraverso un compagno. Nel caso entrambi i test siano negativi, prendiamo come regola che gli avversari siano in possesso del pallone, senza applicare ulteriori test.

Figura 30: Diagramma dell'azione di gioco del giocatore.

Un'ulteriore caratteristica che diversifica il gioco è la possibilià di impostare per ogni singolo giocatore tempi di *suspend* differenti attraverso la valorizzazione del relativo parametro. Il risultato ottenuto è così quello di avere tempo di reazione differenti legati alla dimensione del tempo di sospensione. Giocatori più reattivi possiederanno tempi di sospensione più bassi. Al contrario, giocatori più lenti avranno parametri via via più elevati fino ad un limite ragionevole.

Andiamo ad analizzare infine la situazione della memoria e della CPU durante il gioco nel diagramma di Garrett esposto. Ogni giocatore comincia la partita in una coda di processi pronti. Nel primo istante utile, un processo prende possesso della CPU e compie le azioni viste precedentemente fino allo scadere del tempo di utilizzo del processore o fino alla sospensione. Una volta sospeso, entra in un'altra coda, quella dei processi sospesi, e attende il termine del tempo di sospensione t. Una volta pronto, passa nuovamente nella coda dei processi pronti per procedere con una nuova azione non appena avrà a disposizione la CPU.

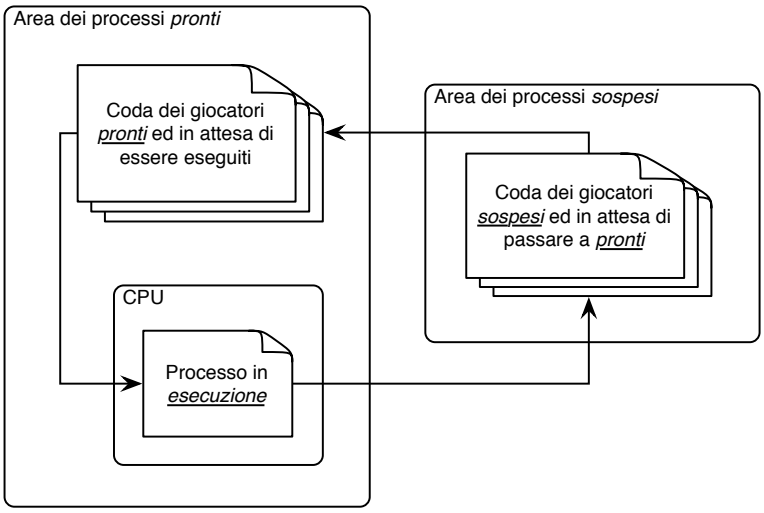

Figura 31: Diagramma di stato dei giocatori in campo.

4.7 Pallone

Il pallone rappresenta la risorsa condivisa nel campo di gioco. Ogni giocatore concorre per ottenerne il possesso e poter condurre la palla nella rete avversaria e segnare un gol in favore della propria squadra.

Il pallone è rappresentato nel campo come una risorsa condivisa il cui accesso da parte dei giocatori è limitato e regolato da un monitor. Ciò preclude la possibilità che la palla possa essere in possesso di più di un giocatore contemporaneamente. Ada 95 mette a disposizione il costrutto *protected type* che si occupa di costruire e gestire i monitor per l'accesso in mutua esclusione alla risorsa.

Dal punto di vista logico possiamo immaginare il pallone come un oggetto sul campo di gioco il quale viene spostato dai giocatori attraverso avanzamenti con palla, passaggi, tiri in porta o contrasti. Dal punto di vista tecnico, si tratta di un'entità passiva che "subisce" le azioni che vengono compiute su di esso da parte dei giocatori.

In questo caso la classe si riduce ad un oggetto protetto con pochi metodi accessibili per ottenere la sua posizione, l'eventuale possessore, effettuare un movimento della sfera in una delle 4 direzioni o eventualmente settarne il possessore.

La risorsa protetta contiene una variabile fondamentale che permette di identificare il thread che ne detiene il possesso. In questa maniera, anche nel caso in cui il giocatore possessore della palla entri in stato di *sleep*, una guardia garantirà che nessun altro dei 21 giocatori rimanenti possa accedere alla risorsa.

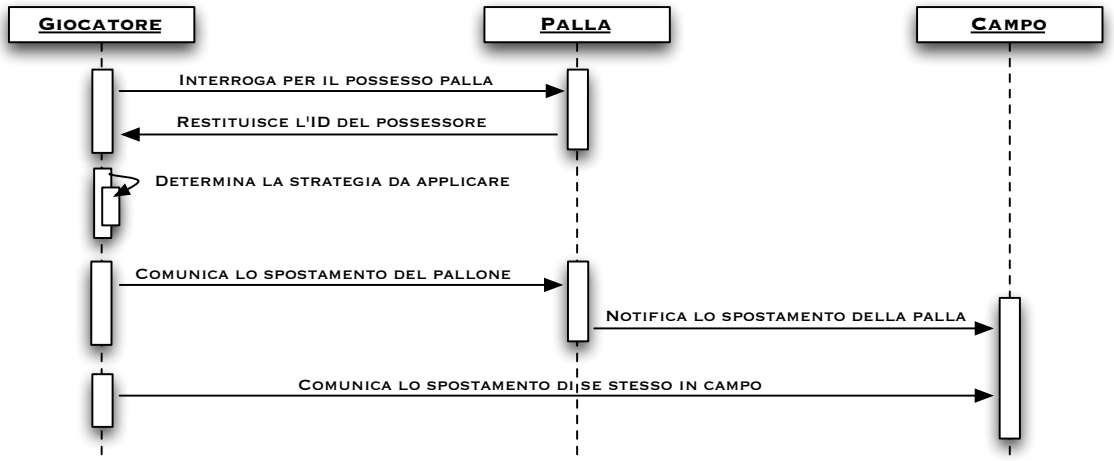

Figura 32: Diagramma di sequenza dello spostamento della palla in campo.

Al fine di gestire il movimento della palla è stato creato un thread apposito *TaskBall* che si occupa di rilevare e gestire il movimento del pallone
Per maggiori dettagli sui metodi legati alla risorsa protetta rimandiamo alla sezione 5.2 in cui vengono analizzati gli aspetti relativi alla concorrenza per un oggetto condiviso.

5 Concorrenza

Prima di entrare nel dettaglio dei problemi legati alla concorrenza, cerchiamo di identificare chiaramente quali sono le entità che fanno parte del nostro gioco e come interagiscano al fine di un corretto funzionamento:

Entità Attive: rappresentate dai giocatori in campo, dal manager che li dirige e dall'arbitro come direttore di gara. Sono i thread attivi che compiono movimenti sul campo secondo scelte determinate in base alla situazione che rilevano nel momento in cui vengono attivati. Lo stesso vale per i manager che rivedono posizioni e formazione in campo in base al risultato e all'andamento della gara. L'arbitro monitora lo stato di gioco e, in base a determinati eventi, decide l'andamento della gara attraverso modifiche allo stato di gioco.

Entità Reattiva: identificata dal pallone e dal campo di gioco. Rappresentano entità che non hanno potere decisionale, ma subiscono variazioni sulla base dell'intervento dei giocatori che ne hanno via via il possesso. Il pallone può muovere assieme al giocatore in caso di avanzamento con palla o in maniera autonoma nei casi di passaggio o tiro in porta. Il movimento della palla è comunque sempre subordinato alla potenza impressagli dal giocatore che effettua l'azione. Il campo subisce tutte queste variazioni e si occupa di mantenere uno stato di gioco consistente.

Le situazioni di concorrenza che dobbiamo necessariamente analizzare e risolvere al fine di perfezionare il gioco ed eliminare situazioni impreviste sono fondamentalmente:

- integrità dello stato di gioco;
- unicità del possesso di palla;
- prevenzione di situazioni di stallo;

Esaminiamo perciò ognuna di esse e forniamo gli elementi necessari per la loro risoluzione.

5.1 Integrità dello Stato di Gioco

Avere 22 processi che modificano costantemente lo stato del campo di gioco in un ciclo di durata pari al tempo totale di gioco può causare facilmente seri problemi di inconsistenza a livello di campo di gioco. In questo senso è necessario prevedere tale situazione ed utilizzare tecniche di programmazione che permettano di evitarla per poter procedere sempre correttamente con l'aggiornamento dello stato e dare la possibilità ai thread che lo necessitano, di conoscere la condizione di gioco.

Le strade percorribili sono fondamentalmente due:

1. instanziare la classe *Field* come processo;
2. creare la classe *Field* come tipo protetto;

La prima soluzione, più simile ad una struttura di tipo client/server, garantisce attraverso l'utilizzo di una struttura di tipo *select* a cui fanno seguito una serie di *accept* legate alla scrittura e alla lettura, accesso esclusivo al thread che gestisce i posizionamenti nel campo. La soluzione è percorribile se immaginiamo di istanziare su macchine differenti i giocatori e il thread del campo di gioco.

Diventa perciò più semplice gestire la classe *Field* come un tipo protetto che garantisce la mutua esclusione delle operazioni che intervengono sulla classe, mantenendo pertanto la correttezza dello stato di gioco attraverso la gestione da parte del linguaggio degli accessi alle variabili protette. La soluzione non impatta direttamente sul processore in quanto la CPU non viene occupata direttamente dalla classe *Field* ma dai thread dei giocatori che vi devono accedere.

La classe *Field* è pertanto dotata dei metodi:

function GetPositions return PositionList che permette di interrogare il campo ed ottenere il vettore contenente le posizioni di tutti i giocatori nell'area di gioco, compresa quella del giocatore interrogante. La funzione, come specifica del linguaggio Ada 95, ha come prerogativa l'accesso ai dati in sola lettura, evitando la modifica del campo di gioco. Questo permette l'accesso concorrente in *read only* al vettore delle posizioni;

procedure SetPosition(ID : in Integer; M : in Move) da la possibilità al giocatore che lancia la procedura di modificare la propria posizione nel campo di gioco, indicando il proprio ID univoco e la tipologia di movimento come da specifiche già trattate nella sezione 2.2.1. In questo modo viene aggiornata la posizione nel vettore senza incorrere in problemi di inconsistenza;

function GetBallPosition return Position permette al giocatore di conoscere la posizione della palla all'interno del campo di gioco. È necessaria al fine di permettere al processo di decidere quale azione compiere, limitatamente alla propria visibilità di gioco;

Queste tre funzioni da sole permettono di ottenere tutte le informazioni necessarie al corretto svolgimento del gioco non solo per i thread giocatori, ma anche eventualmente a manager ed arbitri per effettuare decisioni o correzioni durante lo svolgimento della partita. La garanzia di mutua esclusione è determinata dalla stessa classe *Field* in quanto dichiarata come risorsa protetta ad accesso limitato.

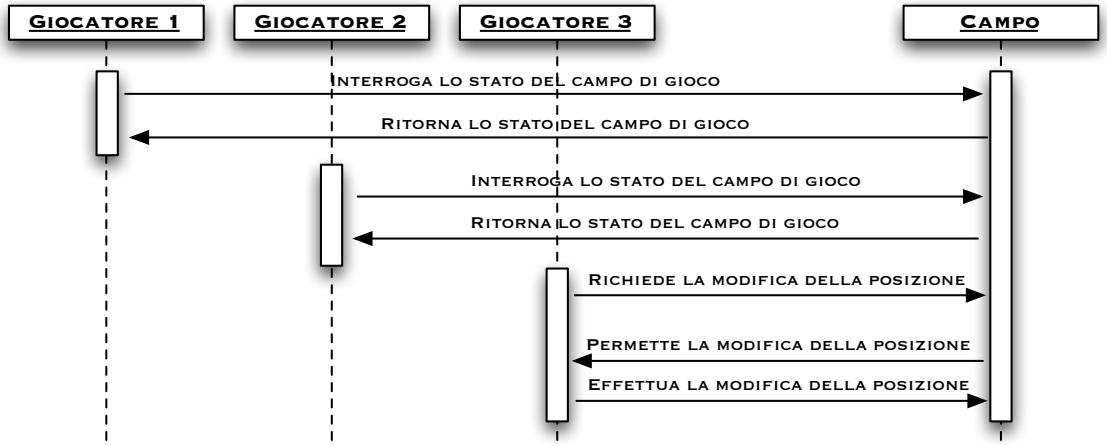

Figura 33: Diagramma di sequenza delle richieste dei giocatori per la notifica e la modifica delle posizioni.

Nello schema sopra esposto è possibile notare come la lettura delle posizioni in campo sia un processo di lettura concorrente, mentre l'aggiornamento della posizione del giocatore sia un'operazione esclusiva. In questo senso, una volta terminate le eventuali operazioni di lettura, il processo prende il controllo della risorsa, effettua il cambiamento e la rilascia.

5.2 Unicità del Possesso di Palla

Un altro problema evidente legato alla concorrenza tra thread è quello del possesso palla da parte dei giocatori in campo. Il pallone risulta l'unica istanza esistente e non è realistico, nella nostra realtà virtuale così come nella realtà di gioco vero e proprio, la condizione in cui due giocatori distinti detengano contemporaneamente il possesso del pallone.

Si rende pertanto necessario l'utilizzo di un meccanismo che garantisca il mutuo accesso alla risorsa. Anche in questo senso Ada 95 ci viene incontro fornendo uno strumento per la gestione delle risorse protette. Dichiarando la palla come risorsa protetta, garantiamo l'accesso esclusivo ai dati in essa contenuti.

I metodi utilizzabili con la classe palla sono:

function GetOwner return Integer permette di ottenere l'ID dell'attuale possessore del pallone o il valore *null* nel momento in cui non esistesse in quell'istante un possessore della risorsa;

function GetBallPosition return Position ritorna la coordinata relativa alla posizione del pallone con lo scopo di prenderne possesso nel caso si trovi in intersezioni adiacenti a quella del giocatore che sta effettuando la richiesta;

procedure SetOwner(ID : in Integer) è forse il metodo principale della classe. Esso permette di settare il valore della variabile relativa al proprietario della risorsa. Essendo una procedura, garantisce la mutua esclusione nell'attività attraverso il meccanismo implicito del monitor, impedendo che un giocatore, una volta flaggato il *lock* sulla risorsa, non possa settare il proprio valore ID nella variabile a causa di una sospensione. Una situazione di questo genere porterebbe ad un'evidente inconsistenza di sistema, causando l'impossibilità di accedere alla risorsa da parte di tutti gli altri giocatori per tutta la durata del match;

procedure KickBall(M : in Move; P : in Interger) rappresenta la liberazione della risorsa attraverso un ideale "calcio". In questa procedura, anch'essa atomica ed indivisibile, il giocatore rimuove il blocco sulla risorsa settando la variabile a *null* e applica una potenza al colpo in una direzione determinata;

Come abbiamo avuto modo di esaminare per la classe *Field*, i metodi appena visti garantiscono accesso esclusivo alla risorsa per eventuali modifiche al suo stato interno, mentre garantiscono accesso concorrente alle informazioni, permettendo a tutti i giocatori di identificarne proprietario e posizione, ma assicurando l'integrità della risorsa.

5.3 Priorità dei Processi

Al fine di gestire il corretto andamento del gioco, è necessario gestire le priorità dei thread. Se così non fosse, ognuno di essi entrerebbe nella coda dei pronti e dovrebbe attendere diversi cicli di CPU prima di poter eseguire. Per questo motivo è necessario raggruppare le tipologie di processi e ordinarle seguendo una logica prioritaria d'esecuzione.

I processi più comuni sono quelli dei giocatori in campo. Tutti i giocatori eseguono con la stessa priorità, attraverso una politica di ordinamento di tipo "FIFOWithinPriorities", demandando eventuali gestioni ulteriori allo scheduler di sistema che li tratterà come processi ordinari.

Con priorità maggiore intervengono i manager delle squadre che, una volta pronti, devono entrare in esecuzione prima dei giocatori in stato di pronto, al fine di modificare le condizioni di gioco preventivamente rispetto al successivo intervento dei giocatori che dovranno già riscontrare la situazione variata.

Infine l'arbitro, in qualità di giudice di gioco deve poter intervenire immediatamente interrompendo il gioco, effettuando eventuali riposizionamenti di giocatori e di palla, prima di ogni giocatore e ancor prima delle decisioni dei manager.

Classe	Priorità	Descrizione
Giocatore	1	Priorità minima di accesso alla CPU.
Manager	2	Priorità superiore a quella dei giocatori.
Arbitro	3	Priorità massima.

Tabella 7: La lista delle priorità dei processi.

In particolare, poichè l'arbitro reagisce a situazioni che ne richiamano l'attenzione in maniera del tutto similare ad una "entità reattiva"
In questo modo viene garantito il corretto accesso al processore ed eventuali modifiche che devono essere applicate al gioco vengono recepite immediatamente da tutti i processi con priorità inferiore per effetto del'ordine di esecuzione.

5.4 Prevenzione delle Situazioni di Stallo

Avere un numero elevato di processi che concorrono ad un'unica risorsa può portare facilmente a situazioni di stallo. Elenchiamo per completezza le tre situazioni in cui il sistema non è in grado di procedere oltre determinando una partita infinita o immodificabile fino alla fine del tempo di gioco:

- deadlock;
- starvation;
- lockout;

Per non incappare nelle situazioni di cui sopra è necessario risolvere una o più tra le precondizioni elencate:

Mutua esclusione: non più di un processo alla volta ha accesso alla risorsa condivisa;

Cumulazione di risorse: i procesi possono accumulare risorse e trattenerle mentre attendono di acquisirne altre;

Assenza di prerilascio: le risorse vengono rilasciate solo volontariamente;

Attesa circolare: un processo attende una risorsa in possesso del successivo processo in coda;

Il primo punto è già stato esaurientemente coperto e risolto nelle sezioni 5.2 e 5.1. Non ci dilunghiamo oltre nell'analisi e nella risoluzione ma rimandiamo ad essi per ulteriori approfondimenti.

Il secondo punto, in parte, è risolto per definizione: il prerequisito di unicità della palla nel campo di gioco è sufficiente per poter rendere inattuabile la precondizione ed evitare situazioni di stallo legate ad esso. Il giocatore che detiene il possesso della sfera non limita le possibilità di gioco degli altri giocatori che possono procedere nella loro strategia fino ad ottenerne il possesso per loro o per la propria squadra. Per quanto riguarda l'altra risorsa protetta, ovvero il campo di gioco, le azioni di spostamento sono atomiche e non comportano il blocco della risorsa ne il possesso di risorse multiple che possono determinare il blocco della procedura. Una volta settata la nuova posizione, la risorsa viene rilasciata immediatamente per eventuali consultazioni o ulteriori modifiche.

La terza precondizione coninvolge per lo più il possesso di palla e non è risolvibile "realisticamente" se non utilizzando un sistema di rilascio forzato della risorsa attraverso, per esempio, un meccanismo di conteggio dei movimenti o dei tempi di possesso. Possiamo assumere che dopo n movimenti consecutivi con palla, il giocatore debba necessariamente effettuare il passaggio ad un compagno o tentare un tiro in porta. Questi due eventi però sono legati alla visibilità del giocatore in questione e possono non poter essere soddisfacibili nel momento in cui viene raggiunto il valore limite. Non è pertanto una soluzione ottimale. È possibile

eventualmente conteggiare gli istanti temporali in cui il giocatore trattiene il pallone. A seguito di t istanti, il processo deve liberare la risorsa, ma come nel caso precedente, si possono verificare condizioni in cui, per effetto della visibilità di gioco non ci siano compagni vicini e la porta sia troppo distante per poter tirare.

Altro tipo di approccio dalle linee decisamente più semplicistiche, ma molto più realistico, è quello di assumere che nel corso della partita il giocatore con palla si trovi in condizione di tirare, passare o subire da *1* a x contrasti e questo comporti inevitabilmente, nel corso del match, la perdita del pallone. Analizziamo il nostro modello per comprendere cosa può succedere nel caso in cui un giocatore non liberi la palla:

1. **Avanzamento:** il giocatore continua ad avanzare nel campo di gioco con la palla. Nel caso non subisca contrasti da parte di avversari o vinca tutti quelli sostenuti, arriverà in prossimità della porta. Per effetto delle logiche di gioco, effettuerà il tiro per tentare di segnare una rete rilasciando la risorsa. Nel caso non tenti il tiro e continui a trattenere il pallone, possiamo assumere che a seguito di n contrasti durante la fase di avanzamento, in almeno uno risulti sconfitto e perda il possesso del pallone, rilasciandolo.

2. **Assenza di compagni per il passaggio:** assumiamo che concorrentemente al giocatore con palla anche tutti i compagni in gioco continuino a muoversi per smarcarsi da avversari e ricevere il pallone. È lecito ritenere che almeno un giocatori visibile dal portatore di palla sia smarcato per dare modo al nostro thread di passare il pallone.

3. **Assenza di porta avversaria:** come già visto nel punto 1 del presente elenco, l'avanzamento porta inevitabilmente il giocatore a trovarsi di fronte alla porta avversaria e questo gli permetterà di effettuare il tiro.

4. **Mantenimento della posizione:** nel peggiore dei casi, il giocatore non effettuerà alcuno spostamento. In questo caso possiamo assumere che sia il gioco svolto intorno ad esso ad essere dinamico e questo porta inevitabilmente all'avvicinamento da parte di compagni a cui passare la risorsa o di avversari che tenteranno di effettuare un contrasto per sottrarre il pallone. Anche in questo caso è perfettamente lecito pensare che un giocatore, anche se fermo, non si trovi mai nella condizione di passare o di perdere la palla.

Ai fini della nostra realtà, quanto asserito nei punti precedenti è sufficiente per poter considerare risolta l'assenza di prerilascio per effetto della costruzione del nostro sistema di gioco. In questo caso possiamo affermare la solidità del modello di gioco rispetto alla mancanza di prerilascio.

La quarta precondizione non si verifica per ipotesi. Per quanto riguarda l'accesso alla classe *Field*, la modifica dello stato di gioco avviene istantaneamente a seguito dell'applicazione di una delle logiche di gioco. Questo non comporta attesa circolare in quanto un processo che ottiene accesso alla risorsa *Field* la libera prima di sospendere la propria esecuzione. Questo comporta che il processo successivo in coda si troverà in condizione di accedere anch'esso alla risorsa, liberandola prima di sospendersi. Per quanto riguarda invece la risorsa *Ball*, nessun giocatore rimane immobile in caso di impossibilità di accedere alla risorsa, ma compie azioni legate all'avanzamento verso la porta avversaria o al mantenimento della propria posizione in base alle condizioni di gioco con cui si trova ad interagire.

Alla luce di quanto visto finora possiamo permetterci di escludere il verificarsi di situazioni di stallo che comportino l'impossibilità di portare a termine la partita.

5.5 Prevenzione delle Situazioni di Starvation

Con il termine "starvation" si intende la situazione in cui un processo pur nella condizione di agire non ottiene mai le risorse di cui necessita al fine di portare a termine la propria esecuzione.

Contestualizzate nel nostro gioco, le situazioni in cui un processo può entrare in uno stato di starvation sono:

- assenza del possesso palla;
- impossibilità nell'ottenere i posizionamenti in campo per blocco della risorsa;
- impossibilità di movimento del giocatore;

Il primo caso però non conduce mai in una situazione del tipo sopra descritto: pur non avendo a disposizione la risorsa "palla", ogni giocatore può continuare la propria esecuzione assistendo il gioco dei compagni attraverso movimenti nel campo di gioco. L'avanzamento come i movimenti laterali sono sempre possibili entro i limiti delle aree di movimento proprie di ogni giocatore. Nel caso un avversario prenda il controllo della palla, avviene un arretramento automatico che porta vicino alla linea difensiva finchè l'avversario non entra nello spettro visibile, momento in cui avviene un pressing e un eventuale contrasto per il possesso della palla.

Il secondo caso non avviene per definizione: la risorsa condivisa viene catturata dai giocatori che necessitano di ottenere i posizionamenti, recuperano la lista delle posizioni e la rilasciano istantaneamente per dare la possibilità agli altri di ottenere le posizioni di compagni e avversari. In questo modo la risorsa e sempre a disposizione degli altri processi che la richiedono. Nel caso in cui un giocatore effettui una modifica ai posizionamenti, trattiene la risorsa, modifica la variabile della posizione e la rilascia. L'effetto ottenuto è un rilascio costante della risorsa "Field" che rende impossibile stati di starvation.

Nel terzo caso un giocatore non ha possibilità di muoversi solo quando:

1. ha raggiunto il limite dell'area di movimento per il proprio ruolo in una o due direzioni (quest'ultimo nel caso abbia raggiunto un angolo);
2. le altre direzioni disponibili sono occupate da altri giocatori;

In questo caso però possiamo fare un ragionamento per assurdo: se il giocatore *G1* viene "messo all'angolo" da altri due o tre giocatori, definiti come *G2* e *G3*, dobbiamo tenere in considerazione il movimento della palla o del giocatore *G4* che la possiede. *G2* e *G3* si muoveranno a loro volta per avvicinarsi all'obiettivo, sia che siano compagni di squadra, sia che siano avversari. Risulta fortemente improbabile pertanto che i giocatori *G2* e *G3* non liberino una delle celle che stanno occupando. In tal caso il giocatore *G1* in stato di blocco ha la possibilità di muovere nella prima cella utile, presupponendo il fatto che seguirà il movimento del giocatore *G2* o *G3* che si è spostato a sua volta per seguire l'azione.

Per effetto delle considerazioni sopra esposte possiamo affermare che situazioni di starvation si possono verificare temporaneamente durante il gioco in casi molto rari. È altrettanto possibile affermare, per come è stato progettato il software, che tali situazioni vengano risolte con il procedere del gioco stesso e non comportino problemi di integrità consistenti a livello di runtime.

6 Distribuzione

Il concetto di distribuzione prevede che un sistema residente su di un unico calcolatore venga scomposto in parti al fine di poter essere suddiviso in una moltitudine calcolatori differenti. La distribuzione di un sistema permette di ripartire il carico di lavoro: quello che dovrebbe essere imputato ad un solo calcolatore viene suddiviso su una serie di macchine, ognuna di esse con una funzione determinata e dei compiti precisi. Il modo in cui viene implementata la distribuzione è completamente trasparente al sistema poichè è direttamente progettata all'interno del linguaggio stesso.

Ada 95 permette di generare in modo estremamente semplice un sistema completamente distribuito partendo dallo sviluppo di un sistema monolitico e definendo in seconda battuta le componenti distribuite attraverso l'uso di alcune direttive per determinare la tipologia di comunicazione e di oggetti nonchè attraverso un file di configurazione che aiuta il compilatore nella generazione dei vari eseguibili. Ognuno di essi conterrà al suo interno gli "stub" generati dal compilatore. Attraverso questi riceverà ed invierà messaggi alle varie componenti con cui è direttamente collegato.

Ai fini della nostra realtà virtuale, la soluzione scelta è basata su un'architettura stile client/server distribuita nella quale possiamo immaginare che tutta la partita si svolga all'interno del server e solo gli eventi principali vengano trasmessi ai client distribuiti. Gli "stub" si occupano di gestire la comunicazione tra le partizioni, convertendo messaggi in ingresso ed uscita e facendo credere al software che tutto il sistema risieda su un'unico calcolatore.

Per quanto riguarda la realizzazione del progetto, è stato sviluppato un modello specifico per mettere in pratica la possibilità di distribuire il sistema. Vediamo nel dettaglio di che cosa si tratta.

6.1 Progettazione e Struttura Distribuita del Software

Il software distribuito è composto da tre entità fondamentali: una entità servente che si occupa di generare una serie di eventi relativi all'avanzamento dello stato di gioco, una entità che si occupa di gestire una coda nella quale vengono accodati gli eventi e infine una entità cliente che si occupa di visualizzare nei terminali coinvolti gli eventi accodati.

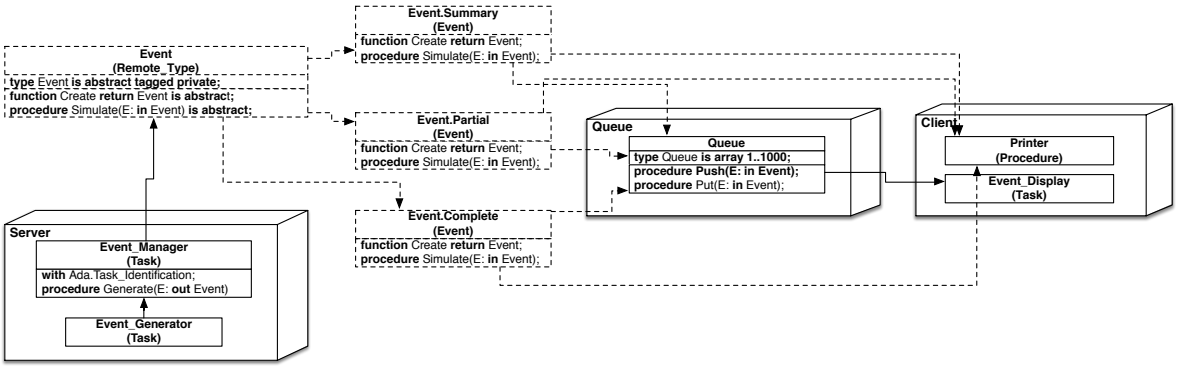

Figura 34: Diagramma del sistema distribuito.

La comunicazione tra le parti avviene attraverso la trasmissione di eventi legati all'avanzamento dello stato di gioco. Trasmettere gli eventi non è propriamente la definizione più adatta ma ci occuperemo nelle parti successive di ridefinire nel dettaglio l'effetto trasmissione.

La struttura distribuita, come già menzionato è composta in linea generale da tre elementi funzionali:

- **Server:** si occupa della generazione degli eventi sul campo di gioco;
- **Queue:** gestisce l'accodamento degli eventi ricevuti all'interno di una coda e rende disponibili due metodi Push e Pop per potervi accedere attraverso operazioni di inserimento ed estrazione;
- **Client:** permette la visualizzazione degli eventi generati dal sistema di gioco su molteplici display distribuiti;

Ognuno di queste strutture è a sua volta composto da una serie di task necessari al corretto funzionamento del meccanismo. Andiamo ad analizzare nel dettaglio la composizione delle parti:

- **Server:** genera e accoda eventi relativi allo stato di gioco, registrandone le statistiche:
  - **Event Generator:** produce eventi casuali simulando la struttura di una partita reale;
  - **Event Manager:** si occupa di ïmpacchettarelë informazioni ricevute dall'Event Generator per produrre un oggetto che sia trasmissibile al gestore della coda in modalità distribuita. Per alcuni degli eventi ricevuti che corrispondono a particolari tipologie definite, si occupa anche della produzione di statistiche di gioco attraverso la chiamata di una procedura che modifica un record apposito contenuto in Stats;
  - **Stats:** rappresenta il registro delle statistiche di gioco;
- **Queue:** attraverso due procedure protette è possibile effettuare l'inserimento (Push) in coda degli eventi prodotti e l'estrazione (Pop) degli stessi per accedere alle informazioni contenute nei pacchetti;
- **Client:** permette la visualizzazione degli eventi generati dal server sul display. Possono esistere moltepli istanze di client, ognuna delle quali con il proprio Event Display e Printer;
  - **Event Display:** accede alla coda ed effettua l'estrazione degli eventi (se disponibili) e richiama la procedura per la stampa dell'evento a video;
  - **Printer:** riceve i valori contenuti nell'evento e li manda a video attraverso i metodi canonici di stampa;

6.2 Struttura degli Eventi

La struttura delle classi relative agli eventi è stata costruita seguendo un pattern basato su una classe generica Basic Event da cui derivano tre classi figlie specializzate, ognuna con l'implementazione dei propri metodi.

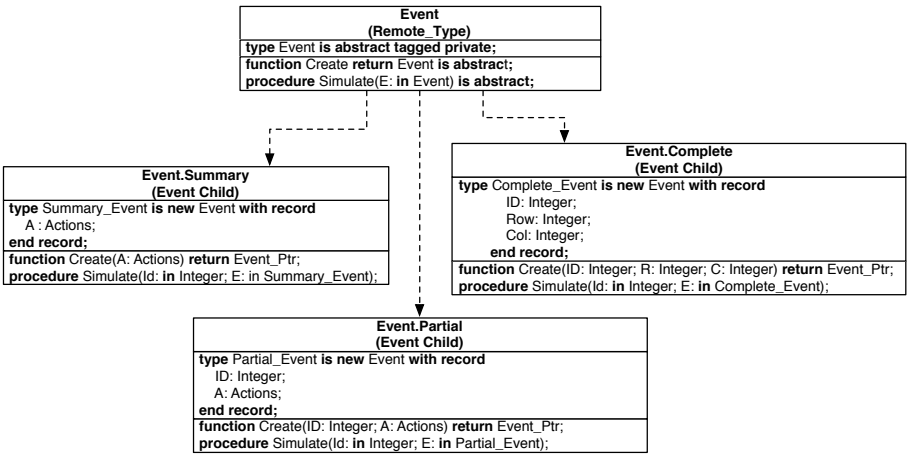

Figura 35: Strutturazione degli eventi nel modello.

Nella figura sopra possiamo vedere l'albero di derivazione delle classi in cui sono presentate la classe base astratta e le classi figlie specializzate. Il principio alla base della struttura è la possibilità di sfruttare il polimorfismo nativo delle classi figlie che possono essere passate indifferentemente a tutti i metodi che richiedono un parametro di tipo Basic Event. In questa maniera non abbiamo necessità di preoccuparci di quale sia il tipo di evento con cui abbiamo a che fare, in quanto il loro padre è una classe astratta comune. Andiamo ad analizzare nel dettaglio la codifica delle classi con alcuni estratti del codice:

- **Basic Event:** la classe è astratta e di tipo Remote Type. Questo garantisce che sia accessibile in maniera distribuita da tutte le parti del software che hanno necessità di richiamarla. Il metodo astratto Create è necessario per la creazione dell'evento e per la restituzione del suo puntatore. Il metodo Simulate, anch'esso astratto, serve per lanciare un'azione specifica legata all'evento particolare. Poichè astratti non hanno un'implementazione concreta ma solo una funzione di interfaccia per i figli di Basic Event.

```
1   package Basic_Event is
3      pragma Remote_Types;
5      type Event is abstract tagged limited private;
       type Event_Ptr is access all Event'Class;
7
       pragma Asynchronous (Event_Ptr);
9
       function Create return Event_Ptr is abstract;
11      procedure Simulate(Id: Integer; E: Event) is abstract;
13  private
       type Event is abstract tagged limited null record;
15  end Basic_Event;
```

  - **Summary Event:** è la più semplice implementazione della classe Basic Event e prevede un tipo record composto da un solo elemento Actions, oltre all'implementazione concreta dei due metodi astratti: Create necessario per la creazione dell'evento e Simulate per l'invio alla classe di stampa del contenuto dell'evento generato.

```
1       package Basic_Event.Summary is
3          type Summary_Event is new Event with private;
           type Summary_Ptr is access all Summary_Event'Class;
5
           function Create(A: Actions) return Event_Ptr;
7          procedure Simulate(Id: in Integer; E: in Summary_Event);
9       private
           type Summary_Event is new Event with
11             record
                  A : Actions;
13             end record;
        end Basic_Event.Summary;
```

  - **Partial Event:** rappresenta in evento similare nella struttura e nell'implementazione a Summary Event di cui è fratello. L'unica differenza consiste nel tipo record protetto e nei metodi che lavorano su di esso. L'evento parziale riporta nello specifico l'ID del giocatore che compie l'azione e la variabile Actions relativa all'azione compiuta.

```
        package Basic_Event.Partial is
2
           type Partial_Event is new Event with private;
4
          function Create(ID: Integer; A: Actions) return Event_Ptr;
6          procedure Simulate(Id: in Integer; E: in Partial_Event);
8       private
           type Partial_Ptr is access all Partial_Event;
10         type Partial_Event is new Event with
               record
12                 ID: Integer;
                   A: Actions;
14             end record;
        end Basic_Event.Partial;
```

– **Complete Event:** anch'esso, in quanto fratello di Summary e Partial, ha la stessa struttura e implementazione. Il record protetto è differente dagli altri due e contiene l'ID del giocatore oltre alla sua posizione realizzata attraverso due valori interi che rappresentano l'intersezione di riga e colonna su cui il giocatore si trova.

```
1   package Basic_Event.Complete is
3       type Complete_Event is new Event with private;
5       function Create(ID: Integer; R: Integer; C: Integer) return Event_Ptr;
        procedure Simulate(Id: in Integer; E: in Complete_Event);
7
    private
9       type Complete_Ptr is access all Complete_Event;
        type Complete_Event is new Event with
11          record
                ID: Integer;
13              Row: Integer;
                Col: Integer;
15          end record;
    end Basic_Event.Complete;
```

6.3 Trasmissione degli Eventi

Dopo aver visto la composizione degli eventi andiamone ad analizzare la trasmissione. Avviene attraverso la creazione di oggetti definiti dalla direttiva *"pragma Remote Type"* (RT) e inviando i loro puntatori (access type) ad oggetti distribuiti attraverso chiamate remote su package realizzati attraverso la direttiva *"pragma Remote Procedure Call"* (RCI). In questo modo, come già detto, è possibile mettere in comunicazione oggetti residenti su partizioni differenti in maniera del tutto trasparente rispetto al sistema.

Andiamo ad analizzare nel dettaglio come avviene la sequenza di comunicazione rispetto agli attori visti nelle sezioni precedenti e rappresentata nell'immagine sottostante:

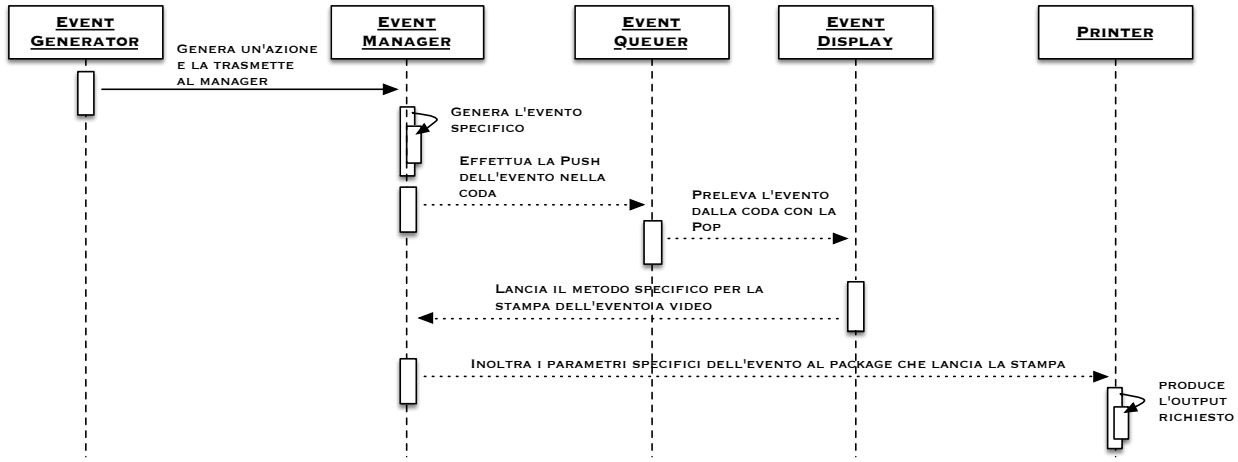

Figura 36: Sequenze del sistema distribuito.

1. **Event Generator**, a seguito della creazione di un evento, richiama l'entry di **Event Manager** per poterne passare i parametri;
2. **Event Manager** riceve i parametri e sulla base di questi genera l'evento creando una nuova istanza del tipo generico **Basic Event** (RT);
3. il metodo generico *Create* rimanda al metodo specializzato dell'evento figlio appropriato che è stato sovrascritto (attraverso overriding);

4. l'access type dell'evento appena creato viene trasmesso alla coda attraverso un'apposita procedura *Push* chiamata in remoto sulla classe **Queuer** (RCI);

5. l'operazione imposta a *True* il valore di una flag apposita che funziona da variabile di controllo per l'accesso alla coda protetta al fine di permettere la *Pop* degli eventi contenuti;

6. quando **Event Display** rileva la variazione della flag lancia la *Pop*, anch'essa da remoto (RCI) per svuotare la coda;

7. una volta ricevuto il puntatore all'evento, **Event Display** richiama l'operazione *Simulate*, locale rispetto alla partizione in cui è residente l'istanza dell'oggetto **Basic Event**, la quale si occupa di attivare la sequenza di stampa specifica dell'evento;

8. l'istanza dell'evento, residente sul server di gioco, richiama così il metodo *Print* di tipo (RCI) della classe **Printer** che si occupa di stampare a video, lato display, le informazioni contenute nell'evento;

9. saltuariamente il cliente richiama il metodo *Get_Stats* proprio del package **Stats** di tipo remoto (RCI) che ritorna la sequenza delle variabili relative alle statistiche di gioco che vengono presentate a video lato cliente;

6.4 Implementazione della Coda

Uno dei requisiti di un sistema distribuito è preservare la consistenza dello stato di gioco nelle varie entità distribuite. Per fare ciò è necessario implementare una strategia attraverso cui le varie parti attingano in maniera ordinata la sequenza di eventi necessari per poter ricostruire lo stato di gioco. La più semplice forma di salvataggio ordinato dei dati è una struttura basata su una coda. Questa funziona da buffer per gli eventi: il gestore degli eventi di gioco, attraverso operazioni di Push controllate e protette, inserisce nuovi eventi all'interno della coda. Il meccanismo è replicato in maniera analoga dall'operazione di Pop, anch'essa di tipo protetto, la quale permette di svuotare la coda prelevandone gli eventi contenuti.

Poichè nel modello è previsto che le entità che accedono alla coda siano due, è stata implementata una struttura di tipo multi-pop queue che permette l'utilizzo di una coda soltanto, con due riferimenti specifici che tengono traccia degli eventi estratti in base all'entità che richiede l'operazione. Analizziamo la struttura del codice relativo alle operazioni fondamentali Push e Pop sulla coda:

- **Queue:** il principio alla base della struttura della multi-pop queue è l'utilizzo di molteplici iteratori, uno per ogni oggetto che può richiamare l'operazione Protected_Pop. Il set delle variabili è configurato per due accessi differenti, ma potenzialmente può essere esteso ad un numero illimitato di attori. Le variabili duplicate sono Head, Tail e Length che servono a tenere traccia del primo evento prelevabile (Head), dell'ultimo evento prelevabile (Tail) e del numero di eventi tra i due iteratori (Length). Duplicandoli non diventa più necessario gestire due code formalmente identiche e con propri iteratori, ma è sufficiente usarne una in comune e tenere traccia degli accessi da parte delle due istanze di display.

```
package Queue is
   pragma Remote_Call_Interface;
   procedure Push (E_Ptr: in Basic_Event.Event_Ptr);
    procedure Pop (Id: in Integer; E_Ptr: out Basic_Event.Event_Ptr);
   pragma Asynchronous (Push);
   private
       Queue_Size : constant := 1000;
       subtype Queue_Range is Positive range 1 .. Queue_Size;
       Data : array (Queue_Range) of Basic_Event.Event_Ptr;
      Tail : Queue_Range := 1;
      -- Parameters for 1th Queue
       Length1 : Natural range 0 .. Queue_Size := 0;
       Head1, Tail1 : Queue_Range := 1;
      -- Parameters for 2nd Queue
       Length2 : Natural range 0 .. Queue_Size := 0;
       Head2, Tail2 : Queue_Range := 1;
end Queue;
```

- **Protected_Push:** quest'operazione non ha particolari differenze rispetto alla Push tradizionale. L'unica accortezza da tenere è nel momento in cui un nuovo evento viene accodato: è necessario mantenere consistente lo stato degli iteratori incrementando di una unità i valori di entrambi i Tail così come quelli di entrambi i Length in modo da farli puntare all'ultimo evento inserito ed incrementare entrambe le distanze tra i corrispettivi Head e Tail.

```
1       entry Protected_Push (E_Ptr: in Basic_Event.Event_Ptr)
           when Length1 < Queue_Size and Length2 < Queue_Size is
3       begin
             Data(Tail) := E_Ptr;
5            Tail := Tail mod Queue_Size + 1;
             Tail1 := Tail1 mod Queue_Size + 1;
7            Tail2 := Tail2 mod Queue_Size + 1;
             Length1 := Length1 + 1;
9            Length2 := Length2 + 1;
        end Protected_Push;
```

- **Protected_Pop:** l'operazione di Protected_Pop funziona in maniera similare ma discrimina sulla base di quale istanza di display lancia la procedura: un valore intero identifica l'attore e lancia la sua specifica sequenza di comandi. Questa si occupa di estrarre e restituire il tipo access dell'evento, incrementare il valore di Head corretto e ridurre di una unità il valore di Length appropriato per mantenere ancora consistenti gli iteratori della coda. La situazione per l'altro display non è soggetta a variazioni e il numero di eventi che rimangono da estrarre a quest'ultimo non varia.

```
        entry Protected_Pop (Id: in Integer; E_Ptr: out Basic_Event.Event_Ptr)
2          when Length1 > 0 or Length2 > 0 is
        begin
4          case Id is
              when 1 =>
6                E_Ptr := Data(Head1);
                 Put_Line ("E_Que: event popped out from queue.");
8                Head1 := Head1 mod Queue_Size + 1;
                 Length1 := Length1 - 1;
10            when 2 =>
                 E_Ptr := Data(Head2);
12               Put_Line ("E_Que: event popped out from queue.");
                 Head2 := Head2 mod Queue_Size + 1;
14               Length2 := Length2 - 1;
              when others =>
16               Put_Line ("E_Que: error in popping from queue.");
           end case;
18      end Protected_Pop;
```

Nella figura sottostante possiamo vedere la coda degli eventi con i duplici puntatori Head, Tail e Length. Nell'esempio pratico:

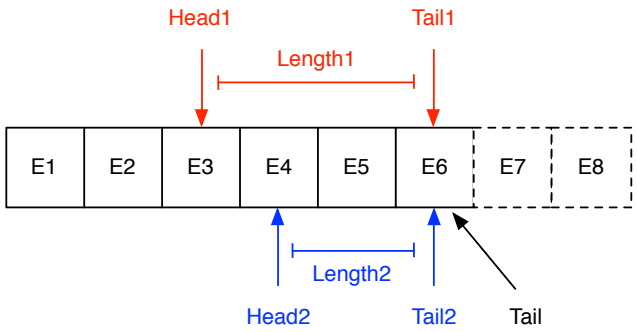

Figura 37: Struttura della coda.

1. La situazione nella coda è la seguente: gli eventi presenti sono E1, E2, E3, E4, E5 ed E6, mentre gli eventi E7 ed E8 non sono ancora stati generati;

2. Display1 ha ricevuto gli eventi E1 ed E2, deve ricevere E3 (iteratore Head1), E4, E5 ed E6 (iteratore Tail1) per un totale di 4 elementi (pari al valore di Length1 = 4);

3. Display2 invece ha ricevuto E1, E2 ed E3, deve ricevere E4 (iteratore Head2), E5 ed E6 (iteratore Tail2) per un totale di 3 elementi (valore di Length2 = 3);

6.5 Richiesta delle Statistiche di Gioco

Chiudiamo l'analisi del sistema distribuito con l'ultimo meccanismo disponibile: la richiesta di statistiche da parte del client al sistema di gioco. A differenza di tutte le trasmissioni viste finora in cui il server inoltra in modo attivo i dati al client (attraverso l'uso della coda) e quest'ultimo riceve e gestisce quanto ricevuto in modo passivo, questa connessione permette al display di interagire direttamente con il server effettuando una richiesta in modo attivo. La comunicazione tra server e client non è più monodirezionale ma diventa bidirezionale permettendo la trasmissione delle informazioni in entrambi i versi.

In base alle necessità (che nel modello è configurato come una richiesta ogni 5 eventi ricevuti) viene richiamata una procedura definita attraverso la direttiva *"pragma Remote Procedure Call"* direttamente sulla classe Stats contenente le statistiche e residente sulla partizione di gioco. Questa risponde a sua volta richiamando un metodo RCI del package Printer che si occupa della stampa sul display delle informazioni statistiche ricevute.

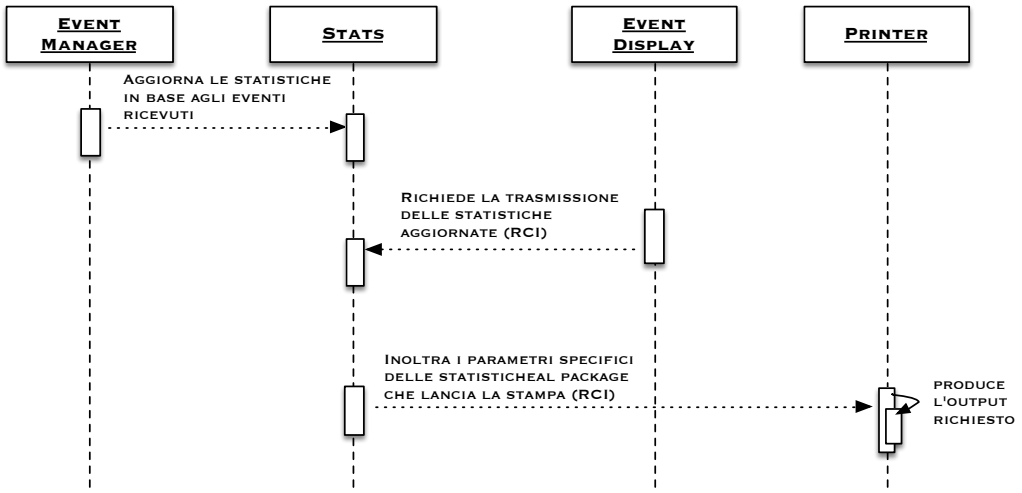

Figura 38: Sequenza della richiesta ed invio delle statistiche.

7 Integrazione tra le Parti - Concorrenza e Distribuzione

Ora che abbiamo affrontato e risolto in maniera scomposta le questioni relative ad un sistema concorrente così come quelle distribuite, siamo pronti per progettare l'integrazione dei sistemi in un unico sistema che sia al contempo concorrente e distribuito. Per fare ciò dobbiamo identificare alcune parti fondamentali, introdurre alcune modifiche del progetto originale e inserire eventuali nuove classi che permettano l'integrazione delle parti.

7.1 Integrazione del sistema

Analizzando strettamente il sistema concorrente ci si rende conto che le strutture di base che permettono al display di ottenere i dati necessari per la ricostruzione del gioco sono le seguenti:

- **Vettore delle Posizoni:** è l'oggetto protetto della classe Field e rappresenta la posizione di tutti giocatori in campo oltre alla posizione della palla nell'istante t;
- **ID Owner della Palla:** rappresenta l'ID del giocatore che possiede la palla nell'istante di tempo t. In caso l'ID sia uguale a zero significa che nessun giocatore è attualmente in possesso della palla (il task TaskBall ne prende il controllo e può muovere la palla secondo le leggi del gioco che le sono state applicate: tiro, passaggio, palla ferma, etc...);
- **Istante di Tempo *t*:** è la rappresentazione digitale dell'avanzare del tempo. Poichè gli eventi in gioco sono legati al momento in cui accadono, è necessario trasmettere al display l'avanzamento temporale;
- **Risultato Parziale:** è la successione in cui vengono segnate le reti dai giocatori;
- **Sequenza delle Azioni:** rappresenta la sequenza lineare degli eventi sommari avvenuti durante l'avanzamento di gioco;

Ognuno degli aspetti sopra elencati deve necessariamente essere analizzato nel dettaglio per poter definire il miglior metodo di implementazione nella sessione concorrente e distribuita.

7.2 Ridefinizione delle Parti

Per procedere è necessario comprendere ad alto livello come avvenga l'integrazione tra le due parti. È fondamentale scomporre la parte concorrenti in due parti indipendenti:

- **Game:** è composta da tutti i task dei giocatori e dalle logiche di gioco che ne permettono lo svolgimento. Oltre a questo contiene il task dell'arbitro, il vettore delle posizioni in campo e il timer di gioco;
- **Display:** è rappresentata dalla parte grafica che mostra a video i giocatori in campo oltre al risultato e al tempo di gioco;

Tra queste due parti è necessario interporre la struttura distribuita che abbiamo analizzato nel capitolo precedente per permettere la comunicazione dei messaggi tra gioco e display che verranno redistribuite su macchine differenti. In parole povere, la struttura di gioco (derivata dalla parte concorrente) si occuperà di comunicare gli avvenimenti durante la partita alla classe Event_Manager (derivata invece dalla parte distributa). Questa avrà il compito di impacchettare l'avvenimento in un evento specifico e trasmetterlo alla coda. Da qui il task Event_Display (anch'esso derivato dalla parte distribuita) recupererà l'evento depositato e ne trasmetterà il contenuto informativo a Display (e qui torniamo alla struttura implementata per il sistema concorrente) che si occuperà di presentarlo a video.

Nell'immagine sottostante troviamo una rappresentazione sommaria del processo di integrazione tra la parte concorrente quella distribuita, come descritto nel paragrafo sopra.

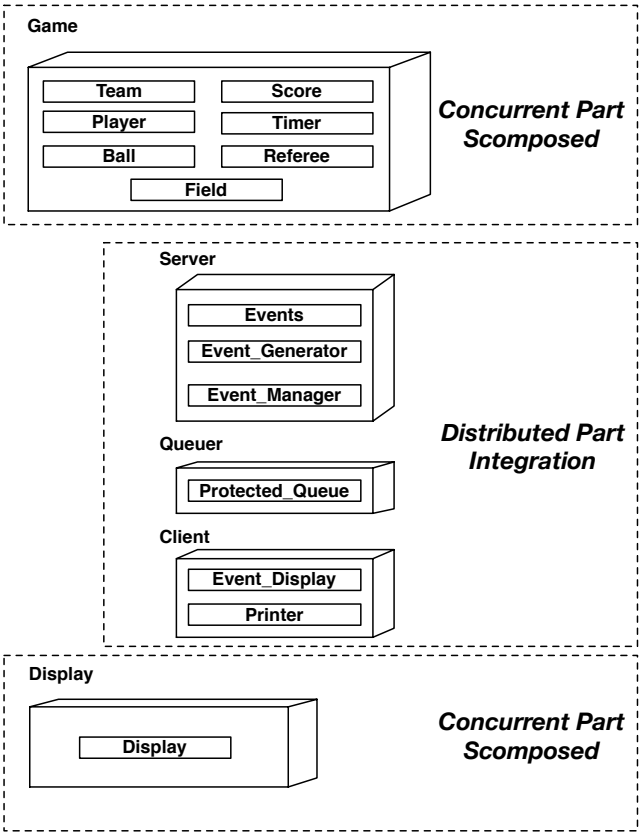

Figura 39: Scomposizione ed integrazione di base del sistema.

7.3 Strutturazione degli Eventi

Per cominciare dobbiamo ridefinire gli eventi per determinare delle tipologie significative per il nostro modello entrando pi nel dettaglio sul contenuto dei pacchetti e sulla loro finalità ai fini dello svolgimento del programma. Ogni evento serve a ricostruire una delle fasi di gioco e per questo si rende necessario identificarle per poterle gertire:

- **Azione Specifica:** è l'azione compiuta da uno specifico giocatore dal quale scaturisce l'avanzamento di gioco dello stesso e degli altri giocatori in campo;
- **Risultato Parziale:** rappresenta il risultato della partita rispetto alle squadre in campo;
- **Movimento in Campo:** non è altro che il movimento di ogni singolo giocatore (a cui andiamo ad aggiungere il movimento della palla identificata da un marcatore apposito) in una direzione e la condizione che sia o meno il possessore di palla;

Ora che abbiamo scomposto le attività in campo in tre macroblocchi, possiamo ricostruire gli eventi appositi legati all'andamento di gioco rispetto alla trasmissione che deve essere effettuata nei confronti del client che dovrà prendere in capo l'evento e gestirlo al meglio per replicare lato cliente quanto sta accadendo lato server. Nello specifico possiamo raggrupare gli eventi in tre tipologie che andiamo a dettagliare:

- **Summary Event:** conterrà le informazioni di sommario relative al tipo di evento accaduto:

1. l'*ID* del giocatore di cui si vuole trasmettere l'accaduto;
2. un tipo enumerate *Action* che identifica la tipologia di evento accaduto tra cui:
   (a) Tiro;
   (b) Passaggio;
   (c) Fallo;
   (d) Fuorigioco;
   (e) Rigore;
   (f) Goal;
   (g) Fuori (palla fuori dal gioco);
   (h) Sostituzione;
   (i) Infortunio;

- **Partial Event:** verranno inseriti i risultati parziali di gioco attraverso 3 variabili:
  1. l'intero *Score1* che identifica il risultato della squadra numero 1;
  2. l'intero *Score2* che identifica il risultato della squadra avversaria;
- **Complete Event:** avrà come contenuto informativo 4 informazioni principali:
  1. l'*ID* del giocatore di cui si vuole trasmettere la posizione;
  2. l'intero *Row* riferito alla riga sul quale si è mosso il giocatore;
  3. l'intero *Col* che rappresenta la colonna sulla quale si è spostato;
  4. il booleano *Own* che identifica se il giocatore sia o meno il possessore di palla;

Attraverso la strutturazione degli eventi come descritto, è possibile ricostruire lato client l'andamento del gioco attraverso i semplici eventi, presentando una lista cronologicamente ordinata di ciò che effettivamente sta accadendo in campo, oppure attraverso la composizione di tutti gli eventi che vengono recepiti dalla GUI e che permettono di ricostruire sul monitor di gioco l'avanzamento della partita.

Per approfondire la gestione degli eventi si rende necessario analizzare come questi vengano generati lato server e come questi vengano poi presi in carico lato client.

7.4 Costruzione degli Eventi Lato Server

Considerando quanto analizzato finora e basandoci sulla struttura esistente è possibile implementare la generazione degli eventi con un meccanismo che non stravolge la struttura pre−esistente: è sufficiente introdurre lato server l'entità Event_Manager, un task passivo che rimane in attesa di ricevere comunicazioni attraverso un meccanismo di select, il quale si occuperà prendere in carico ogni azione compiuta all'interno del server di gioco attraverso una serie di entry, ognuna legata alla specifica tipologia di evento in atto, costruendo il pacchetto adatto a gestirlo ed inoltrandolo alla coda di eventi.

Alcune entità in campo devono essere in grado di comunicare direttamente con Event_Manager per poter lanciare l'entry adatta alla generazione del pacchetto evento. Nello specifico i componenti in grado di parlare direttamente con Event_Manager saranno:

- **Arbitro:** necessario per la comunicazione degli stati di gioco e delle azioni che avvengono in campo poichè è egli stesso a scandirle ed a segnalarle opportunamente. In caso di fallo, palla fuori dal campo, fuorigioco, goal, etc... è lo stesso arbitro che interrompe l'andamento della partita e ne segnala la continuazione. In questo caso, per qualunque azione, apre la comunicazione con Event_Manager e trasmette il giocatore coinvolto oltre al tipo di azione generata. Si occupa inoltre di comunicare eventuali variazioni di risultato non appena la palla entra in rete e viene rilevato un goal. In quest'ultimo caso l'entry non genera un pacchetto di tipo *Summary Event* ma un evento di tipo *Partial Event* contenente il nuovo risultato della partita;

- **Field_Controller:** un nuovo task generato appositamente si occupa di trasmettere ogni modifica del vettore posizioni contenuto nell'oggetto protetto *Field*. Field_Controller non ha premessi di scrittura sul vettore ma ha accesso alla risorsa in sola lettura. Questo impedisce possibili modifiche accidentali del vettore andando a compromettere l'andamento del gioco. Anch'esso, come l'arbitro, comunica con Event_Manager attraverso un'apposita entry, trasmettendo la tupla {ID, Row, Col, Own} che rappresenta la posizione del giocatore in questione e l'eventuale possesso di palla;

Ora che abbiamo risolto velocemente la generazione degli eventi lato server è necessario analizzare come gli eventi vengano presi in carico lato client per poter essere riprodotti dall'interfaccia grafica.

7.5 Ricostruzione degli Eventi Lato Client

Allo stato attuale dell'arte, il Display è in grado di mostrare le posizioni dei giocatori in campo e i loro movimenti attraverso la chiamata di alcuni metodi che ritornano le informazioni contenute all'interno del vettore delle posizioni protetto dall'oggetto Field. Accededendo alla lista delle posizioni è possibile ridisegnare, istante per istante[3] i vari giocatori in campo, la palla, il possessore del pallone ed eventualmente il risultato parziale fino a *t*.

Per limitare il più possibile il numero di interventi necessari alla modifica sarebbe sufficiente creare un nuovo task *Vector_Manager* lato client che riceva gli eventi di tipo *Complete Event* e ricostruisca il vettore originale in un oggetto protetto. Il nuovo vettore conterrebbe 23 elementi (22 per i giocatori ed 1 per la palla) che permetterebbero di ridisegnare il campo e le posizioni di giocatori e palloni sul campo senza snaturare il prodotto realizzato per la parte esclusivamente concorrente. Oltre al vettore sarebbe necessario implementare una variabile intera con range tra 1 e 22 a segnalare il giocatore che nell'istante t detiene il possesso di palla. In questo modo l'interfaccia continuerebbe a funzionare "as is" con un numero di modifiche non eccessivo. Il risultato sarebbe contenuto su un secondo vettore, anch'esso in un oggetto di tipo protetto e modificabile esclusivamente da *Vector_Manager*, contenente due variabili intere *Score1* e *Score2* alle quali l'interfaccia avrebbe accesso in sola lettura. Gli eventi di tipo *Summary Event* potrebbero semplicemente essere mostrati a video attraverso una textbox che ricostruirebbe l'andamento del gioco e chiarirebbe eventuali interruzioni o riposizionamenti dei giocatori in campo (per esempio a seguito di un goal o dell'uscita della palla dal campo).

7.5.1 Riprogettazione dell'Interfaccia Grafica

L'alternativa, molto piú laboriosa della precedente ma decisamente più efficace e scalabile comporterebbe la ristrutturazione dell'interfaccia grafica in una serie di layer distinti:

1. la struttura del campo ovvero il rettangolo verde di fondo, le linee bianche della metà campo, delle aree (2 grandi e 2 piccole), delle porte e del centrocampo. Questa struttura rimarrebbe fissa ed invariabile nel tempo indipendentemente dall'avanzamento o meno del gioco e dell'evento accaduto sul campo di gioco;

2. una serie di layer, uno per ogni giocatore, nel quale l'unico oggetto sarebbe il cerchio colorato rappresentante il giocatore stesso, blu o rosso dipendentemente dalla squadra di appartenenza, ed eventualmente il pallone colorato di nero per differenziarlo dai giocatori in campo;

Attraverso questo sistema la metodologia per la gestione degli eventi di tipo *Summary Event* e *Partial Event* rimarrebbe pressochè invariata. Verrebbe stravolta invece la strategia di ridisegnaento del campo: non sarebbe pi necessario ridisegnare in toto la struttura di gioco, ogni singolo giocatore e la palla per ogni singola modifica del vettore posizionale lato client, ma sarebbe possibile lavorare direttamente sul layer del singolo giocatore interessaro lanciando il comando "clear" e ridisegnando il cerchio del giocatore riposizionato rispetto a quanto indicato nelle coordinate di *Complete Event*. Nel contempo sarebbe possibile disegnare un cerchio nero intorno al giocatore in questione, convenzione che identifica il possesso di palla, in base al

[3] con instante intendiamo un intervallo di tempo t fissato

valore contenuto nella variabile *Own* presente nell'evento. Lavorare in questo senso comporta una rilevante modifica di quanto fatto finora rispetto all'interfaccia, anche se il risultato ottenuto sarebbe decisamente migliore e sfrutterebbe al meglio le risorse legate agli eventi oltre a dare un senso preciso ai contenuti degli eventi stessi.

7.6 Duplicità dei Client di Gioco ed Accesso alle Risorse

Il fatto di avere un sistema distribuito ci da la possibiltà di gestire una moltitudine di client, ognuno dei quali permette di visualizzare l'avanzamento del gioco in maniera indipendentemente dagli altri client collegati. Per implementare questa funzionalità è sufficiente lavorare sulla coda degli eventi gestendo, come già fatto nel sistema esclusivamente distribuito, una coda con n iteratori, uno per ognuno dei client collegati.

Il meccanismo è semplice ed efficace: ogni monitor ha un proprio task *Display* specifico che accede alla coda ed effettua la pop degli eventi che non ha ancora ricevuto. Per ognuna delle pop che richiama, il task *Display* modifica l'evento puntato facendo avanzare l'iteratore di una posizione. In questo modo ognuno dei client è in grado di modificare il monitor in maniera indipendente dagli altri monitor visualizzando l'avanzamento della partita in modo temporalmente asincrono rispetto agli altri client. Questo però non comporta la visione di un gioco differente rispetto agli altri monitor coinvolti, semplicemente comporta un eventuale anticipo/ritardo rispetto agli altri.

In caso di disconnessione temporanea, lo stato di gioco viene preservato dalla coda in quanto l'iteratore rimane fermo sulla propria posizione. A seguito di ricollegamento, il task *Display* effettua una serie di pop consecutive recuperando tutti gli eventi che non sono stati estratti durante il downtime.

Potenzialmente sarebbe possibile implementare, oltre ai 2 monitor di gioco principali, olteriori monitor di sola "visualizzazione" della partita intervenendo sul numero di iteratori, incrementandone il numero fino ad un valore ragionevole legato al numero di spettatori che vogliamo poter collegare al nostro sistema di gioco. Questi terminali non hanno la possibilità, a differenza dei 2 terminali principali, di modificare lo stato di gioco (come andremo a vedere nella prossima sezione), ma potrebbero comunque visualizzare il gioco e godere dello svolgimento della partita in corso.

7.7 Modifiche dello Stato di Gioco dal Client

Come in ogni gioco che si rispetti deve essere prevista una certa iteratività con i giocatori umani che si trovano a fare i manager del gioco durante la partita. È possibile, come fatto finora, considerare il gioco completamente automatizzato e non permettere interventi esterni da parte delle persone che giocano con il simulatore, demandando tutte le scelte strategiche all'intelligenza artificiale dei Manager che, in condizioni determinabili, effettuano le scelte tattiche opportune e le sostituzioni necessarie per il buon esito della partita per la loro squadra. L'alternativa è quella di permettere al giocatore umano di effettuare modifiche alle tattiche di gioco come la variazione del modulo o lo sbilanciamento dello schieramento di gioco in attacco piuttosto che in difesa oppure di sostituire giocatori che non ritiene idonei in favore di giocatori migliori.

Per permettere l'iterazione è necessario prevedere un meccanismo di comunicazione basato su eventi e pilotati da un'interfaccia grafica che metta il giocatore umano in condizione di effettuare modifiche. Per fare ciò è possibile implementare un semplice pannello grafico in cui siano presenti delle combobox con i moduli di gioco più comuni (4−4−2, 3−4−3, 3−5−2, etc...), eventualmente la strategia (offensiva, normale, difensiva) e una sezione per le sostituzioni in cui sia possibile flaggare un giocatore in campo e uno in panchina per la sostituzione.

Per implementare a livello di sistema questa modifica è necessario ripercorrere i passi fatti finora legati alla gestione degli eventi e costruire una struttura similare ma inversa in cui gli eventi viaggino dal client al server (viceversa, quanto fatto finora prevede che gli eventi si spostino dal server al client). È necessario perciò prevedere 3 tipologie di eventi fondamentali che andiamo ad esaminare nel dettaglio:

- **Modulo:** l'evento prevede la gestione del modulo di gioco, ovvero come siano disposti i giocatori in campo attraverso l'utilizzo di una tripletta di interi che rappresentano:
  1. *Defense:* numero di giocatori in difesa;
  2. *Center:* numero di giocatori al centrocampo;
  3. *Offense:* numero di giocatori in attacco;
- **Strategia:** in questo caso per la codifica dell'evento viene utilizzato un tipo enumerazione che rappresenta una serie di tipologie di strategie da applicare in campo:
  1. **Offensivo:** tutte le posizioni dei giocatori in campo vengono incrementate in valore assoluto di 4 unità in direzione della porta avversaria;
  2. **Moderatamente Offensivo:** come il precedente ma le posizioni dei giocatori sono incrementate in valore assoluto di 2 unità in direzione della porta avversaria;
  3. **Normale:** le posizioni dei giocatori non subiscono modifiche rispetto alle posizioni previste in campo dal sistema in modalità predefinita;
  4. **Moderatamente Difensivo:** anche in questo caso vengono modificate tutte le posizioni dei giocatori in campo, ma vengono arretrate di 2 unità in valore assoluto in direzione della propria porta;
  5. **Difensivo:** come il precedente, ma il valore di arretramento verso la propria porta è di 4 unità;
- **Sostituzione:** quest'ultimo evento prevede l'utilizzo di 2 interi che rappresentano i due giocatori coinvolti nella sostituzione:
  1. *Out:* il numero del giocatore che deve uscire dal campo;
  2. *In:* il numero del giocatore che deve entrare in campo in sostituzione del precedente;

- gestione dei messaggi attraverso PartitionID per l'identificazione di destinatari dei messaggi della coda
- modifica del gioco dal client al server attraverso sostituzioni e modifiche della tattica di gioco

8 Conclusioni

Il gioco proposto e dettagliato nei capitoli visti fino ad ora risponde a tutti i requisi presentati nella fase introduttiva e fornisce un buon supporto a concorrenza e distribuzione per effetto delle scelte progettuali messe in atto.

Il modello di gioco è ridotto alle entità base necessarie per il funzionamento del progetto. Tutti i processi sono formalmente corretti e adempiono ai loro compiti in modo semplice ed efficace. Le logiche di movimento e di gestione delle risorse che i giocatori applicano durante la partita sono anch'esse molto semplici ed estremamente funzionali. I manager dirigono il match favorendo strategie il più adatte alle situazioni in cui si può trovare lo scontro. L'arbitro, dal canto suo, viene richiamato da situazioni particolari e regola la partita attraverso scelte più o meno corrette, come in una partita reale.

Come per ogni progetto, sono possibili migliorie alla strutture e agli algoritmi applicati al fine di perfezionare gli aspetti più grossolani e meno curati o eventuali errori non rilevati, ma ad una prima analisi il sistema è ben progettato. Nonostante la sua semplicità, assolve a tutte le richieste. È alquanto probabile che un sistema più complesso e con scelte progettuali più estreme comporti un maggior rischio nel portare a situazioni non facilmente gestibili. In particolar modo si può incorrere in circostanze di forte concorrenza che portano inevitabilmente a situazioni di *race condition*. La semplicità del prodotto presentato permette di analizzare e risolvere facilmente tali situazioni attraverso soluzioni accademiche eleganti e logicamente corrette.

La concorrenza tra giocatori è risolta attraverso la scelta del linguaggio di programmazione, Ada 95, e all'implementazione di risorse protette che ne garantiscono mutua esclusione. Vengono risolte le più comuni situazioni che portano il sistema ad uno stato di blocco parziale o totale, permettendo la costanza nel flusso di gioco. Anche la comunicazione tra processi avviene in sicurezza, utilizzando costrutti propri del linguaggio scelto.

Non soltanto la concorrenza è stata oggetto di indagine. Anche la distribuzione del sistema ha comportato un lavoro dettagliato di analisi e ricerca delle soluzioni più adatte per permettere al sistema di funzionare anche se ridistribuito. La distribuzione, come la concorrenza, se non gestita in maniera appropriata rischia di portare a situazioni inconsistenti di gioco e alla corruzione della partita. Sono state prese in considerazione tutte le condizioni più comuni e sono state proposte e motivate soluzioni per risolvere i quesiti oggetto di discussione.

Per quanto riguarda la codifica del prodotto, sono state tagliate le parti meno significative della specifica al fine di produrre in tempi contenuti un prototipo funzionante del sistema. Il modello così realizzato prevede lo svolgimento regolare della partita a cui vengono applicate regole di gioco semplificate, ma che riproducono effettivamente una simulazione funzionante. La codifica di tutte le regole trattate avrebbe comportato un aggravio dei tempi di produzione e di debug troppo oneroso per il gruppo (che allo stato attuale delle cose è composto da un solo elemento). Sono stati mantenuti comunque gli aspetti principali dei processi, definiti ed esaminati nella relazione.

9 Bibliografia

Riferimenti bibliografici

[1] *Ada Reference Manual*, 2011.

[2] A. Burns, A. Wellings, *Concurrent and Real-time Programming in Ada*, Cambridge University Press, 3rd Edition, 2007.

[3] A. S. Tanenbaum, M. van Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2nd Edition, 2006.

[4] Ada Resource Association http://www.adaic.org/

[5] AdaPower.com - The Home of Ada, http://www.adapower.com/

[6] Ada Programming, *Wikibooks, open books for an open world*, http://en.wikibooks.org/