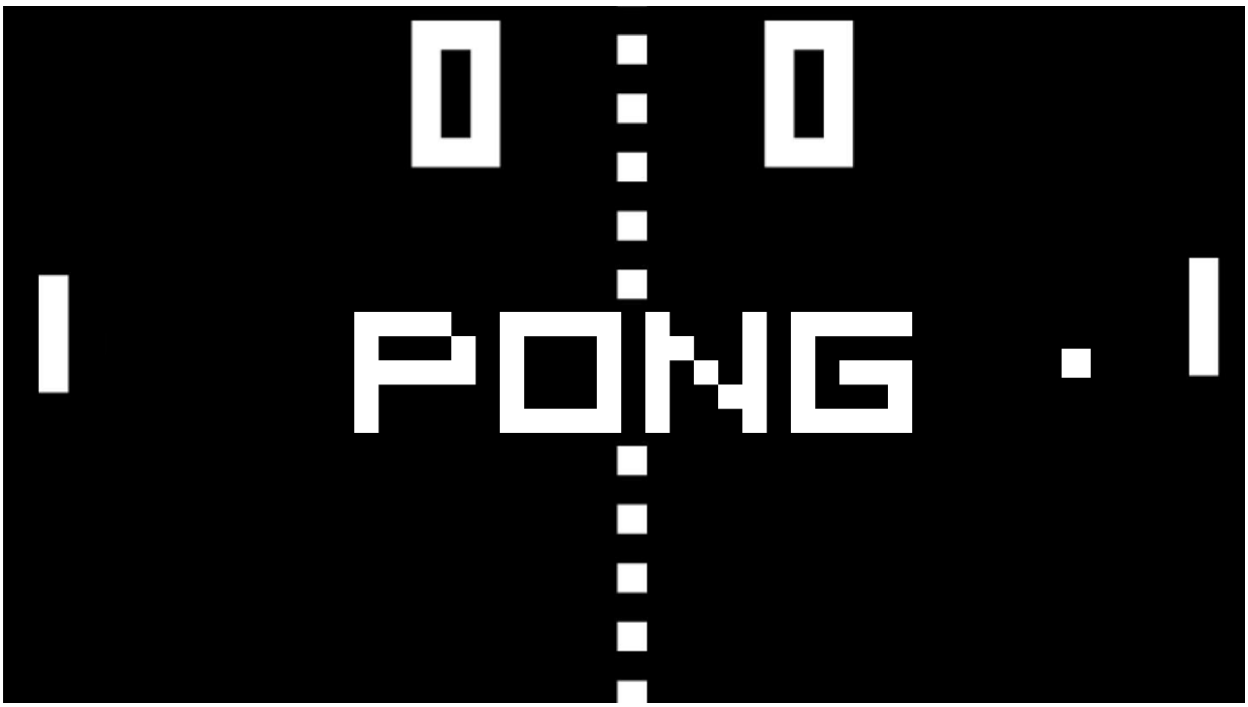
Progetto Sistemi Embedded

Prof.re Alessandro Ricci



Anno accademico 2015/2016

Marco Mancini
0000695219

Ivan Gorbenko
0000693772

Federico Marinelli
0000691530

INDICE

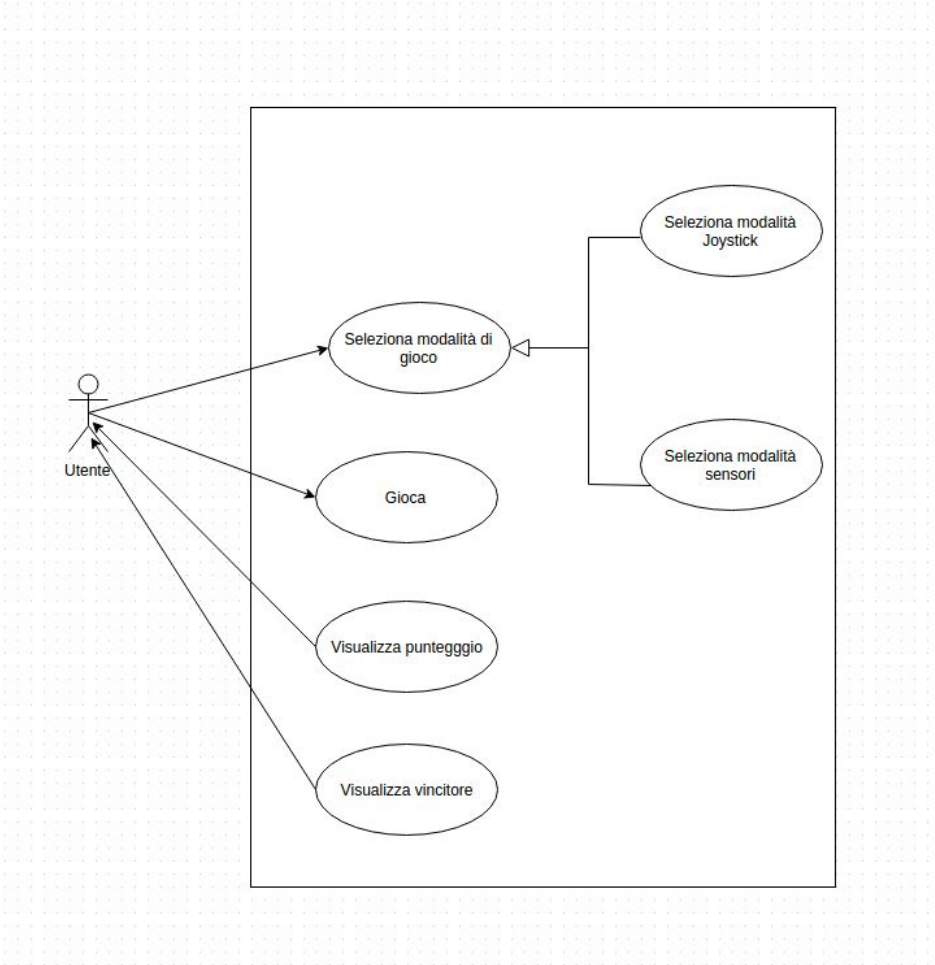
1.0 Descrizione del sistema

1.1 Descrizione in linguaggio naturale

L'obbiettivo di questo progetto è realizzare un sistema in grado di riprodurre il gioco del Pong tra due giocatori. Oltre alle normali funzionalità di gioco, sarà disponibile una modalità di interazione diversa dal solito. Gli utenti potranno interagire con il gioco, non solo utilizzando due joystick, ma anche attraverso il movimento dei propri arti.

1.2 Diagramma dei casi d'uso

Attraverso questo semplice diagramma dei casi d'uso si può capire meglio quali sono le funzionalità del sistema e come l'utente interagisce con il sistema.



L'utente potrà selezionare la modalità di gioco, la quale si distingue tra modalità joystick e modalità a sensori. Dopo aver selezionato la modalità di gioco potrà giocare contro l'altro utente avendo la possibilità, alla fine di ogni round, di visualizzare il punteggio corrente e, alla fine della partita, di visualizzare il vincitore.

2.0 Analisi generale

2.1 Hardware utilizzato

Per la realizzazione di questo progetto abbiamo deciso di utilizzare le seguenti componentistiche:

- 2x Arduino Uno
- 1x Matrice Led 16x32 RGB (https://www.adafruit.com/product/420)
- 2x Joystick (https://www.adafruit.com/products/512)
- 4x Sensori di prossimità

L'utilizzo di due Arduino Uno è dovuto al fatto che uno fungerà da controller del gioco, cioè sarà lui ad occuparsi dell'utilizzo dei joystick e dei sensori di prossimità mentre, il secondo Arduino conterrà la logica del gioco e la gestione della matrice su cui il gioco verrà visualizzato. Ovviamente i due Arduino Uno dovranno comunicare, sarà quindi necessario progettare un meccanismo di comunicazione tra i due.
Visto che il sistema è scomposto nei due sottosistemi sopracitati si procederà separatamente con l'analisi e la progettazione dei due, per poi effettuare un merge ed avere una visione completa ed esaustiva dell'intero sistema.

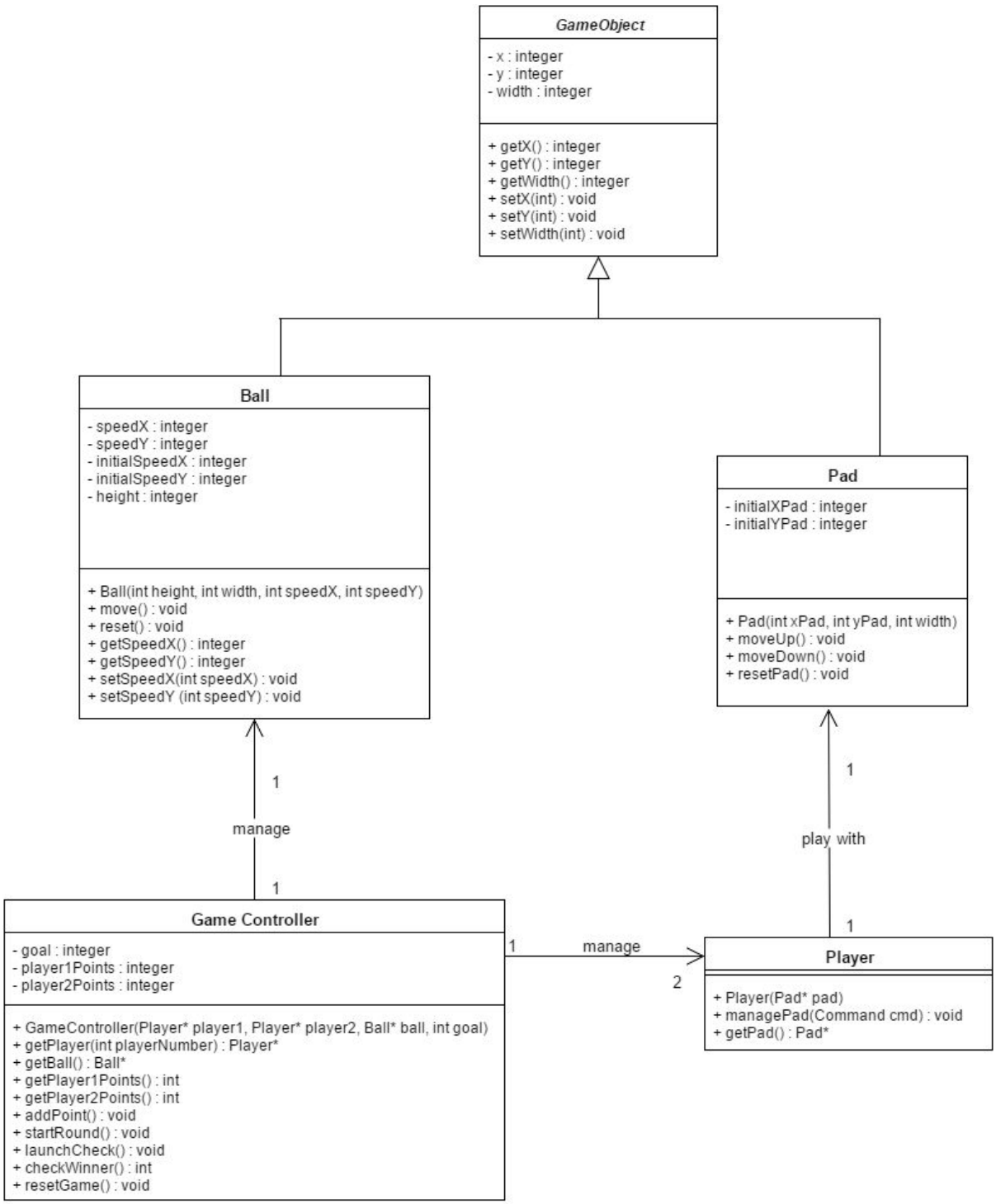
3.0 Analisi e Progettazione sottosistema logico

3.1 Analisi

Il primo sottosistema analizzato è il più corposo tra i due, infatti esso deve gestire la logica del gioco e la gestione della matrice. Visto che il sistema da progettare è un gioco, abbiamo deciso di massimizzare il distaccamento tra la logica di gioco ed il suo rendering grafico.

3.2 Analisi logica di gioco

Procediamo ora analizzando gli elementi che saranno presenti nel gioco e come sono interconnessi tra loro. Per comprendere al meglio questa fase è utile fare uso del seguente diagramma delle classi.

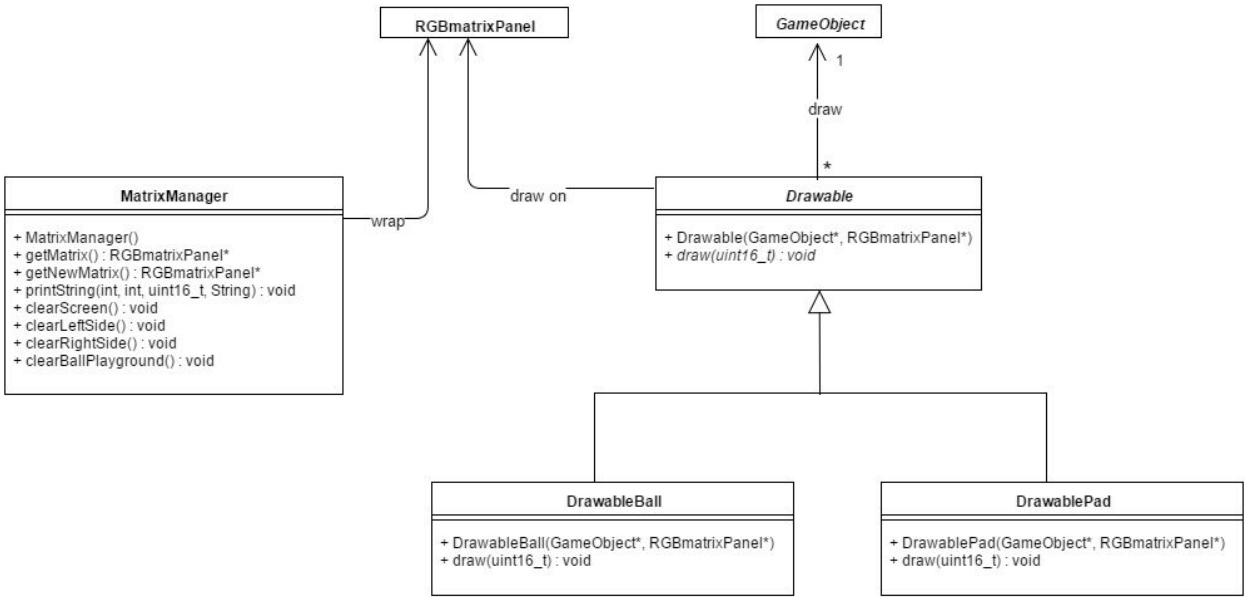


Dal diagramma si capisce che i principali elementi di gioco, Pad e Ball, estendono una classe astratta GameObject, quest'ultima è utilizzata per modellare un oggetto di gioco all'interno del sistema ed offre tre campi comuni comuni a Ball e Pad, cioè x, y e width con i diversi metodi getter e setter. In questo punto dell'analisi non ha un grande significato ma, nella fase di rendering dei diversi oggetti , sarà di grande aiuto. Abbiamo deciso di modellare sia il Pad che la Ball attraverso un unico punto

che, per i pads, indica il punto centrale del pad mentre per la palla, che può essere vista come un quadrato di punti, è il primo punto in alto a sinistra, inoltre, ha dei campi speedX e speedY i quali identificano il suo vettore movimento. Gli oggetti di Ball e Pads avranno la possibilità di muoversi nel campo di gioco, in particolare Ball offre un metodo move() a questo proposito, mentre Pad ha i metodi moveUp() e moveDown(). Ad ogni Pad è associato un Player il quale può eseguire comandi come moveUp, moveDown e reset sul proprio Pad. Infine c'è un game controller, questa classe può essere vista come il gestore del gioco infatti è lui a mantenere informazioni come il punteggio del gioco ed il numero di punti da effettuare per vincere la partita, inoltre ha la possibilità di aggiungere punti, lanciare controlli sulle collisioni della palla, avviare i diversi round, resettare il game e controllare la presenza o meno di un vincitore.

3.3 Analisi rendering oggetti in gioco

Come abbiamo precedentemente detto, abbiamo cercato di distaccare il più possibile la logica di gioco dal suo rendering. Dal seguente diagramma delle classi è possibile capire come è stata possibile un'implementazione del genere.



La classe astratta GameObject è la stessa del precedente schema mentre la classe RGBmatrixPanel rappresenta la matrice e contiene tutti i metodi necessari a pilotarla, ovviamente questa classe non è stata implementata da noi ma è stato possibile reperirla attraverso la pagina del prodotto. La classe astratta Drawable è utilizzata per modellare un qualsiasi oggetto all'interno della matrice e, oltre al

costruttore, offre un metodo astratto draw che, a seconda dell'implementazione (nel nostro caso DrawableBall e DrawablePad), definirà come l'oggetto sarà disegnato all'interno della matrice. Inoltre è stata aggiunta una classe MatrixManager, questa classe altro non è che un wrapper della classe RGBmatrixPanel, infatti offre solo i metodi strettamente necessari all'utilizzo del sistema (stampa di stringhe, pulizia della matrice, pulizia del playground, pulizia delle zone dei pad, reset della matrice).

3.4 Analisi dei tasks

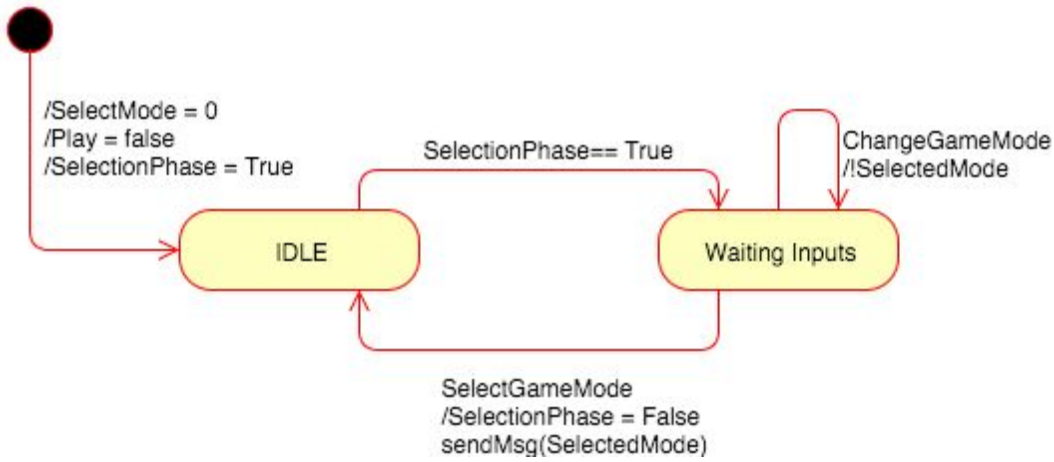
Questa porzione di sistema è suddiviso in diversi tasks modellati come macchine a stati sincrone con periodi diversi. In particolare abbiamo deciso di suddividire i tasks in due macrocategorie, tasks che agiscono sul modello e i task che eseguono rendering grafico sulla base del modello. Ovviamente, dovendo gestire diversi tasks, abbiamo dovuto utilizzare uno scheduler e diverse variabili condivise. I tasks sono i seguenti:

1. PadsTask: Si occupa della logica di movimento dei pad.
2. BallTask: Si occupa della logica di movimento della palla e delle fasi di roundover e gameover.
3. SelectGameModeTask: Si occupa della logica della fase iniziale di gioco, cioè la fase in cui si seleziona la modalità di gioco.
4. PrintBallTask: Si occupa del rendering della palla sulla matrice e della visualizzazione delle fasi di roundover e gameover.
5. PrintPadsTask: Si occupa del rendering dei pads sulla matrice.
6. PrintSelectionPhaseTask: Si occupa della visualizzazione della fase iniziale di gioco.

Prima di passare ad un'analisi dettagliata dei singoli task è meglio fare delle precisazioni. Avendo un'ampia suddivisione tra logica e rendering abbiamo deciso di dare ai tasks solo lo stretto necessario per svolgere i loro compiti. In particolare tutti i tasks che si occupano del modello hanno accesso alla message box e al gamecontroller, con la message box potranno inviare e ricevere messaggi mentre con il gamecontroller potranno agire sul modello. I tasks che si occupano del rendering, invece, necessiteranno solamente del matrixmanager e degli elementi drawable da disegnare.

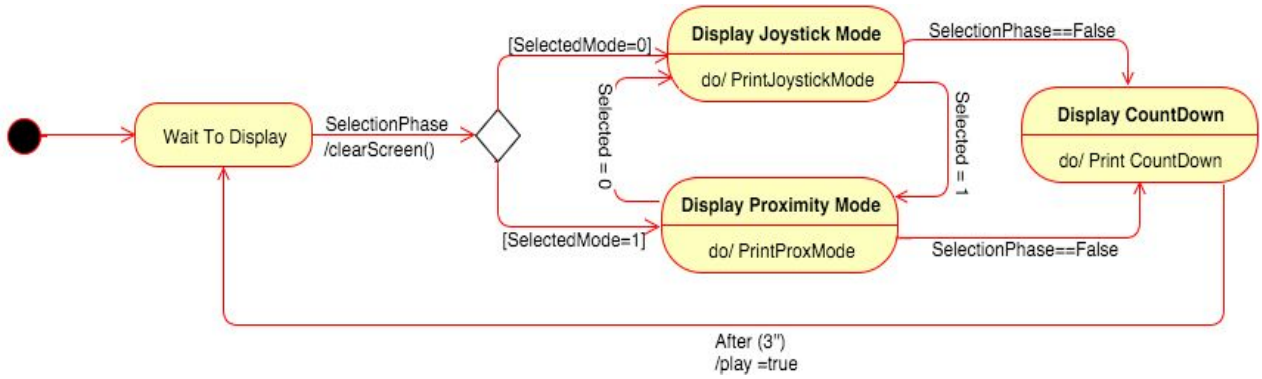
Passiamo ora all'analisi dei singoli task che verranno accompagnate dai relaitivi diagrammi di stato, per poi arrivare ad avere una visione completa del sottosistema.

3.4.1 Analisi SelectGameModetask



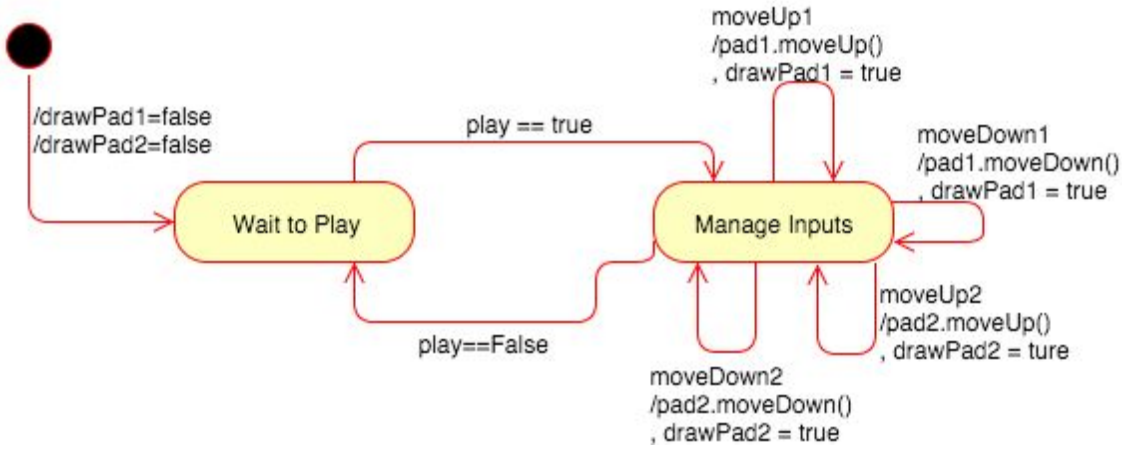
Questo task ha il compito di gestire la logica della fase iniziale di gioco. In questa fase l'utente potrà selezionare la modalità di gioco, che si distingue tra la modalità joystick e la modalità con i sensori di prossimità. All'inizializzazione del task vengono settate tre variabili condivise che sono selectMode, selectionPhase e play. La prima identifica la modalità di gioco selezionata (0 = joystick, 1 = proximity) ed è inizializzata a 0, la seconda è una variabile booleana inizializzata a true ed utilizzata per identificare la fase di selezione (true = in fase di selezione, false = fuori dalla fase di selezione) mentre play, anch'essa variabile booleana, identifica la fase di gioco (true = in gioco, false = fuori dalla fase di gioco) ed è inizializzata a false. Questo task, come si può capire dal diagramma, è caratterizzato da due stati. Nel primo stato, cioè quello di IDLE, non si fa altro che aspettare che la fase di selezione abbia inizio, appena questo succede si passa nello stato di WAITING_INPUTS. In questo secondo stato si aspettano messaggi (che arrivano dalla messageBox) e, se il messaggio è un messaggio di cambio modalità di gioco (changeGameMode) allora non si fa altro che invertire la modalità di gioco selezionata. Invece, se il messaggio è un messaggio di selezione modalità di gioco (SelectGameMode), allora si torna nello stato di IDLE settando fase di selezione di gioco terminata (selectionPhase = false) ed inviando un messaggio contenente la modalità selezionata attraverso la messageBox.

3.4.2 Analisi PrintSelectionPhaseTask



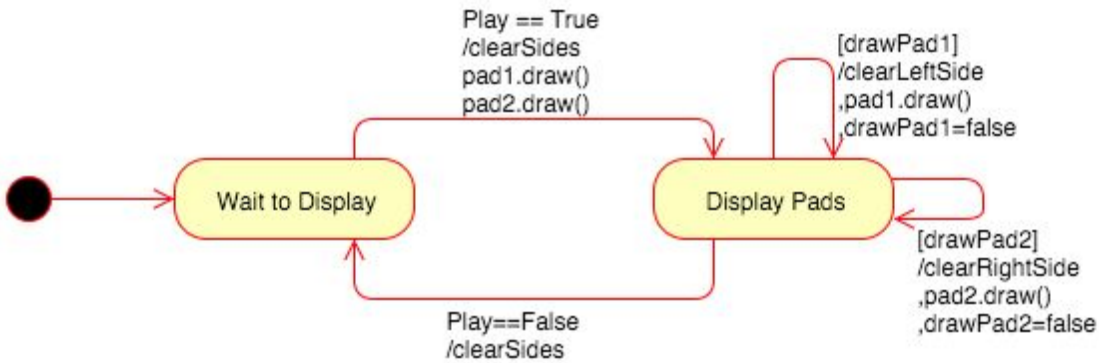
Il compito di questo task è quello di effettuare il rendering della fase iniziale di gioco. Esso condivide variabili sia con il task precedentemente descritto, sia con i task che si occupano dei pad e del gioco. Il task è suddiviso in quattro stati, WAIT_TO_DISPLAY, DISPLAY_JOYSTICK_MODE, DISPLAY_PROXIMITY_MODE e DISPLAY_COUNTDOWN. Nel primo stato, come nel primo stato del precedente task, non si fa altro che aspettare che la fase di selezione inizi (selectionPhase = true). Una volta che questa fase ha inizio la matrice viene pulita attraverso il metodo clearScreen() del matrixManager e, in base alla modalità di gioco selezionata (di default è joystick mode), c'è una transizione verso l'apposito stato. Entrambi gli stati DISPLAY_JOYSTICK_MODE e DISPLAY_PROXIMITY_MODE visualizzano la modalità selezionata sulla matrice e, a fronte di una variazione della variabile selectedMode, transitano nello stato apposito. Non appena la variabile selectionPhase torna a false, cioè quando l'utente seleziona una modalità di gioco, da entrambi i precedenti stati c'è una transizione verso l'ultimo stato, DISPLAY_COUNTDOWN. Questo stato non farà altro che visualizzare il countdown sulla matrice e, una volta terminato, transiterà nello stato iniziale, settando la variabile play a true e, quindi, avviando la fase di gioco.

3.4.3 Analisi PadsTask



Lo scopo di questo task è quello di occuparsi della logica del movimento dei pads. Quello che fa è leggere messaggi dalla message box, in base al messaggio, lanciare i comandi sui pad e comunicare al task di rendering dei pads che i pads devono essere disegnati. Con l'inizializzazione del task vengono settate le variabili booleane drawPad1 e drawPad2 a false, queste sono le variabili condivise con il task di rendering dei pads. Il primo stato in cui si trova il task è WAIT_TO_PLAY, in questo stato aspetta che la variabile play venga settata a true e, appena avviene questo evento, transita nello stato MANAGE_INPUTS. In questo stato aspetta l'arrivo di messaggi dalla message box, in base al messaggio ricevuto agisce sui pads e, in base al pad controllato, setta le variabili drawPad1 o drawPad2. Appena la variabile play viene settata a false il task torna nello stato iniziale.

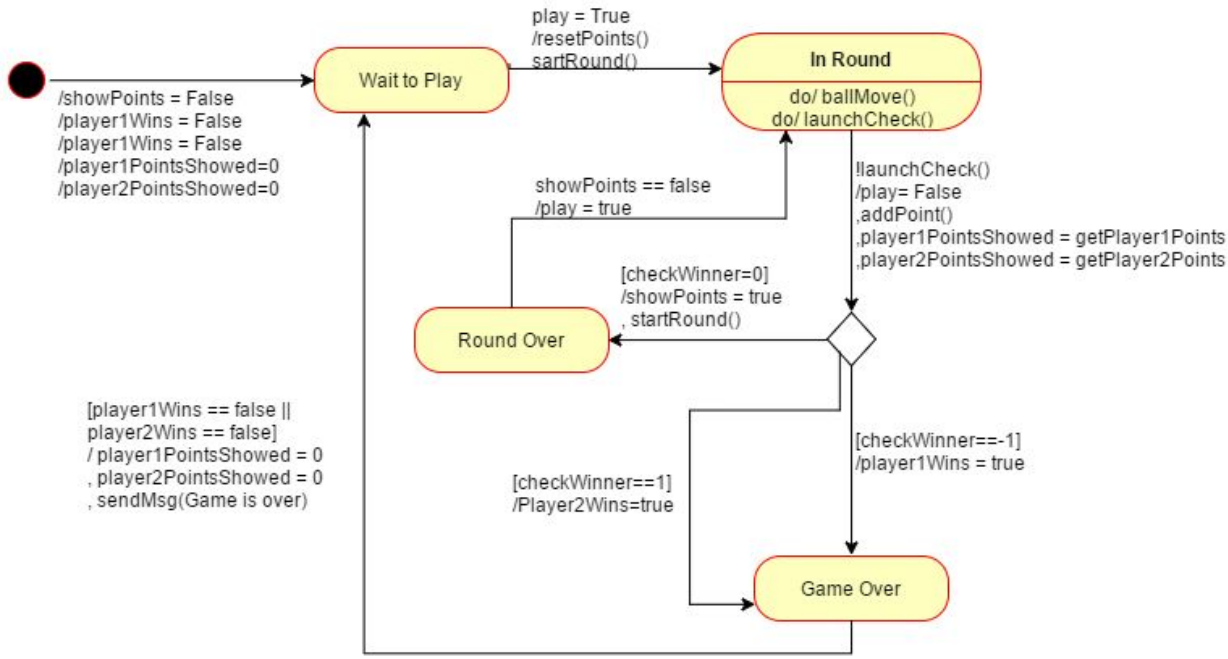
3.4.4 Analisi PrintPadsTask



Il compito di questo task è quello di visualizzare sulla matrice i due pad. Inizialmente si trova nello stato WAIT_TO_DISPLAY e, non appena la variabile condivisa play viene settata a true, effettua una transizione di stato verso

DISPLAY_PADS, con questa transizione vengono puliti i lati della matrice (attraverso il MatrixManager) che contengono i due pad che, subito dopo, vengono disegnati nelle corrette posizioni. Nello stato DISPLAY_PADS si aspetta che il PadsTask indichi a questo task che uno o entrambi i task vengano disegnati attraverso le variabili drawPad1 e drawPad2 che, alla fine del rendering del pad, vengono settate nuovamente a false. Quando si esce dalla fase di gioco (play = false), si torna nello stato WAIT_TO_DISPLAY pulendo nuovamente i lati della matrice che contengono i pads.

3.4.5 Analisi BallTask



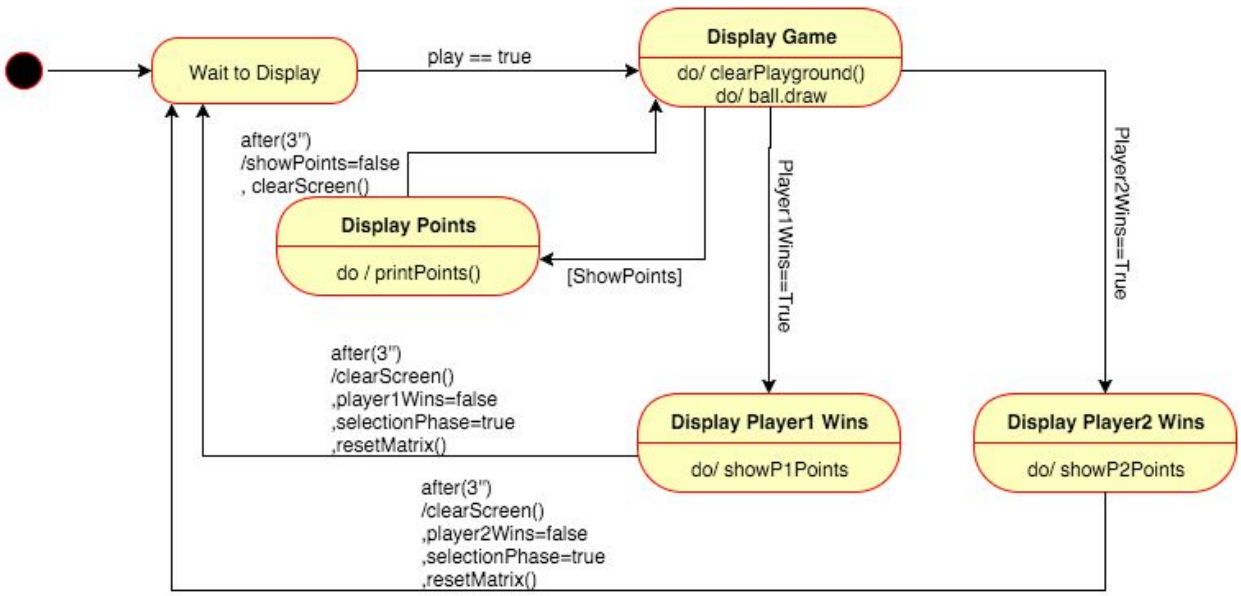
Questo task si occupa della logica di movimento della palla e delle fasi di roundover e gameover. Con il suo istanziamento vengono settate diverse variabili condivise:

- showPoints,player1Points, player2Points: vegonono utilizzate per la fase di roundover, dove dovranno essere mostrati i punteggi dei giocatori.
- player1Wins, player2Wins: vengono utilizzate per la fase di gameover, dove dovrà essere visualizzato il vincitore.

Lo stato iniziale del task è WAIT_TO_PLAY nel quale si aspetta che si entri nella fase di gioco. Quando si entra in questa fase avviene una transizione verso lo stato IN_ROUND e con essa si resettano i punteggi dei giocatori (metodo resetPoints() del gameController) e si riposizionano sia i pad che la palla nelle posizioni iniziali (metodo startRound() del gameController). Nello stato IN_ROUND non si fa altro che muovere la palla e controllare le collisioni (metodo move() della classe Ball e

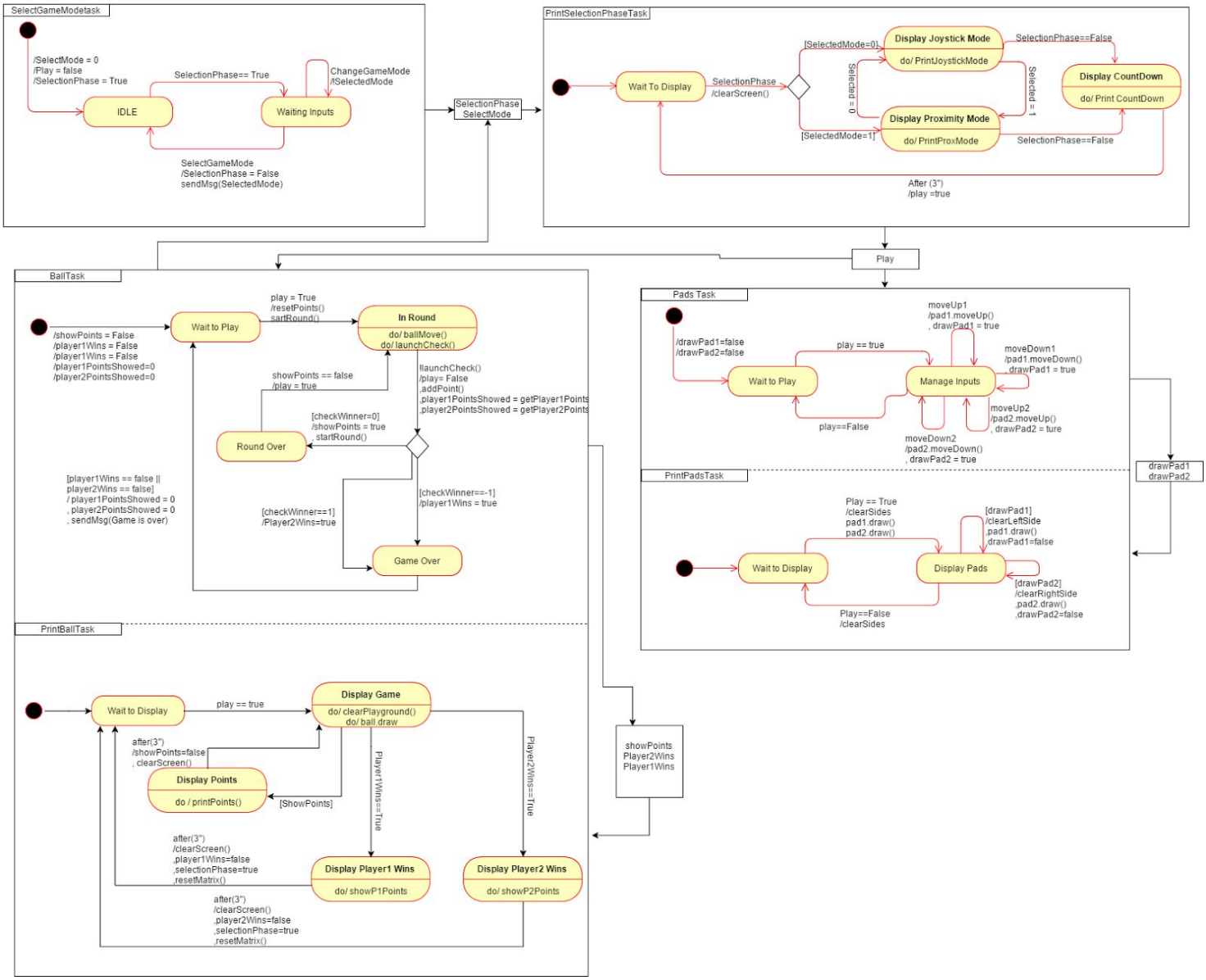
launchCheck() della classe gameController). Da questo stato si esce solamente se launchCheck() ritorna false, cioè se è avvenuto un roundover od un gameover. Appena si verifica questo evento la variabile play viene settata a false, viene aggiunto il punto al giocatore che ha vinto il round (metodo addPoint() del gameController), vengono assegnati i punti alle variabili condivise player1Points e player2Points e, a seconda del valore tornato dalla funzione checkWinner() (0 = round over, -1 = player 1 wins, 1 = player 2 wins) si transita di stato. Nel caso di un round over si transita nello stato ROUND_OVER resettando le posizioni della palla e dei pads mentre, in caso di game over si passa nello stato di GAME_OVER decretando il vincitore (attraverso le variabili player1Wins o player2Wins). Nello stato di ROUND_OVER si aspetta che la variabile showPoints torni a false, questa variabile sarà settata dallo stato del task che si occupa della visualizzazione dei punteggi sulla matrice, non appena questo accade si può tornare nello stato IN_ROUND settando la variabile condivisa play a true. Nello stato di GAME_OVER, invece, si aspetta che una delle variabili tra player1Wins e player2Wins torni a false, queste variabili sono settate a false dagli stati del task che si occupano di visualizzare il vincitore, appena una delle due variabili torna a false questo stato invierà, attraverso la message box, un messaggio di game over, setterà le variabili player1Wins, player2Wins, player1PointsShowed e player2PointsShowed al loro stato iniziale e si effettuerà una transizione verso lo stato WAIT_TO_PLAY.

3.4.6 Analisi PrintBallTask



Il compito di questo task è quello di effettuare il rendering della palla e delle fasi di roundover e gameover. Esso condivide variabili sia con BallTask che con i tasks che si occupano della fase iniziale di gioco. Lo stato iniziale in cui si trova è WAIT_TO_DISPLAY e rimarrà quì finchè non viene avviata la fase di gioco (play == true). Appena si entra nella fase di gioco il task passa nelo stato DISPLAY_GAME, questo stato ha il compito di visualizzare la palla sulla matrice. Appena BallTask si accorgerà che il round è finito setterà o showPoints o player1Wins o player2Wins a true. Nel primo caso si passerà nello stato di visualizzazione dei punteggi, DISPLAY_POINTS, da cui si uscirà dopo 3 secondi settando nuovamente showpoints a false e pulendo la matrice. Negli altri casi si finirà o nello stato DISPLAY_PLAYER1_WINS o DISPLAY_PLAYER2_WINS, che hanno entrambi il compito di visualizzare il vincitore. Una volta finita la visualizzazione, cioè dopo 3 secondi, il task tornerà nello stato WAIT_TO_DISPLAY settando nuovamente le variabili player1Wins o player2Wins a false, pulendo lo schermo, resettando la matrice e impostando a true la variabile selectionPhase, cioè facendo ricominciare la fase di selezione di gioco.

3.4.7 Diagramma di stato completo del sottosistema



Da questo schema si può vedere meglio come i diversi tasks interagiscono attraverso le variabili condivise e come il sottosistema funziona nel suo complesso.

3.5 Scheduling e tempistiche

Di seguito analizzeremo come lo scheduler è stato progettato e come le tempistiche dei diversi tasks sono state scelte.

3.5.1 Scheduling e relativi problemi

Tutti i tasks appena descritti vengono eseguiti da uno scheduler. Normalmente lo scheduler dovrebbe funzionare sulla base di interrupt generati dal Timer1 di Arduino ma durante l'implementazione ed il testing dei diversi tasks ci siamo accorti che le tempistiche non erano rispettate e avevamo un comportamento del sistema non deterministico. Analizzando la libreria della matrice abbiamo notato che anch'essa utilizza il Timer1 e genera interrupt con una frequenza molto alta resettando il contatore del timer. Secondo gli autori della libreria l'interrupt handler viene chiamato con una frequenza di circa 200Hz ma, eseguendo diversi test, ci siamo poi accorti che in realtà la frequenza con cui viene chiamato l'interrupt handler è dipendente dallo stato della matrice in quanto il metodo updateDisplay() chiamato ad ogni interrupt è ottimizzato in base allo stato di quest'ultima. Quindi abbiamo deciso di temporizzare lo scheduler sulla base delle tempistiche impostate dalla libreria della matrice. Per fare ciò abbiamo aggiunto due variabili condivise tra la matrice e lo scheduler che sono tFlag e overflows. Lo scheduler non utilizzerà più il metodo waitForNextTick() ma aspetterà che la variabile tFlag sia impostata a true prima di proseguire. Questa variabile viene settata a true ogni X interrupt lanciati dalla matrice, a tenere conto di questi interrupt è proprio la variabile overflows. Per fare in modo che le variabili siano visibili sia dalla libreria che dallo scheduler abbiamo deciso di dichiararle in due files (.h, .cpp) posizionati nella directory delle librerie di ArduinoIDE. Eseguendo diversi tests ci siamo accorti che con un numero di overflow pari a 200 abbiamo un periodo base di circa 25ms nella fase di gioco che è quindi accettabile.

Per quanto riguarda l'ordine dei task all'interno dello scheduler è stata seguita una logica ben precisa, prima vengono schedulati tutti i tasks che si occupano del modello (padTask, ballTask, selectGameModeTask) e a seguire tutti i task che si occupano del rendering (printBallTask, printPadsTask, printSelectionPhaseTask). In questo modo prima si aggiorna tutto il modello e dopo viene effettuato il suo rendering.

3.5.2 Tempistiche Tasks

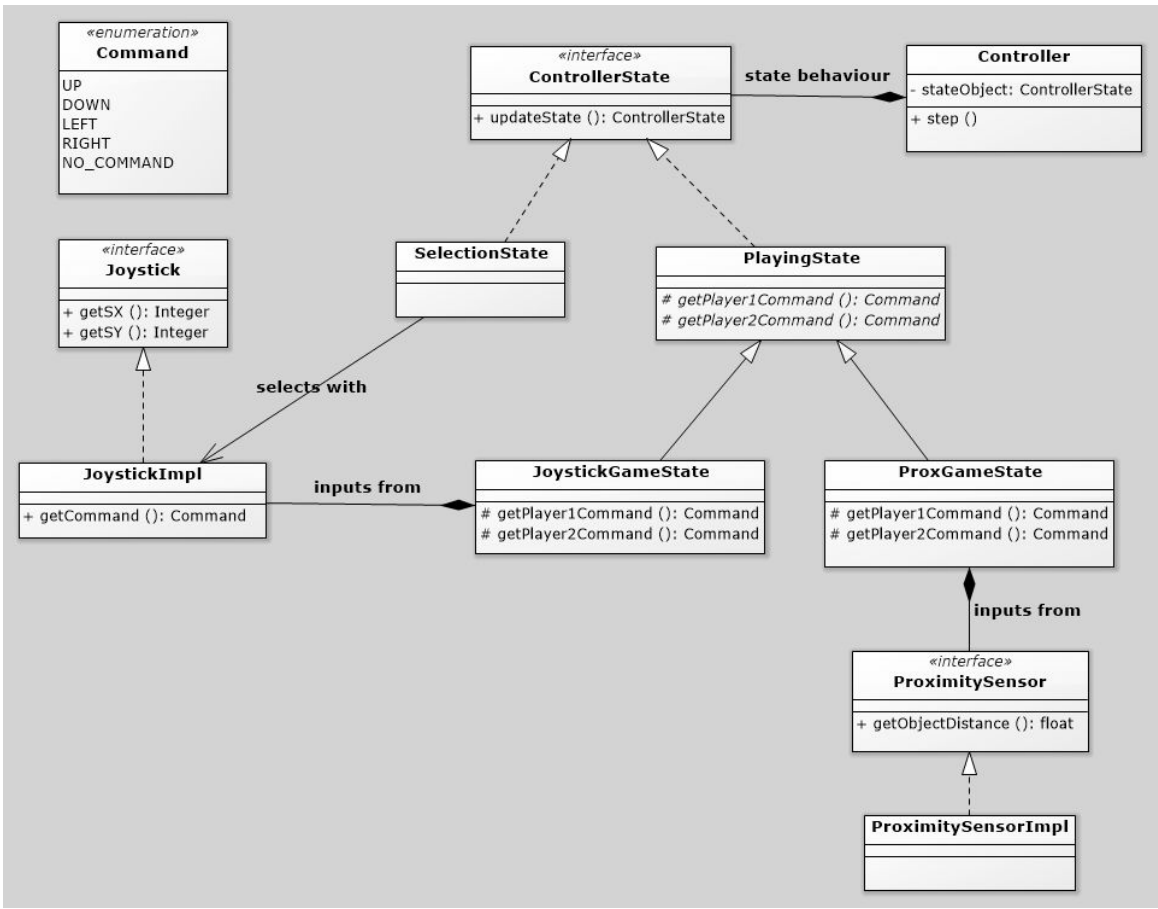
La scelta dei tempi dei tasks si basa soprattutto sulla fluidità di gioco, visto che i tasks che si occupano del movimento della palla e del suo rendering modellano questa fluidità sono loro a porre la base per tutte le tempistiche. Dando a questi due tasks un periodo di 25ms la palla percorre l'orizzontale dell'intera matrice in circa un secondo e si raggiunge una fluidità di gioco elevata. I task che invece si occupano della fase iniziale di gioco non hanno bisogno di un periodo così basso, quindi abbiamo deciso di dare a questi due tasks (selectGameModeTask e printSelectionPhaseTask) un periodo di 175ms, cioè equivalente al periodo presente nel sottosistema che funge da controller. Ai task che si occupano dei pad (PadsTask e PrintPadsTask) è stato assegnato un periodo di 75ms visto che devono essere responsive ma non quanto i task che si occupano della palla. Ovviamente tutte le tempistiche sono assegnate a coppie, visto che c'è un distaccamento tra modellazione e rendering.

4.0 Analisi e progettazione sottosistema di input

In questa parte si discute dell'analisi e progettazione dell'applicazione che sarà in esecuzione sul secondo Arduino che dovrà ricevere gli input dai joystick o dai sensori di prossimità e inviare i corrispondenti comandi all'altro Arduino il quale agirà di conseguenza sul modello del gioco

4.1 Analisi componenti

Ora analizziamo quali sono gli elementi necessari per riuscire a riprodurre il comportamento desiderato con un diagramma delle classi.



Necessariamente i primi elementi che devono essere modellati sono i dispositivi di input che sono stati rappresentati con le interfacce Joystick e ProximitySensor, che vengono rispettivamente implementate dalle classi JoystickImpl e ProximitySensorImpl. La classe JoystickImpl, inoltre, aggiunge il metodo getCommand() che attraverso chiamate a getSX() e getSY() restituisce un'istanza dell'enumerazione Command. Abbiamo scelto di utilizzare dei valori di soglia che il joystick deve superare per restituire uno specifico comando, perché la leva del joystick in posizione centrale restituisce 500 per i valori di x e y, e i valori di x e y sono compresi tra 0 e 1023 ma una volta spostata e lasciata tornare in posizione centrale non sempre ritorna esattamente a 500 perciò abbiamo scelto di utilizzare 300 come valore di soglia minimo e 700 come valore di soglia massimo per essere sicuri che lo spostamento della leva sia stato reale e non dovuto a un non perfetto ritorno in posizione iniziale. Per rappresentare l'elemento chiave dell'applicazione è stata utilizzata la classe Controller che concettualmente è l'unico task del sottosistema ed ha il solo metodo step() che fa avanzare lo stato del task. Inizialmente abbiamo scelto di modellare la macchina a stati con un semplice switch nel quale i singoli case rappresentati gli stati della macchina, ma successivamente

abbiamo notato che questa scelta risultava poco flessibile perché indipendentemente dalla modalità di gioco scelta, era necessario istanziare sia i joystick che i sensori di prossimità, ed inoltre la logica di ciò che dovevano fare gli stati "durante il gioco" era la stessa ma cambiavano solo i sensori da cui effettuare le letture e il modo in cui interpretarle per ricavare il corrispondente comando. Alla luce di questi accorgimenti abbiamo scelto di cambiare il design della macchina a stati per passare a un design OO e di utilizzare lo state pattern per modellarla visto che semplificava il progetto in quanto ogni stato deve solo conoscere il proprio comportamento e quali condizioni innescano la transizione di stato. Per rappresentare gli stati quindi è stato scelto di utilizzare l'interfaccia ControllerState che offre il metodo updateState() il quale aggiorna lo stato attuale e , in caso di una transizione, restituisce il nuovo stato, mentre se bisogna rimanere nello stato attuale restituisce NULL. Quindi come si può notare dal diagramma delle classi, la classe Controller ha come campo privato un'istanza dell'interfaccia ControllerState, alla quale delega l'aggiornamento del proprio stato nel metodo step().
Adottando lo state pattern è necessario fare una classe per ogni stato della macchina a stati, quindi le sottoclassi che sono state utilizzate sono:

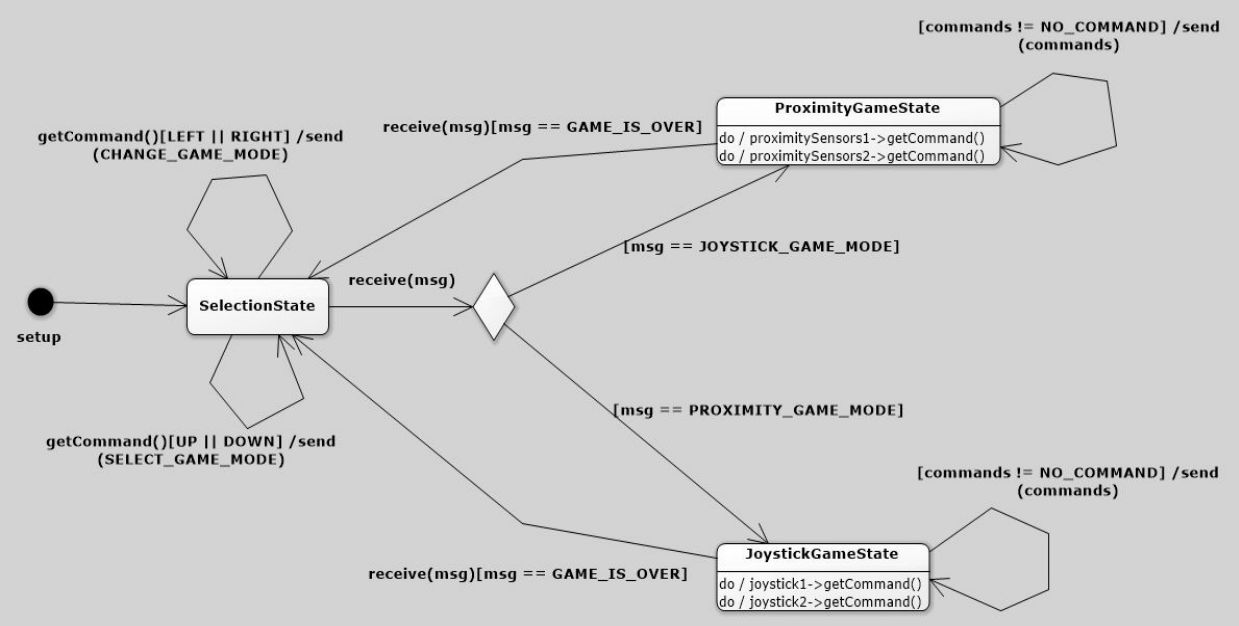
- SelectionState che ha il compito di far scegliere la modalità di gioco
- PlayingState invece incapsula la logica della lettura dei sensori e la comunicazione dei comandi all'altro Arduino durante il gioco
- ProxGameState e JoystickGameState sono sottoclassi di PlayingState

SelectionState di default utilizza il joystick del primo giocatore per far selezionare la modalità di gioco, e quando viene fatta la scelta restituisce un istanza di ProxGameState o JoystickGameState in base alla scelta. Per dare la possibilità in futuro di poter giocare con dei diversi dispositivi di input abbiamo deciso di utilizzare il template method per fare le letture dei sensori nel metodo updateState() di PlayingState, visto che varia solo il modo in cui devono essere interpretati gli input che dipendono dal particolare sensore, per fare ciò sono stati aggiunti due metodi protetti nella classe PlayingState per ricevere il comando dai sensori del primo e del secondo giocatore: getPlayer1Command() e getPlayer2Command(). Ora sarà sufficiente fare una sottoclasse di PlayingState per ogni tipo di dispositivo di input diverso e implementare i metodi astratti getPlayer1Command() e getPlayer2Command() che incapsuleranno la logica di interpretazione del comando sulla base dei sensori scelti. Nel caso di JoystickGameState è sufficiente richiamare i metodi getCommand() direttamente sulle istanze di JoystickImpl, mentre in

ProxGameState interpretiamo i comandi in base alla presenza o meno di un oggetto a una distanza minore o uguale a 10 cm per ogni sensore di prossimità.

4.2 Analisi task

La macchina a stati risultante dovrà permettere di selezionare la modalità di gioco ad uno dei giocatori e notificare la scelta all'altro Arduino, e una volta ricevuta la conferma della modalità di gioco da parte di quest'ultimo, dovrà permettere ai giocatori di ricevere input e inviarli al secondo Arduino per interagire con il gioco. Necessariamente il secondo Arduino dovrà notificare al primo quando una partita finisce per permettergli di transitare di stato e far nuovamente scegliere la modalità di gioco per una nuova partita. Tale comportamento è facilmente realizzabile con una singola macchina a stati sincrona, quindi analizziamo ora il relativo diagramma a stati



Dallo schema possiamo osservare che gli stati possibili sono tre. Il primo stato da cui la macchina entra in esecuzione è SelectionState, ovvero lo stato che permette di scegliere la modalità di gioco. E' stato scelto di modellare come auto-anelli gli eventi di input rappresentati dalle invocazioni al metodo getCommand(), che sostanzialmente legge lo stato del joystick del primo giocatore, il quale è stato scelto per effettuare la selezione della modalità di gioco. Come si può notare, ciò che accade allo spostamento della leva analogica verso destra o sinistra, è che viene cambiata la modalità di gioco e ciò è realizzato attraverso una notifica tramite

messaggio all'Arduino principale con il messaggio di CHANGE_GAME_MODE, che ha l'effetto di cambiare la modalità di gioco visualizzata sullo schermo. Una volta che il giocatore ha scelto quale modalità di gioco desidera utilizzare, dovrà spostare la leva del joystick verso l'alto o verso il basso per comunicare all'altro Arduino che la modalità di gioco che vuole scegliere è quella attualmente visualizzata sullo schermo, e anche questo comportamento è realizzato attraverso l'invio del messaggio SELECT_GAME_MODE. Per semplificare il design abbiamo scelto di delegare l'effettiva selezione della modalità di gioco all'Arduino che contiene la logica del gioco, e quindi l'Arduino Controller non fa altro che chiedere di cambiare la modalità di gioco in fase di selezione ed infine selezionare la modalità di gioco attualmente visualizzata sul display. Questa scelta però ha come conseguenza il fatto che l'Arduino Controller effettivamente non conosce quale modalità gioco sia stata selezionata, perciò abbiamo deciso che una volta che il giocatore conferma la modalità di gioco, e quindi viene inviato il messaggio di SELECT_GAME_MODE da parte del Controller, una volta fatto ciò il Controller attenderà che l'Arduino principale gli invii un messaggio di tipo JOYSTICK_GAME_MODE oppure PROXIMITY_GAME_MODE. La ricezione del messaggio con la modalità effettivamente selezionata, permetterà al Controller di transitare nell'opportuno stato di ProxGameState oppure in JoystickGameState in base al contenuto del messaggio, cosa che può essere osservata nel diagramma degli stati. Una volta effettuata la transizione nell'opportuno stato di gioco, il Controller non farà altro che leggere i comandi ricevuti dai dispositivi di input e inviare i messaggi corrispondenti ai comandi all'Arduino contenente la logica di gioco, il quale interpretando i messaggi ricevuti, agirà sul modello del gioco. Durante la fase di gioco l'Arduino principale dovrà gestire il momento in cui uno dei due giocatori perderà la partita e dovrà notificarlo all'Arduino Controller in modo da innescare la transizione di stato allo stato di SelectionState che dovrà nuovamente far scegliere la modalità di gioco per la prossima partita. Questa funzionalità è stata implementata negli stati ProxGameState e JoystickGameState attraverso il controllo di eventuali messaggi ricevuti dall'Arduino principale, il quale in caso di game over, invierà al Controller il messaggio GAME_IS_OVER.

4.3 Scheduling e tempistiche

Il Controller effettivamente è stato realizzato con un singolo task e quindi non è stato necessario l'utilizzo di uno scheduler e dell'astrazione del task realizzata dalla

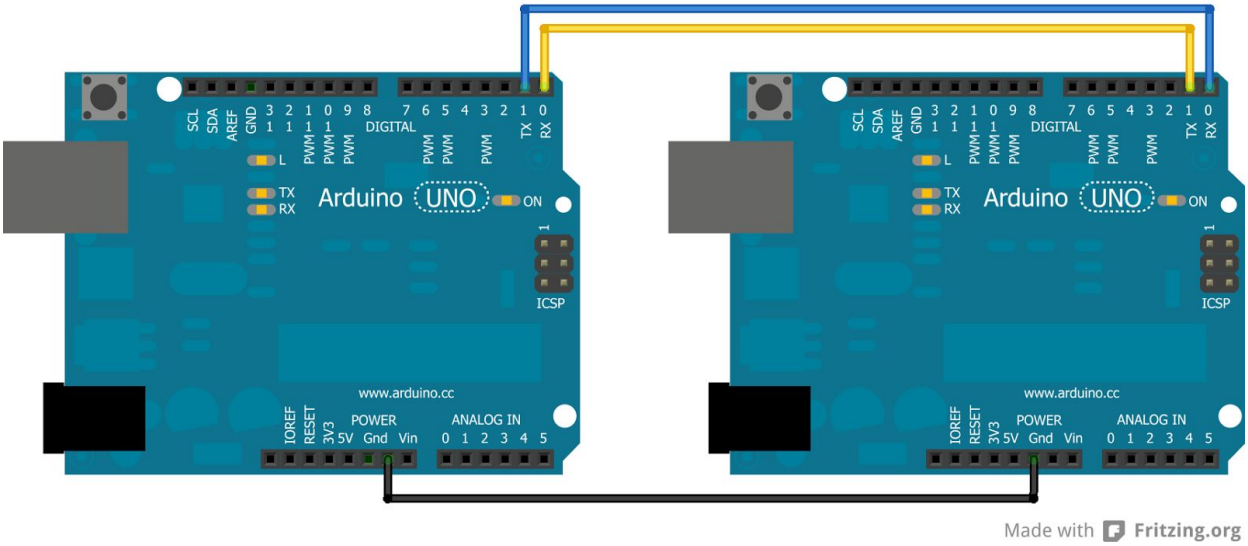
classe Task come nell'Arduino contenente la logica del gioco. Per temporizzare il task è stato sufficiente usare la classe Timer che fa uso del timer1 di Arduino e quindi il metodo step() del Controller viene eseguito a ogni tick del timer. Il periodo di tale task è stato scelto con la stessa metodologia dei task di logica del gioco, ovvero un compromesso tra la reattività della ricezione di input e la conseguente velocità di gestione dei messaggi da parte dell'Arduino principale. Con diversi test abbiamo notato che utilizzando tempi molto bassi c'è un eccessivo scambio di messaggi e quindi una reazione esagerata agli input che compromette la giocabilità e la fluidità del sistema in se. Quindi abbiamo deciso di assegnare al task un periodo di 175ms che risultano adeguati, sia per il particolare gioco che per il tasso di scambio di messaggi.

5.0 Interazione tra i due sottosistemi

Fino ad ora abbiamo trattato i due sottosistemi in maniera disgiunta ma, ovviamente, hanno un modo per comunicare ed interagire. Di seguito svolgeremo l'analisi di come questa comunicazione avviene, partendo dal cablaggio hardware per poi passare all'implementazione software.

5.1 Cablaggio hardware

Il cablaggio è molto semplice, infatti la comunicazione tra i due avviene mediante la seriale di Arduino cioè i pin Tx ed Rx, che devono essere collegati in modo sfalsato tra i due, come si può vedere nella seguente immagine:



5.2 Implementazione scambio di messaggi

Per quanto riguarda l'implementazione software entrambi i sottosistemi sono in possesso di una message box che usano per inviare e ricevere messaggi, per capire meglio la sua struttura guardiamo il relativo diagramma delle classi:

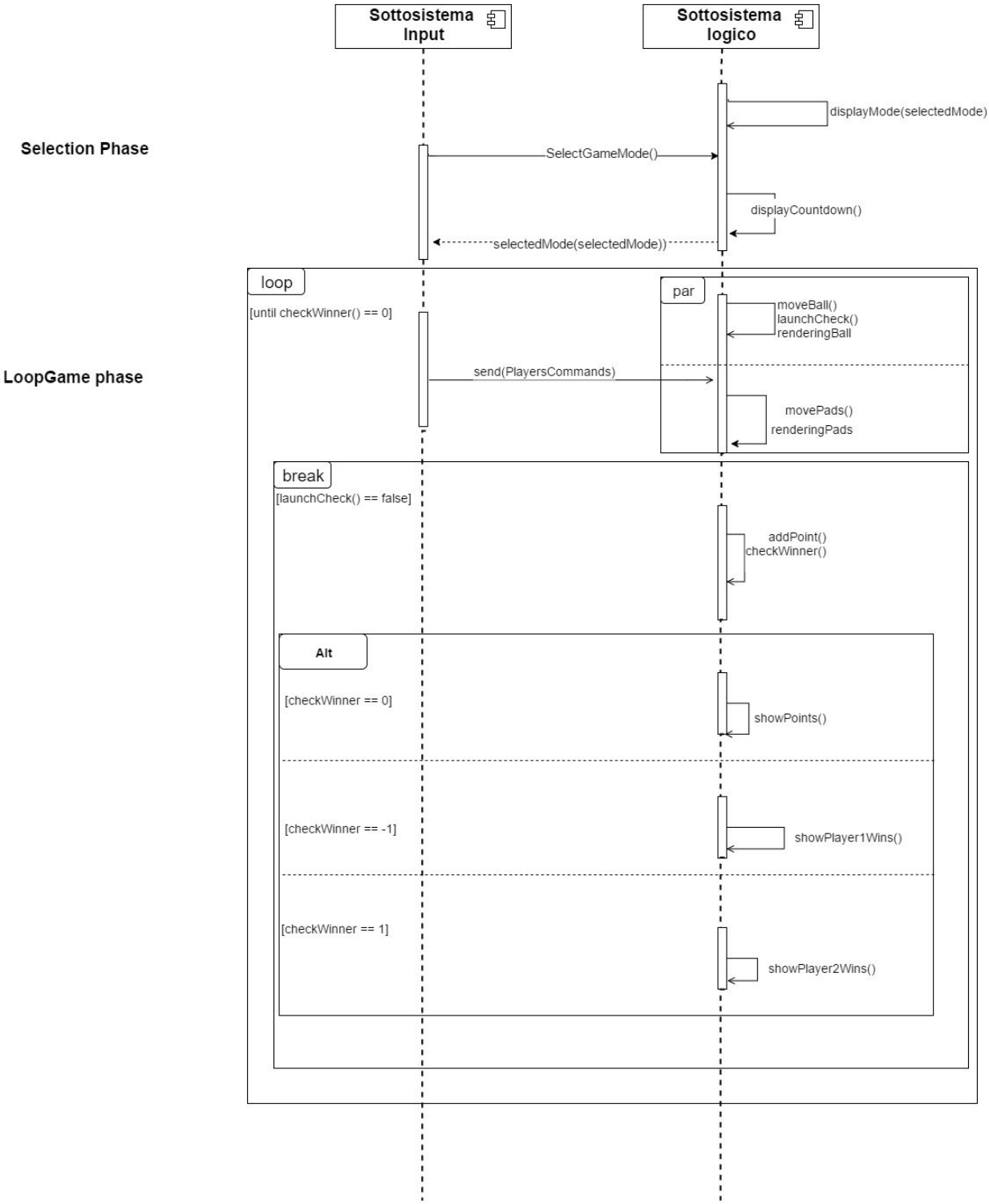
La classe Message modella un semplice messaggio, visto che c'è uno scambio di messaggi piuttosto elevato abbiamo deciso di modellare i messaggi come semplici char per mantenere il numero di bytes scambiati il più basso possibile e massimizzare l'efficenza, ovviamente abbiamo mappato tutti i messaggi per rendere più chiara la loro sintassi a livello di codice. Questa classe offre un costruttore con il quale si crea il messaggio ed un metodo getter che restituisce il contenuto del messaggio.

La classe MessageBox, invece, altro non è che un wrapper della seriale di Arduino, offre il metodo costruttore che si occupa dell'inizializzazione della seriale stessa, un metodo messageAvailable() per controllare la presenza o meno di messaggi sulla

seriale, un metodo getMessage() che ci permette di estrapolare un messaggio ricevuto, il metodo send(char) utilizzato per inviare un messaggio ed infine il metodo clearBox() che viene usato per pulire la seriale da messaggi che non vogliamo siano letti. Quest'ultimo metodo è stato inserito nella fase finale del progetto per risolvere un problema riscontrato durante la fase di round over, in questa fase vengono mostrati i punteggi dei giocatori e qualsiasi messaggio ricevuto deve essere scartato, quindi abbiamo deciso di aggiungere questo metodo che permette questa funzionalità.
Questa classe viene utilizzata in entrambi i sottosistemi, nel sottosistema di input è utilizzato dall'unico task presente mentre nel sottosistema logico sarà utilizzato solo dai task che si occupano del modello e che quindi devono comunicare, invece i task che devono occuparsi solamente del rendering non avranno accesso ad essa.

5.3 Esempio interazione

Analizziamo ora come i due sottosistemi interagiscono basandoci sullo scenario base in cui viene selezionata la prima modalità visualizzata sulla matrice, per aiutarci utilizziamo il seguente diagramma di sequenza:



Inizialmente il sottosistema logico visualizzerà la modalità di gioco, a seguito di una selezione della modalità da parte del sottosistema di input verrà visualizzato sulla matrice il countdown e, il sottosistema logico, invierà un messaggio al controller contenente la modalità di gioco selezionata il quale effettuerà una variazione di stato (precedentemente descritta) sulla base di essa. Successivamente entrambi i sistemi entreranno nella fase di gioco, in questa fase i giocatori potranno inviare comandi attraverso il sottosistema di input e il sistema logico reagirà a questi input muovendo i pads a seconda dei messaggi, parallelamente si occuperà anche del movimento della pallina e del suo rendering. Non appena uno dei due giocatori guadagna un punto il sottosistema logico controlla la presenza di un vincitore, se nessuno ha vinto (checkWinner == 0) visualizzerà i punteggi sulla matrice ed il round ricomincerà, altrimenti visualizzerà il vincitore e la partita finirà.

6.0 Conclusioni e possibili migliorie

Dopo aver effettuato una analisi completa del sistema embedded realizzato possiamo affermare che esso copre la maggior parte degli argomenti visti a lezione. Durante la realizzazione del progetto ci siamo imbattuti in problemi di vario genere come quello causato dal conflitto tra la libreria della matrice e la classe Timer utilizzata dallo scheduler ma che siamo comunque stati in grado di risolvere. Oltre a problemi di natura software abbiamo avuto problemi con hardware difettoso come i bottoni che, vista la reattività che il sistema richiede, risultavano inadatti e ci siamo dovuti mettere alla ricerca di dispositivi più opportuni come i joystick.
Una delle possibili migliorie a cui abbiamo pensato è la possibilità di aggiungere dell'intelligenza artificiale che permetterebbe di giocare con un unico giocatore. Inoltre visto il design scelto lato controller sarà facilmente possibile aggiungere nuovi dispositivi di input.