F1 Symulator

Progetto di Sistemi concorrenti e distribuiti

02 Luglio 2009

Sommario

Relazione sul progetto di Sistemi Concorrenti e Distribuiti.

Informazioni documento

Nome file	F1_Sym.pdf
Versione	0.3
Distribuzione	Prof. Vardanega Tullio
	Miotto Nicola
	Nesello Lorenzo

Indice

Elenco delle figure

1 Progetto

Il progetto riguarda l'analisi e la risoluzione delle problematiche di progettazione di un simulatore concorrente e distribuito di una competizione sportiva assimilabile a quelle automobilistiche di Formula 1. Il sistema da simulare dovrà prevedere:

- un circuito selezionabile in fase di configurazione, dotato della pista e della corsia di rifornimento, ciascuna della quali soggette a regole congruenti di accesso, condivisione, tempo di percorrenza, condizioni atmosferiche, ecc.
- un insieme configurabile di concorrenti, ciascuno con caratteristiche specifiche di prestazione, risorse, strategia di gara, ecc.
- un sistema di controllo capace di riportare costantemente, consistentemente e separatamente, lo stato della competizione, le migliori prestazioni (sul giro, per sezione di circuito) e anche la situazione di ciascun concorrente rispetto a specifici parametri tecnici
- una particolare competizione, con specifica configurabile della durata e controllo di terminazione dei concorrenti a fine gara.

2 Problematiche

2.1 Introduzione

In questo capitolo verranno analizzate le problematiche legate alla realizzazione di un sistema concorrente e distribuito.

2.2 Problematiche di concorrenza

Nel corso dell'analisi ad alto livello del progetto, sono emerse numerose problematiche legate alla coesistenza nel sistema di più *task* concorrenti e di risorse tra di essi condivise.

2.2.1 Percorrenza concorrente della pista

Come da specifica, il sistema dovrà prevedere una pista per lo svolgimento della gara. I partecipanti percorreranno quindi i tratti del circuito in modo concorrente e questo pone già numerose problematiche per il corretto svolgimento della competizione. Si può vedere ogni tratto come una risorsa condivisa fra i concorrenti della gara. Ciascuna risorsa avrà una motleplicità limitata, coerentemente

a quanto accade nella realtà. Questo porta inevitabilmente a due problematiche da affrontare:

1. I concorrenti dovranno attraversare ogni tratto in modo da non violare i limiti di molteplicità imposti. Bisognerà quindi impedire che ad un istante **t** dello svolgimento della gara, su un tratto con molteplicità **n** vi siano, contemporaneamente, **n+1** auto che lo stanno attraversando;

2. Evitare l'effetto "teletrasporto". Uno possibile scenario che si potrebbe infatti presentare nel corso della gara potrebbe essere il seguente (in caso di progettazione poco attenta): Un tratto Tr_N a molteplicità **1** viene attraversato da **2** auto, **A** e **B**. Logicamente, quindi, se **A** arriva prima di **B**, all'inizio del tratto Tr_{N+1} l'ordine dovrà essere mantenuto. Ad alto livello si potrebbe pensare, come soluzione plausibile, di affidare ad **A** la risorsa Tr_N e, una volta effettuato virtualmente l'attraversamento, farla rilasciare e affidarle il tratto Tr_{N+1}, mentre **B** starà cercando di ottenereTr_N e percorrerlo. Questa soluzione non tiene però in considerazione le problematiche legate alla gestione dei processi dallo scheduler. Potrebbe infatti accadere che:

   - **A** ottiene Tr_N e **B** rimane in attesa;
   - **A** rilascia Tr_N e viene prerilasciato dallo scheduler;
   - **B** ottiene Tr_N, lo rilascia e successivamente ottiene Tr_{N+1} prima di essere prerilasciato dallo scheduler;
   - **A** è di nuovo attivo ma si trova in una posizione non coerente con quanto previsto: è avvenuto un sorpasso in una zona non consentita.

   Da questo emerge il problema della gestione dei processi a livello di sistema operativo. Non è possibile infatti fare assunzione sulle politiche di ordinamento e esecuzione dello scheduler, poichè dipende dalle scelte architetturali del sistema operativo sottostante. Di conseguenza è necessario sviluppare un strategia di svolgimento della gara che preservi la coerenza della competizione indipendentemente dal comportamento, talvolta non prevedibile, dello scheduler.

3. Il problema precedente potrebbe essere raggirato introducendo l'accumulo di risorse. Ovvero prima di rilasciare il tratto Tr_N, il concorrente Tr_{N+1} deve aver già ottenuto l'accesso al tratto Tr_{N+1}. In questo modo, però, si potrebbe presentare una prospettiva di stallo. Infatti, se il numero di tratti è minore o uguale al numero di concorrenti, potrebbe verificarsi un'attesa circolare potenzialmente infinita sui tratti della pista. Bisogna quindi assicurarsi che non ci siano le condizioni per il verificarsi di stalli.

2.3 Problematiche di distribuzione

Un sistema che funzioni grazie alla comunicazione remota fra componenti dislocati in diversi nodi nella rete, presenta certamente delle specifiche problematiche da affrontare. La più critica riguarda di sicuro la robustezza. È difficile o addirittura utopico sperare nell'affidabilità della rete. La rete presenta sempre dei fault, la cosa importante è gestirli e non farli propagare in errori. Sicuramente quindi bisognerà minimizzare la probabilità che un nodo distribuito, perdendo il contatto con il resto del sistema, possa provocare un malfunzionamento globale.

3 Architettura ad alto livello

Nella seguente sezione verrà illustrata l'architettura ad alto livello del sistema sviluppato, escludendo i dettagli implementativi e legati al linguaggio. Viene fornito un diagramma delle componenti per mostrare molto ad alto livello l'architettura del sistema.

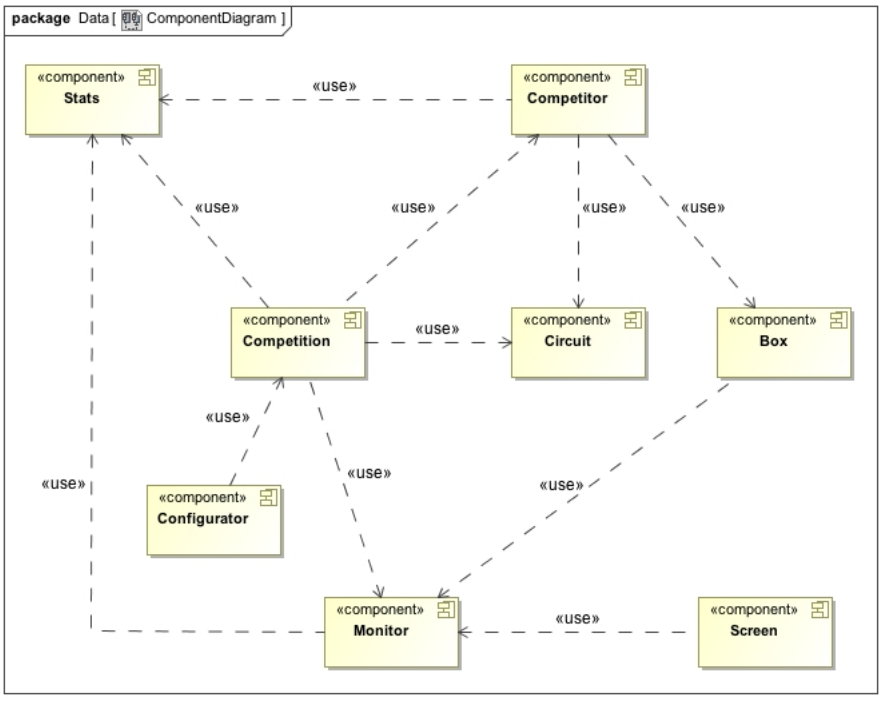

Figura 1: Diagramma delle componenti

3.1 Componenti di sistema

Le principali componenti del sistema sono:

3.1.1 Competition

La *Competition* è l'unità atta ad orchestrare l'avvio e la conclusione della corsa. Tale componente, dunque, è stata concepita per offrire le seguenti funzionalità:

- Configurazione parametri di gara:
  - numero di giri;
  - numero di concorrenti;
  - circuito;
- Gestione della sessione di iscrizione e accettazione concorrenti (configurati a livello della componente *Box*)
- Avvio delle componenti necessarie al monitoraggio della gara (quali ad esempio *Monitor*)
- Avvio controllato della competizione vera e propria nel momento in cui tutti i prerequisiti di inizio sono soddisfatti, ovvero:
  - la competizione è stata configurata correttamente;
  - le componenti di controllo e gestione della competizione sono attive e in attesa di comandi;
  - i concorrenti sono stati correttamente registrati alla competizione e in attesa di partire;

3.1.2 Competitor

Il *Competitor* è l'entità pensata ad svolgere la gara. È caratterizzato dalle seguenti sotto-componenti:

- **Auto**, ovvero tutte le caratteristiche fisiche legate all'auto, ovvero:
  - motore;
  - capacità del serbatoio;
  - massima accelerazione;
  - massima velocità;
  - gomme montate (mescola, modello, tipo);
  - livello usura gomme;

  - livello della benzina nel serbatoio;

- **Guidatore**, cioè le informazioni che descrivono più dettagliatamente il concorrente in gara:
  - nome e cognome pilota;
  - nome scuderia

- **Strategia**, ovvero la strategia che sta adottando il pilota, suggerita dai box e dinamica nel corso della gara:
  - stile di guida, variabile tra conservativo, normale, aggressivo, a seconda dello stato della macchina e delle previsioni fatte dai box
  - numero di lap prima del pit stop
  - addizionalmente, quando viene fatto un pit stop, la strategia determina anche quali siano le nuove gomme da montare, la quantità di benzina da avere nel serbatoio e il tempo impiegato per il pit stop.

Tutte queste informazioni insieme creano quello che viene definito il concorrente di gara. Tali informazioni verranno poi usate nel corso della gara per:

- scegliere al momento giusto la miglior traiettoria da seguire, in base alla presenza o meno di altri concorrenti nelle vicinanze e alla difficoltà del tratto;
- fornire costantemente aggiornamenti sul suo stato (tramite una parte del modulo *Stats*, informando il computer di bordo riguardo a:
  - livello di usura gomme;
  - livello di benzina;
  - checkpoint attraversato con tempo di arrivo;
  - insieme al checkpoint verranno aggiornate le informazioni relative a settore e lap;
  - velocità massima raggiunta.
- contattare ad ogni giro il box per ottenere una strategia aggiornata;
- se suggerito dai box, effettuare un pitstop;
- ritirarsi dalla gara una volta che le condizione dell'auto non permettano di poter correre ulteriormente;
- banalmente, continuare a correre fino alla fine delle lap prestabilite, dopodichè fermarsi.

3.1.3 Circuit

Il circuito è una risorsa finalizzata ad offrire il piano su cui svolgere la competizione. È condivisa fra tutti i concorrenti in gara e offre un insieme di funzionalità per poter conoscere le caratteristiche dei vari tratti della pista (compresi i concorrenti presenti al momento dell'attraversamento). È composto dalle seguenti sottocomponenti:

- **Checkpoint**: i *Checkpoint* rappresentano punti di arrivo intermedi del circuito. Come una suddivisione in fette da 1 secondo possono discretizzare 1 minuto, così i *Checkpoint* discretizzano il circuito. Ogni *Checkpoint* introduce un tratto della pista potenzialmente diverso da quello precedente. Per esempio, il $Checkpoint_n$ potrebbe essere il punto di entrata di un tratto della pista accessibile ad un numero massimo di 4 concorrenti insieme, mentre il successivo $Checkpoint_{n+1}$ potrebbe esporre un tratto pi'u stretto e quindi accessibile solo a 2 concorrenti. Schematizzando, il *Checkpoint* è caratterizzato da:
  - **molteplicità**: ovvero il numero di concorrenti che possono trovarsi contemporaneamente nel tratto a seguire;
  - **posizione nella pista**: un *Checkpoint* può essere il traguardo, l'inizio del settore, la fine di un settore, all'uscita dei box, l'entrata per i box, i box oppure un punto intermedio fra altri due *Checkpoint*;
  - **tempi di arrivo**: ogni *Checkpoint* tiene traccia dell'istante in cui un concorrente ci è passato sopra.
- **Path**: rappresenta una delle possibili traiettorie da usare per andare da un *Checkpoint* a quello successivo. La traiettoria presenta un numero di *Path* uguale alla molteplicità del *Checkpoint* che la precede. Ogni *Path* è descritto da:
  - lunghezza
  - angolo
  - grip, ovvero l'aderenza sul tratto

  Normalmente i path appartenenti allo stesso tratto differiscono di poco.
- **Iteratore**: l'unità permette di sapere la struttura della pista. Si suppone venga usata per sapere quale *Checkpoint* ne segue un altro, oppure per sapere dove sia il *Checkpoint* di inizio box.

3.1.4 Stats

Questa componente mantiene la storia della gara e offre un insieme di funzionalità che permettono di elaborare tali dati per offrirne differenti viste:

- migliori performance in un determinato istante di tempo
- classifica aggiornata per istante di tempo
- informazioni sui concorrenti relative ad un particolare lap, checkpoint o settore (in una specifica lap);

3.1.5 Box

Il *Box* è l'entità che si occupa di gestire la configurazione e la corsa di un concorrente. Durante la competizione, il *Box* verifica costantemente lo stato dell'auto e fornisce eventuali cambi di strategia se ritenuto opportuno. Inoltre decide quando i pitstop del concorrente con tutte le caratteristiche ad esso legate, quali:

- benzina da aggiungere nel serbatoio
- gomme da montare

Ogni *Box* è caratterizzato da uno fra 4 tipi di strategia, diversi per grado di "ottimismo" nelle valutazioni e nei calcoli dati lo stato della macchina, le medie calcolate e lo stile di guida del concorrente:

1. **Cautious**: cauto, sottostima il numero di giri ancora fattibili;
2. **Normal**: stima abbastanza realistica delle possibilità del concorrente, considera anche un margine di errore nei calcoli per effettuare una valutazione;
3. **Risky**: le stime vengono effettuate in base a calcoli esatti che di solito non tengono in considerazione fattori che nella realtà possono incidere in modo negativo;
4. **Fool**: nella realtà normalmente non si arriva a tanto, ma per fini di test è stato inserito anche un tipo di strategia che sovrastima le possibilità del concorrente, portandolo a squalifica quasi certa.

Ciò che il box suggerisce al concorrente durante la gara è:

- stile di guida. Verrà suggerito uno stile più conservativo se i consumi si sono rivelati maggiori del previsto e viceversa;
- numero di lap al pitstop

Il *Box* riceve informazioni sullo stato del concorrente alla fine di ogni settore e ricalcola la strategia alla fine del secondo settore. Il concorrente richiede la nuova strategia al box in prossimità del checkpoint dove è possibile proguire o andare ai box.
È sembrato più realistica la scelta di non calcolare la strategia alla fine del terzo settore, perchè si suppone che nella realtà non si possa essere così veloci da calcolare una nuova strategia istantaneamente alla fine del circuito con i dati del terzo settore. È piuttosto più probabile che qualunque cambio di strategia o richiesta di rientro ai box venga stabilita già alla fine del secondo settore. In modo che in prossimità dei box il concorrente possa ottenere l'informazione istantaneamente e possa quindi decidere come e dove procedere.

3.1.6 Monitor

La componente *Monitor* e costituita dall'insieme delle unità concepite per esporre le informazioni e i dati prodotti a basso livello dal simulatore. Ne esistono due tipi:

- **monitor di competizione**: utilizzato per offrire informazioni riguardanti la gara e i singoli concorrenti;
- **monitor di box**: utilizzato per esporre i dati relativi alle computazioni del box, e le informazioni grezze legate al concorrente appartenente alla sua scuderia e i dati rielaborati di settore in settore per formulare nuove strategie.

3.1.7 Screen

Gli *Screen* sono la parte più alta del sistema. Servono a mettere in connessione l'utente finale e la parte logica del simulatore. Tramite gli *Screen* è possibile ricevere aggiornamenti grafici (di base) sullo stato di avanzamento della simulazione sotto il punto di vista dei singoli box o della gara complessiva.

3.1.8 Configurator

Questa componente offre la possibilità di configurare le parti parametriche del sistema, ovvero:

- concorrenti
- competizione (i parametri elencati nella sezione *Competition*
- box

Il *Configurator* ha inoltre l'onere di inviare i parametri così configurati alle componenti opportune.

3.2 Interazione fra le componenti

Nella seguente sezione verranno spiegate le interazione principali fra le componenti.

3.2.1 Configurator-Competition

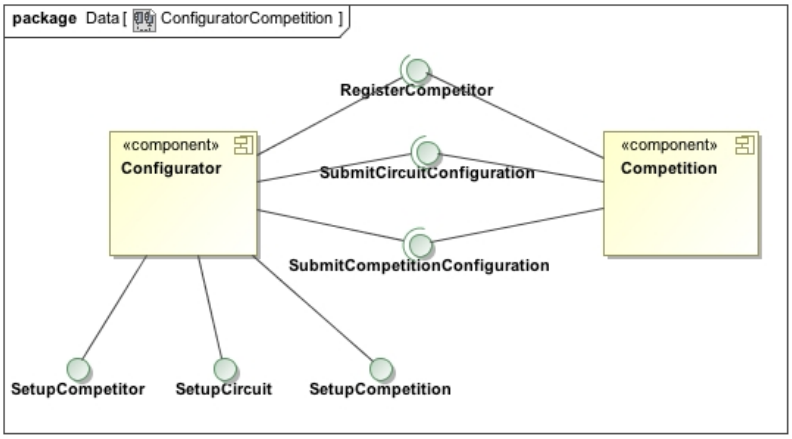

Figura 2: Protocol / Interface diagram - Configurator/Competition

Al livello più alto della fase di configurazione c'è la componente *Configurator*, la quale viene utilizzata per impostare i parametri relativi alla *Competition*. Tali parametri vengono prima impostati dall'utente (o letti da file) tramite le interfacce **SetupCompetitor**, **SetupCompetition** e **SetupCircuit** e successivamente inviati alla componente *Competition* per l'inizializzazione.
RegisterCompetitor è utilizzata una volta per ogni competitor da iscrivere.

3.2.2 Competition-Competitor

L'interazione fra la *Competition* e il *Competitor* avviene, come per le altre interazioni *Competition*-componente in fase di configurazione e avvio. La *Competition* si occupa di ricevere i parametri relativi a ogni *Competitor* dal *Configurator* (come verrà spiegato fra poco) per poi inizalizzare il competitor. Quando tutti i concorrenti sono pronti e anche i *Box*, l'interfaccia **Start** verrà utilizzata per dare il via ai concorrenti.

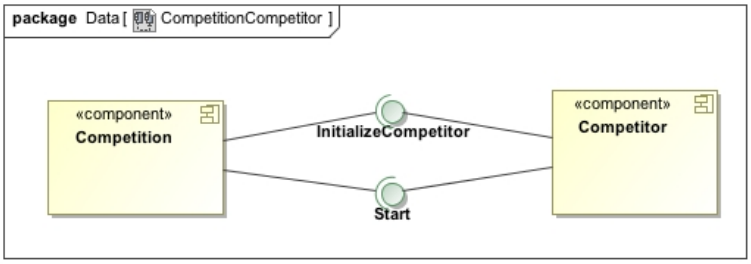

Figura 3: Protocol / Interface diagram - Competition/Competitor

3.2.3 Competition-Monitor

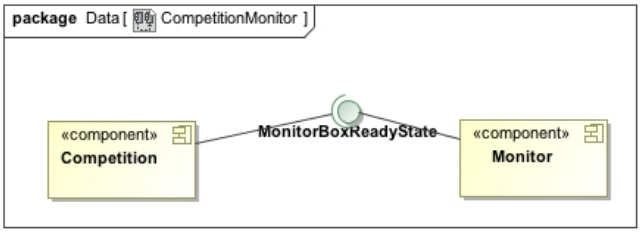

Figura 4: Protocol / Interface diagram - Competition/Monitor

L'interazione *Competition-Monitor* riguarda solo una piccola fase dell'inizializzazione, processo che verrà esplicato in dettaglio in seguito. A concorrenti registrati, la *Competition* sfruttera un interfaccia offerta dal monitor per sapere quando tutti i *Box* sono pronti e avviati. L'interfaccia è l'unica illustrata in figura.

3.2.4 Competition-Circuit

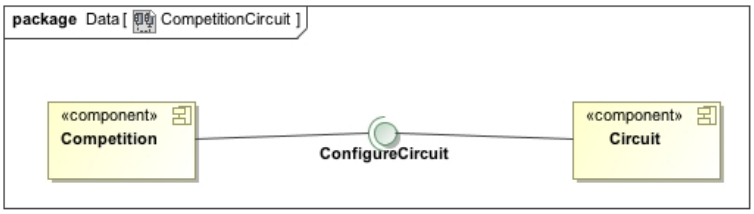

Figura 5: Protocol / Interface diagram - Competition/Circuit

Anche in questo caso, l'interazione *Competition-Circuit* si ha in fase di inizializzazione quando le due componenti entrano in contatto per la configurazione. La *Competition* inizializza cioè il circuito impostando i parametri che lo caratterizzano, quali:

- numero di checkpoint
- caratteristiche dei tratti fra checkpoint (lunghezza, angolo ...)
- posizione dei checkpoint di entrata e uscita box
- posizione del checkpoint traguardo

3.2.5 Competition-Stats

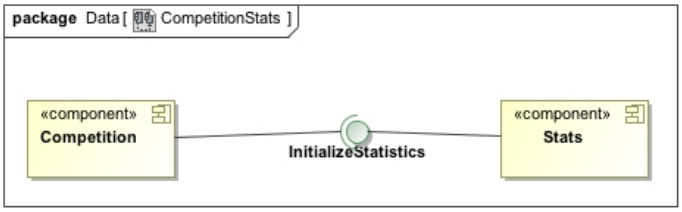

Figura 6: Protocol / Interface diagram - Competition/Stats

La componente *Stats* ottiene i parametri di configurazione in fase di inizalizzazione dalla *Competition*, tramite **InitilizeStatistics**.

3.2.6 Competitor-Stats

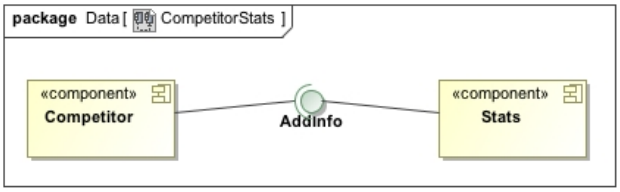

Figura 7: Protocol / Interface diagram - Competitor/Stats

La componente *Stats* offre al *Competitor* l'interfaccia **AddInfo** che il concorrente utilizza per fornire costantemente a *Stats* dati aggiornati riguardo alla

gara in corso (dati relativi al singolo concorrente). Ad ogni checkpoint quindi il concorrente manda un aggiornamento a *Stats* che poi verranno utilizzate per effettuare calcoli di insieme riguardo alla gara o per essere mandate a chi le richiedesse.

3.2.7 Competitor-Circuit

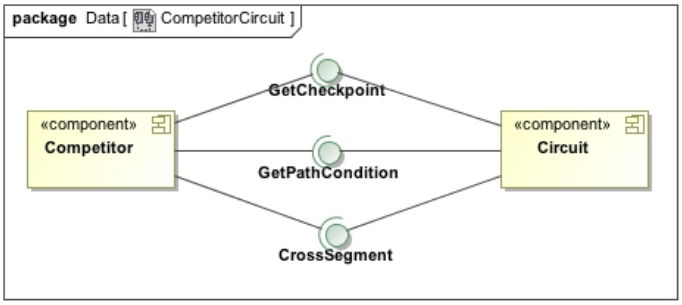

Figura 8: Protocol / Interface diagram - Competitor/Circuit

L'interazione *Competitor-Circuit* avviene durante lo svolgimento della competizione. Il concorrente sfrutta le interfacce del *Circuit* per ottenere informazioni sul circuito e "informarlo" degli spostamenti nel corso della gara.
GetCheckpoint garantisce che il concorrente ottenga sempre il checkpoint corretto a seconda della posizione corrente. **GetPathCondition** permette di ottenere informazioni sulle caratteristiche statiche e dinamiche della tratto da attraversare. Le caratteristiche dinamiche sono legate ai concorrenti attualmente presenti sul tratto. **CrossSegment** assicura che il tratto possa essere attraversato senza collisioni e che il *Circuit* possa tracciare l'avvenuto passaggio dell'auto.

3.2.8 Monitor-Stats

La componente *Monitor* si appoggia a *Stats* per poter reperire le informazioni da essere esposte. Per questo motivo *Stats* offre un insieme di interfacce finalizzate a fornire i dati grezzi di competizione sotto forma di viste utili alla "pubblicazione".
GetBestPerformance fornisce i migliori tempi relativi a settori e giro.
GetCompetitorInfo reperisce informazioni legate al singolo concorrente, come ad esempio lo stato della macchina ad un determinato istante.

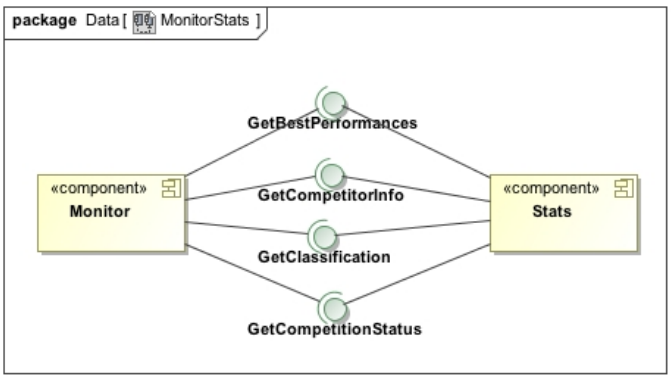

Figura 9: Protocol / Interface diagram - Monitor/Stats

GetClassification, come dice il nome, ritorna informazioni legate alla classifica.
GetCompetitionStatus espone informazioni legate alla competizione nel suo insieme, come ad esempio la posizione dei concorrenti nel circuio in un determinato istante di tempo.

3.2.9 Configurator-Box

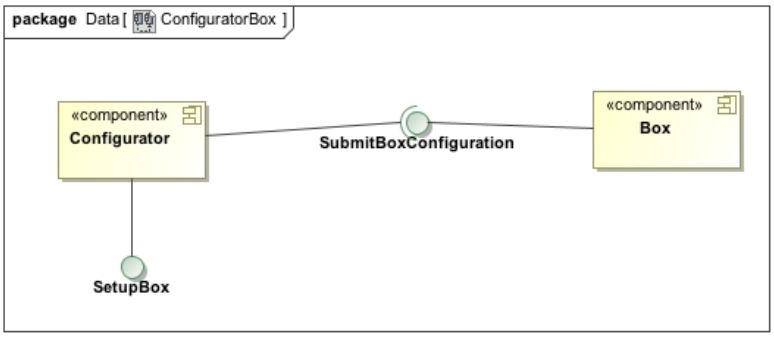

Figura 10: Protocol / Interface diagram - Configurator/Box

La componente *Configurator* offre anche la possibilità di configurare un *Box*. Per questo motivo il *Box* espone un'interfaccia da utilizzare per sottomettere i parametri di configurazione.

3.2.10 Box-Monitor

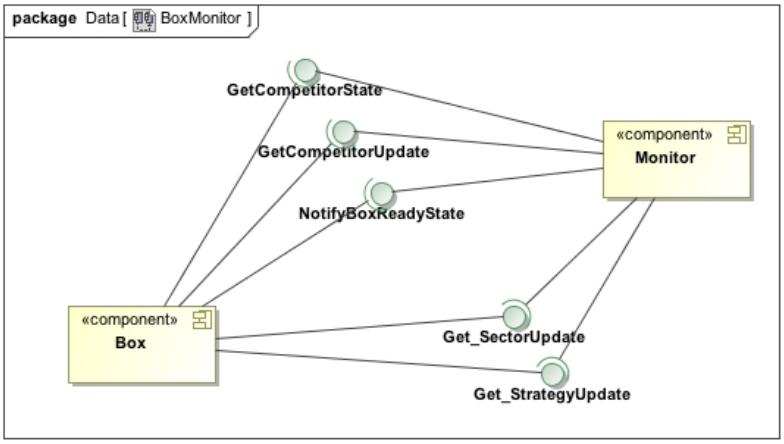

Figura 11: Protocol / Interface diagram - Box/Monitor

La prima interazione che la componente *Box* ha con il *Monitor* è in fase di inizializzazione della gara. La sequenza di azioni verrà esplicata dettagliatamente in seguito, per ora basti sapere che il *Box*, una volta configurato e pronto per monitorare la gara ed elaborare i dati del suo concorrente, dovrà mandare una notifica tramite la componente *Monitor* utilizzando **NotifyBoxReadyState**.
Le rimanenti due interfacce offerte dal *Monitor* al *Box* sono utilizzate per reperire informazioni sullo stato del concorrente (livello di benzina rimasta, usura gomme...) e aggiornamenti riguardo al posizionamento e tempi del concorrente durante la gara.
Vi sono poi altre due interfacce offerte dal *Box* al *Monitor*. Per quanto possa sembrare un po' paradossale, l'architettura acquisisce senso se si pensa che la componente *Monitor*, ad alto livello, è stata pensata per pubblicare informazioni. Tali informazioni possono riguardare il singolo concorrente e quindi essere utili al *Box*. Altre possono invece riguardare il *Box* e i suoi calcoli per essere esposte ad un utente (per esempio). Le due interfacce offerte dal *Box* infatti servono per ottenere aggiornamenti sulle operazioni interne del *Box* qualora essere dovessero essere esposte ad un qualunque client. Come vedremo più in dettaglio in seguito, questa componente è in realtà costituita da due sottocomponenti, una dedicata a *Competition* e l'altra a *Box*.

3.2.11 Screen-Monitor

La componente *Screen* comunica con il *Monitor* per ottenere informazioni utili da esporre graficamente all'utente. Tali informazioni possono riguardare la

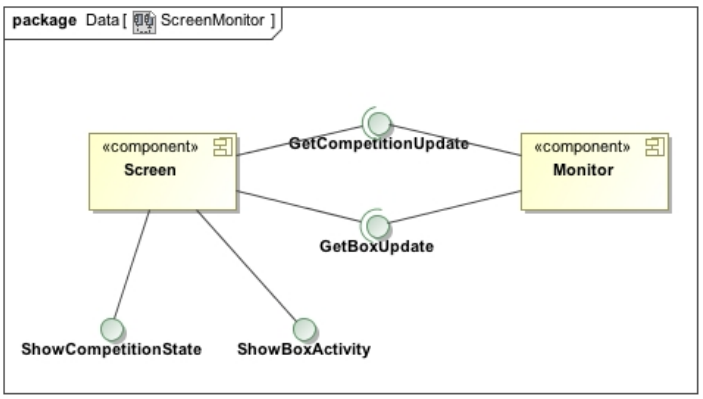

Figura 12: Protocol / Interface diagram - Screen/Monitor

competizione in senso globale, oppure essere legate ai singoli concorrenti e box.

3.2.12 Competitor-Box

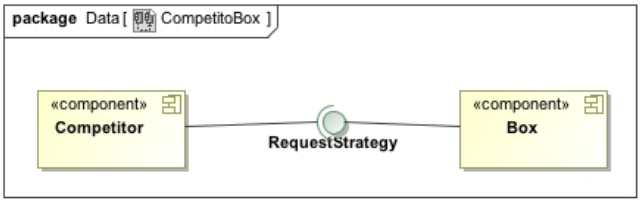

Figura 13: Protocol / Interface diagram - Competitor/Box

Man mano che la competizione procede, il *Box* colleziona i dati di gara del rispettivo concorrente per calcolarne medie e statistiche. A partire da queste informazioni produce una nuova strategia ogni giro. Per questo offre un'interfaccia **RequestStrategy** che il concorrente utilizza nel corso della gara per ottenere suggerimenti utili per proseguire. La strategia fornita dal *Box* potrebbe anche richiedere un pitstop per un rifornimento benzina e cambio gomme.

3.2.13 Strategia di simulazione

In questa sezione viene spiegata la strategia che è stata adottata per ottenere una simulazione realistica della gara. Le problematiche da affrontare sono già state discusse nel capitolo 2.2.
La soluzione prevede l'esistenza di concorrenti che simultaneamente percorrono

il circuito e un insieme di entità di supporto per evitare i problemi visti. Tali entità sono:

- **coda al checkpoint**: ogni checkpoint introduce tratti di circuito con caratteristiche diverse rispetto a quello precedente. È quindi possibile che da un tratto a molteplicità 5 si passi ad uno a molteplicità 3. L'accesso a tale tratto va quindi regolata per evitare situazioni anomale. La coda è ordinata in base ai tempi segnati (ulteriori dettagli in seguito).
- **istante di arrivo**: è l'istante in cui il concorrente è previsto arrivare. Indica l'istante reale di arrivo e dipende dal tempo di attraversamento (oltre che dal tempo accumulato fino a quel momento).
- **istante limite di arrivo previsto**: è l'istante in cui il concorrente arriverebbe sotto ipotesi ottimistiche al massimo. E' impossibile che il concorrente possa arrivare ad un checkpoint prima di questo istante.
- **flag "arrivato"**: ogni posizione della coda di un checkpoint è caratterizzata anche da questa flag. Se è settata, significa che il concorrente in tale posizione è effettivamente in attesa sul checkpoint pronto ad attraversare. Altrimenti significa che è fisicamente su un altro tratto e che il tempo segnato è un tempo limite (e non l'istante di arrivo effettivo).
- **istante di liberazione traiettoria**: ogni traiettoria presente in un tratto è contrassegnata da questo tempo. Se un concorrente arriva all'istante 42.0, la traiettoria in quel momento è libera e calcola un tempo di attraversamento di 3.0, l'istante di liberazione della traiettoria sarà 42.0+3.0 = 45.0.

Detto questo è possibile vedere l'algoritmo per l'attraversamento di un tratto dal momento in cui il concorrente arriva fisicamente ad un checkpoint fino al raggiungimento di quello successivo.
Premessa: un concorrente controlla la sua posizione su una coda ogni volta che essa viene riordinata.
Precondizione: il concorrente ha già segnato il suo tempo di arrivo effettivo sulla coda del checkpoint che vuole attraversare.

1. il concorrente setta la flag "arrivato" nella coda del checkpoint sulla posizione a lui riservata;
2. controlla in che posizione è nella coda;
3. se non è primo, attende di essere primo;
4. quando è primo, richiede l'insieme di traiettorie da cui è costituito il tratto;

5. valutando gli istanti di liberazione e le caratteristiche della traiettore, decide quale sia quella che garantisce l'arrivo al checkpoint successivo all'istante di tempo più vicino.

6. aggiunge il tempo di attraversamento calcolato (in base alle caratteristiche dell'auto e alla strategia) al tempo di liberazione del tratto o, se il tempo di liberazione segnato è minore dell'istante di arrivo del concorrente, segna come istante di liberazione *istante di arrivo+tempo di attraversamento* (poichè significa che la traiettoria era già stata liberata all'arrivo del concorente).

7. segna il tempo di arrivo effettivo sulla coda del checkpoint successivo, che verrà **riordinata** in base al nuovo tempo.

8. segna il tempo di arrivo limite su tutte le altre code, da quella successiva al checkpoint che si sta per raggiungere a quell del checkpoint appena attraversato. Ad ogni aggiornamento del tempo limite, la coda viene **riordinata**.

9. delay per fini di simulazione;

10. ritorna al passo uno per il checkpoint successivo;

La strategia adottata si presta a simulare la corsa tenendo in considerazione ad ogni istante le auto presenti sulla pista e sui tratti senza comportamenti anomali. Questo perchè:

- *quando un concorrente è primo sulla coda ed effettivamente presente sulla coda, nessun concorrente potrà diventare primo finchè non sia il concorrente stesso attualmente primo a riordinare la lista.* Questo perchè, i tempi limite segnati sono i tempi minimi. Se il tempo effettivo del concorrente attualmente primo è inferiore a tutti tempi limite, essendo minimi, non potranno diventare più piccoli successivamente. Se nel frattempo qualcun'altro arriva sul checkpoint, esso segnerà un tempo effettivo. Essendo il tempo effettivo per definizione maggiore o uguale al tempo limite, non sarà possibile che un tempo effettivo segnato successivamente possa essere minore di quello del concorrente attualmente primo.

- *quando un concorrente valuta le traiettorie, lo sta facendo in modo esclusivo.* Questa condizione è assicurata dal punto precedente. Mentre il concorrente valuta le traiettorie, nessuno può diventare primo e effettuare una valutazione parallela delle stesse traiettorie. Il concorrente riordinerà la lista solo a traiettoria valutata, scelta e segnata con istante di liberazione, permettendo ad un altro concorrente di procedere.

- *non possono esserci teletrasporti.* Prendendo il caso peggiore, ovvero un tratto ad una traiettoria, un segnale di teltrasporto avvenuto sarebbe che nel checkpoint N il concorrente A abbia tempo di arrivo effettivo 4.0 e il concorrente B 7.0. Dopo l'attraversamento di entrambi dovrebbe esserci nella coda del checkpoint N+1 il concorrente B con 6.0 in prima posizione (ad esempio) e A con 8.0 a seguire. Il che significherebbe che in qualche modo B, nonostante abbia avuto accesso al tratto dopo A, sia riuscito lo stesso a sorpassarlo arrivando primo al checkpoint dopo. La cosa non può succedere. Per i punti spiegati prima, quando A arriva primo effettivo sulla coda del checkpoint N, potra eseguire la valutazione delle traiettore in modo esclusivo. Verrà quindi segnato un istante di liberazione sull'unica traiettoria disponibile maggiore o uguale a 4.0. Supponiamo 8.0. Verrà quindi segnato 8.0 Nella coda del checkpoint N+1. A seguire verranno segnati $8.0 + \epsilon$ su tutti gli altri tratti, fino al tratto N di nuovo dove la coda verrà riordinata e B risulterà primo effettivo. Ora, supponiamo (caso pessimo) che A non possa eseguire per un bel po' di tempo. B diventa primo su N, valuta la traiettoria e segna l'istante di liberazione. Tale istante non potrà essere minore di 8.0, poichè è stato il tempo segnato da A. Quindi, qualunque tempo di arrivo effettivo segnato al checkpoint N+1, sara maggior di 8.0. Quindi B finirà in una posizione nella coda meno prioritaria rispetto ad A. In qualunque momento A venga riattivato quindi, sarà sicuro di non trovarsi B inaspettatamente davanti.

- *un numero maggiore di o uguale a 2 di concorrenti eseguono in modo concorrente*: prendendo il caso precedente, dove A ha appena finito di valutare le traiettorie del tratto N. A segna un tempo pari a 8.0 effettivo sulla coda N+1 e $N+1+\epsilon$ (crescente) su tutti gli altri fino ad N di nuovo. Supponiamo B abbia segnato un tempo inferiore a $N+1+max(\epsilon)$ su N. B valuterà le traiettore del tratto N. Nel frattempo A è primo sul tratto N+1 (assumendo che B avesse segnato un tempo limite comunque più basso del tempo effettivo di A). A valuterà le traiettorie "simultaneamente" a B. Abbiamo quindi un sistema con task concorrenti.

Del rientro ai box non se ne è ancora discusso per evitare di rendere più confusionaria la spiegazione. Comunque la cosa non costituisce un grande problema: vi saranno due checkpoint speciali, quello esattamente prima della corsia dei box e quello subito all'uscita dei box. Quando un concorrente arriva sul checkpoint precedente ai box, richiede delle informazioni di strategia ai box. Se tali informazioni richiedono un rientro, il concorrente prenderà il tratto dei box. Il box è poi rappresentato da un checkpoint come tutti gli altri nel circuito. Tale checkpoint introduce il tratto di uscita dai box, che porterà al checkpoint di

uscita (nel circuito).
I tratti nella corsia dei box hanno molteplicità pari al numero di concorrenti. Questo perchè i concorrenti vanno alla stessa velocità nelle corsie dei box e potenzialmente tutti insieme (uno dietro l'altro), quindi non possono avvenire sorpassi. L'unica ipotesi di sorpasso data dalla velocità che un concorrente richiede per il rifornimento e il cambio gomme. Tale tempo influirà sull'istante di arrivo che verrà segnato nel checkpoint di uscita.

3.2.14 Assenza di stallo

Le situazioni di stallo non si verificano per i seguenti motivi. Ragionando per assurdo, uno stallo si potrebbe verificare se: una macchina A attende la prima posizione nella coda associata al tratto 3 (per esempio). Nel frattempo Un concorrente B è in attesa sulla coda associata al tratto 2. Ma B risulta prima di A nella coda 3 e A risulta prima di B nella coda 2 e il tratto 2-3 possiede una sola traiettoria.
lo stallo porterebbe ad una contraddizione.

Dimostrazione. Se nella coda 2 ci sono A in testa seguita da B, avremo che tempo(2A)¡tempo(2B), poichè l'ordine della coda è determinato dai tempi di arrivo segnati. Se A è in attesa sulla coda 3, vuol dire che avrà aggiornato per ultima la coda 3-1, ovvero 2, e di conseguenza tempo(2A)¿tempo(3A). B invece è in attesa effettiva sulla coda 2, quindi avrà aggiornato la coda del tratto successivo con un valore dato da $t(2B) + \delta t$, con δt un valore minimo diverso da zero previsto per l'attraversamento da 2 a 3. Quindi t(2B)¡t(3B). L'ipotesi di stallo prevede che nella coda 3 vi sia prima B e poi A, quindi t(3B)¡t(3A). Avremo quindi: t(3A)¡t(2A)¡t(2B)¡t(3B)¡t(3A) che è una contraddizione, quindi la situazione non si può verificare. □

4 Architettura in dettaglio

Si spiegherà ora con maggior dettaglio l'architettura di sistema, esplicando come le principali classi implementate svolgono le funzionalità esposte dalle interfacce illustrate nel capitolo precedente.

4.1 Diagrammi delle classi

4.1.1 Competition

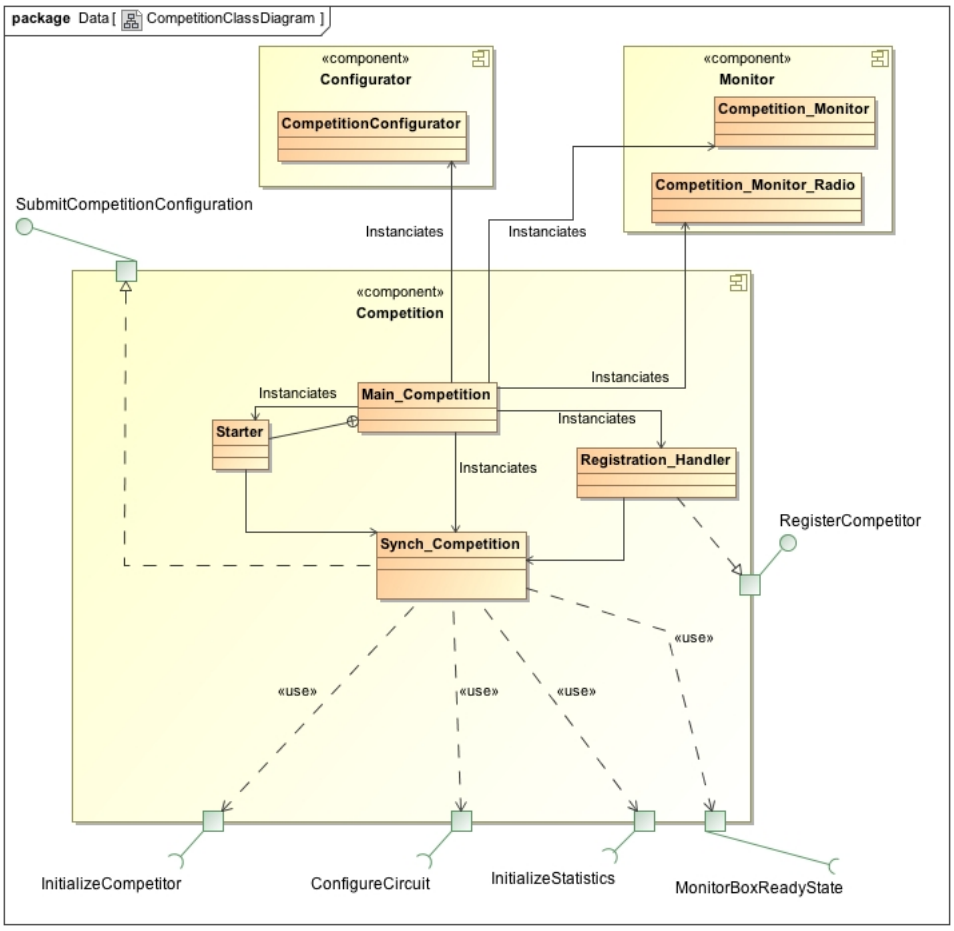

Figura 14: Class diagram - Competition

La *Competition* è, come già accennato, una componente di init.
Il tutto ha inizio a partire da **Main_Competition** che, come dice il nome, è l'unità di avvio.

Main_Competition si occupa di istanziare gli oggetti necessari all'avvio e configurazione della competizione. Per la configurazione vengono istanziati **RegistrationHandler** e **CompetitionConfigurator** (della componente *Configurator*). Per l'avvio invece **Starter**.
Ad orchestrare la scena, un unica istanza del **Synch_Competition** condivisa fra le altre 3 unità. Il loro scopo è:

- **Synch_Competition**, è una risorsa protetta che gestisce l'accesso in mutua esclusione alla configurazione della competizione e dell'inizializzazione dei concorrenti. La risorsa assicura tramite *entry* a guardia booleana che avvengano in ordine prima la configurazione della competizione (da parte del **CompetitionConfigurator**) e poi la registrazione dei concorrenti (per opera del **RegistrationHandler**). Questo vincolo è dato dal fatto che alcune impostazioni di competizioni devono essere fornite ai box dopo la configurazione del concorrente, come ad esempio il numero di lap. Inoltre fra i parametri configurabili vi è anche il numero massimo di partecipanti alla gara. Di conseguenza è prima necessario conoscere il limite per poi poter regolare il flusso di registrazioni.
  Durante la fase di configurazione (metodo Configure), vengono inizializzati *Circuit* e *Stats* tramite le interfacce illustrate nel diagramma.
  A configurazione di competizione avvenuta, verrà aperta le entry Register_NewCompetitor, utilizzata dal **RegistrationHandler** per registrare i vari concorrenti. Ad ogni invocazione verrà inizializzato un concorrente (**TaskCompetitor**) con un iteratore al circuito e le impostazioni passate in input. Il task del concorrente così istanziato rimarrà in attesa dello "start".
  Oltre alla funzionalità di configurazione, questa classe offre anche la funzionalità di avvio (metodo Start). Tale entry si aprirà solo quando tutti i concorrenti previsti sono stati iscritti. Viene invocata dall'unità **Starter**. Il metodo mette il task richiedente in attesa sulla risorsa **StartHandler** (componente *Monitor*) sull'entry MonitorBoxReadyState. Quanto tutti i box avranno dato il loro ok (maggiori dettagli a seguire), il thread potrà continuare la sua esecuzione e passare allo Start di tutti i concorrenti in attesa di partire.

- **Starter** è il task finalizzato a gestire l'avvio della competizione. Una volta avviato dal main, il task utilizza il metodo Ready offerto dal package *Competition* per manovrare l'avvio. All'interno del metodo, prima si mette in attesa che tutti i concorrenti si siano iscritti (utilizzando il metodo Wait del singleton di **Synch_Competition**) per poi invocare il metodo Start dello stesso **Synch_Competition**, descritto poche righe più sopra.

- **RegistrationHandler** è l'oggetto che rimane in attesa di concorrenti. Più precisamente, è un server dedicato ad accogliere le richieste di registrazione dei concorrenti. Con il supporto di Polyorb è possibile invocare questo oggetto da remoto. Ad ogni richiesta, vengono salvati i parametri di configurazione in un file xml il cui nome e locazione verranno passati a **Synch_Competition** per i effettuare il resto delle procedure di inizializzazione del concorrente. Di ritorno ci saranno l'ID del concorrente, il numero di lap, la lunghezza del circuito e il corbaloc del **Competition_Monitor** che il *Box* dovrà utilizzare per rimanere sincronizzato sugli sviluppi del rispettivo concorrente.

4.1.2 Competitor

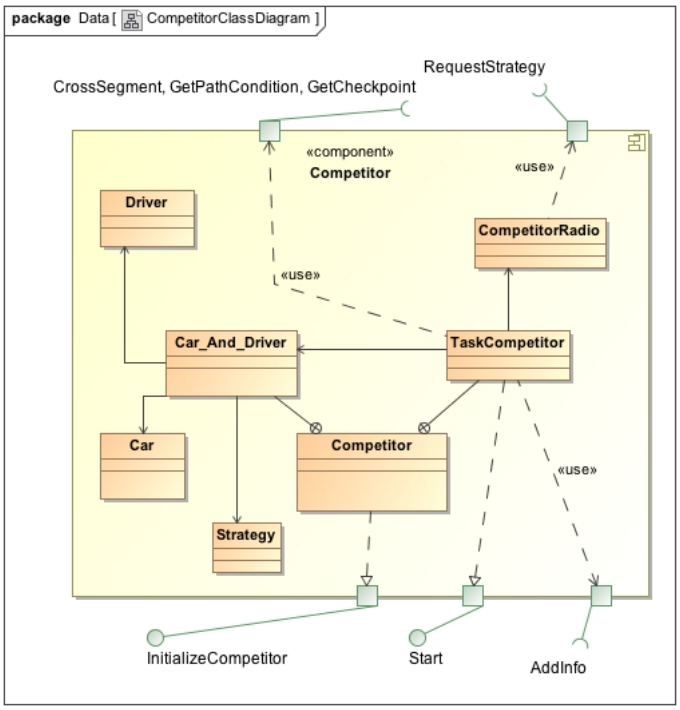

Figura 15: Class diagram - Competitor

L'unità principale del *Competitor* è il **TaskCompetitor**. È la risorsa attiva che si occupa di percorrere la pista e valutare la traiettoria ottima in base alle condizioni del tratto e dei competitor presenti nelle vicinanze. Le informazioni statiche del concorrente sono contenute in **Driver**. **Car** mantiene alcune carat-

teristiche statiche sulla macchina e altre dinamiche, come il livello di benzina e l'usura delle gomme. **Strategy** mantiene le seguenti informazioni:

- Tipo gomme
- Stile di guida
- Numero di giri al pitstop (se 0, bisogna rientrare al pitstop)
- Durata del pitstop (nel caso il pitstop sia avvenuto)
- Livello di Gas (dopo il pitstop in caso sia avvenuto)

Stile di guida e Numero di giri al pitstop vengono aggiornate ad ogni giro, Tipo gomme, Durata del pitstop e Livello di gas vengono tenute in considerazione solo in caso di pitstop. Tipo di gomme e livello di gas vengono usate per aggiornare le impostazioni di **Car**. La durata del pitstop per aggiornare correttamente il tempo di arrivo segnato nel checkpoint di uscita dai box.
Quando un concorrente si trova sul checkpoint prima della corsia dei box, effettua una chiamata al *Box* tramite la **CompetitorRadio** per richiedere una strategia aggiornata. Se la strategia tornata contiene un numero di giri al pitstop pari a 0, il **TaskCompetitor** valuterà e percorrerà la corsia dei box invece che quella del circuito, per poter effettuare il pitstop.

Il **TaskCompetitor** e tutte le unità associate vengono inizializzate in fase di init tramite i seguenti metodi offerti dal package **Competitor**:

- Configure_Driver
- Configure_Car
- Costruttore **TaskCompetitor**

4.1.3 Circuit

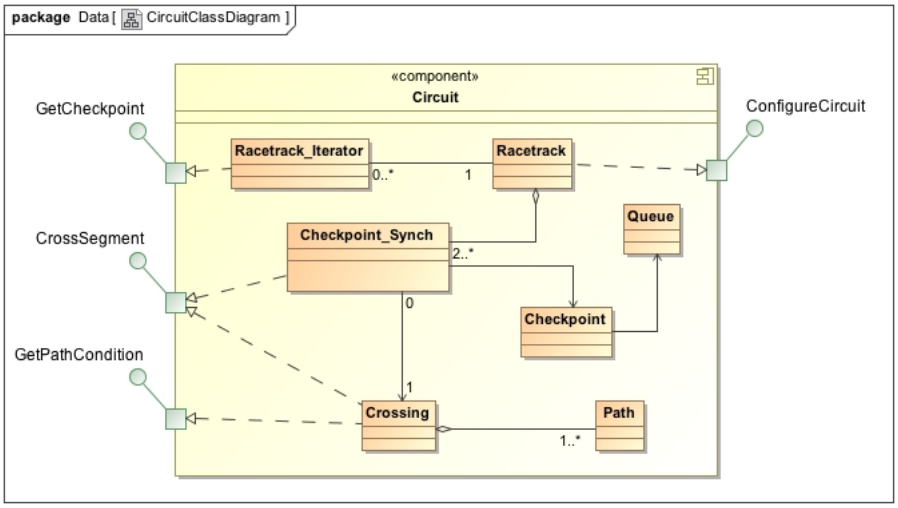

Figura 16: Class diagram - Circuit

Il circuito prende forma a partire da **Racetrack**. Questa unità si occupa di leggere e parsare il file xml dato in input e ricavarne i parametri per la creazione del circuito. Il file XML descrittore del circuito elenca i checkpoint presenti in ogni settore (che sono 3 costanti) e per ognuno specifica le caratteristiche del tratto di pista a seguire:

- lunghezza
- angolo
- molteplicità (numero di concorrenti che possono attraversare contemporaneamente il tratto)
- grip, ovvero aderenza sul tratto

Vi sono inoltre 3 checkpoint che dovranno avere uno dei seguenti attributi booleani:

- goal, ovvero il checkpoint è il primo della pista
- prebox, ovvero dal checkpoint è possibile raggiungere i box
- exitbox, ovvero il checkpoint di arrivo una volta vuori dalla corsia dei box

Dati questi parametri, vengono inizalizzati tutti i checkpoint stabiliti ed inseriti in un array chiamato **Racetrack**. Questo sarà l'array iterato dal **Race_Iterator**.

Ogni checkpoint è del tipo **Checkpoint**, il quale contiene tutte le informazioni elencate in precedenza oltre ad una coda per gestire i concorrenti che tentano di accedere al tratto. Il **Checkpoint** è poi inserito in una struttura protetta denominata **Checkpoint_Synch** finalizzata a regolare l'accesso in mutua esclusione. La risorsa inoltre incapsula la lista di **Path** che costituiscono il tratto associato al checkpoint. Infine offre una serie di metodi da utilizzarsi per interagire con le risorse sottostanti.
I **Path** appena accennati vengono generati in fase di creazione del **Checkpoint** a partire dalle informazioni di base reperite dal file di configurazione. Vengono generati tanti **Path** quanto la molteplicità impostata del tratto. Ogni **Path** riceve valori di lunghezza che possono variare. Bisogna assicurare 1.5 m di larghezza per ogni macchina (approssimativamente). La prima riceve i valori di base e viene virtualmente posizionata su un bordo del tratto. Le altre vengono posizionate man mano una di fianco all'altra, quindi la lunghezza della traiettoria cresce in rapporto alla distanza dalla prima e all'angolo del tratto. Tutti i **Path** di un tratto vengono incapsulati in una risorsa protetta denominata **Crossing** che ne regola l'accesso in mutua esclusione e offre i metodi di accesso pubblici.
Infine viene creata una corsia dei box coerente con la distanza fra i checkpoint "prebox" e l'"exitbox". I tratti prima e dopo il checkpoint dei box sono a molteplicità uguale al numero di concorrenti. Questo perchè potenzialmente ai box potrebbero esserci contemporaneamente tutte le auto. Inoltre si è deciso di posizionare il checkpoint del box in coincidenza con il traguardo. Quindi ogni box è anche un goal.

Verranno ora elencati metodi più rilevanti esposti dal **Checkpoint_Synch** per poter mettere in pratica la strategia di attraversamento descritta nella sezione 3.2.13:

procedure Signal_Arrival(CompetitorID_In : INTEGER)
Il metodo marca nella coda del checkpoint l'arrivo effettivo del concorrente;

procedure Signal_Leaving(CompetitorID_In : INTEGER)
Il metodo marca nella coda del checkpoint l'uscita del concorrente;

procedure Set_ArrivalTime(CompetitorID_In : INTEGER; Time_In : FLOAT)

Il metodo segna il tempo previsto di arrivo del concorrente nella coda del checkpoint;

procedure Remove_Competitor(CompetitorID_In : INTEGER)
Il metodo rimuove il competitor dalla coda. Ciò significa che l'id e il tempo del competitor non apparirà più nella coda. Questo metodo è utilizzato, per esempio, quando un concorrente finisce la gara prima di altri e deve quindi liberare i checkpoint per evitare di creare starvation;

function Get_Time(CompetitorID_In : INTEGER) return FLOAT
La funzione ritorna il tempo segnato sulla coda del concorrente con ID dato in input;

entry Wait_Ready(Competitor_ID : INTEGER) Nel momento in cui un concorrente arriva fisicamente su un checkpoint, dopo aver marcato il suo arrivo utilizza questo metodo per sapere quando arriva il suo turno per attraversare (viene cioè posizionato nella prima posizione della coda);

procedure Get_Paths(Paths2Cross : out CROSSING_POINT; Go2Box : BOOLEAN) Quando il concorrente sa di essere primo sulla coda del **Checkpoint** (in seguito all'invocazione del metodo Wait_Ready), potrà invocare questa procedura per ottenere l'insieme di **Path** che costituiscono il tratto e procedere alla valutazione. Il booleano "Go2Box", se valorizzato a "true", impone al **Checkpoint_Synch** di tornare (se presente) l'insieme di **Path** relativi alla corsia dei box. Si è sicuri che nessuno starà effettuando operazioni sul tratto nel frattempo perchè tale azione da parte degli altri concorrenti non è ammisibile fino a che il **Competitor** corrente non abbia segnalato la sua partenza dal checkpoint tramite Signal_Leaving.

I metodi della risorsa **Crossing** invece servono a ritornare le caratteristiche di ogni **Path** (a partire dall'indice) e a aggiornare l'istante temporale segnato nel path.

Infine sono offerti un insieme di metodi da utilizzare insieme al **RaceTrack_Iterator** per navigare iterare il circuito. I metodi sono più rilevanti sono:

procedure Get_CurrentCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;
CurrentCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint correntemente "puntato" dall'iteratore;

procedure Get_NextCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;

NextCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint successivo. Nota Bene: il checkpoint dei box non è previsto essere ritornato da questo metodo. Per ottenere il checkpoint dei box è necessario utilizzare Get_BoxCheckpoint;

procedure Get_PreviousCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;
PreviousCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint precedente, con le stesse regole del metodo precedente;

procedure Get_ExitBoxCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;
ExitBoxCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint all'uscita della corsia dei box;

procedure Get_BoxCheckpoint
(RaceIterator : in out RACETRACK_ITERATOR;
BoxCheckpoint : out CHECKPOINT_SYNCH_POINT)
Per ottenere il checkpoint del box;

function Get_Position
(RaceIterator : RACETRACK_ITERATOR) return INTEGER
Per tornare la posizione corrente dell'iteratore.

Non è stato necessario inserire **Racetrack** o il suo iteratore in una risorsa protetta perchè ogni concorrente disponde della sua copia dell'**Racetrack_Iterator**.

4.1.4 Stats

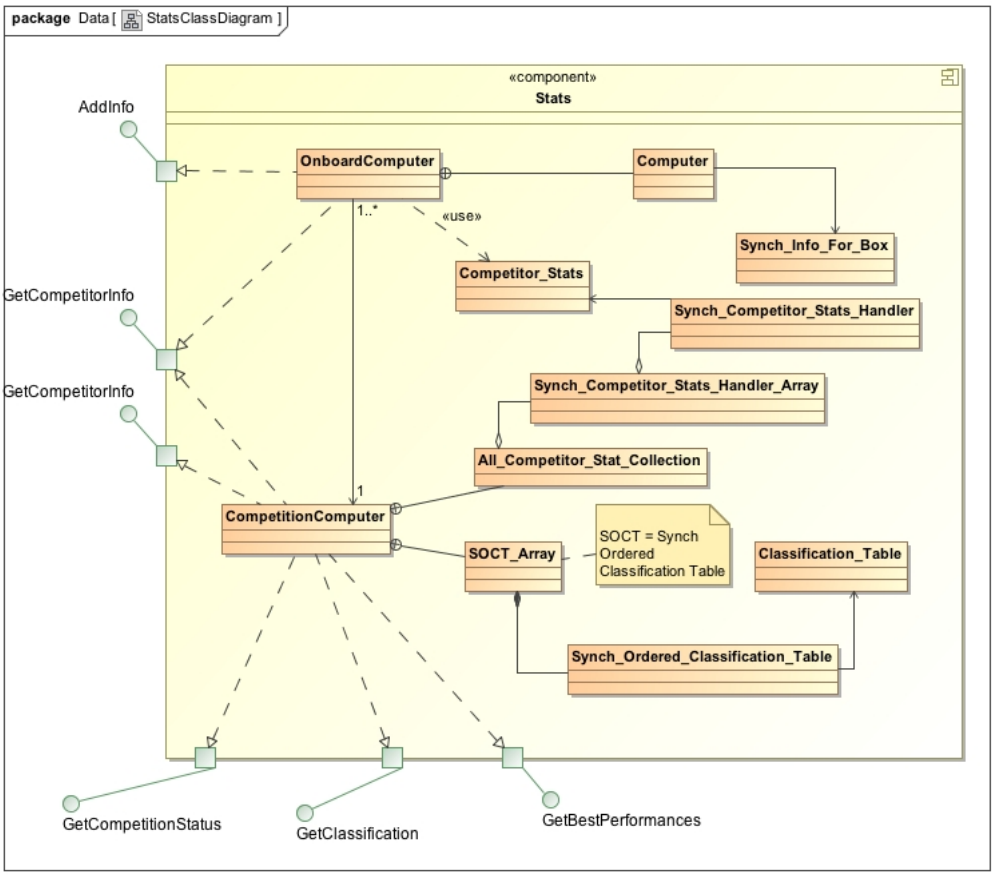

Figura 17: Class diagram - Stats

La componente *Stats* è implicitamente suddivisa in 2 sottocomponenti: **OnboardComputer** e **CompetitionComputer**. Prima di discutere dei dettagli delle due, è necesario introdurre la struttura di una risorsa che viene utilizzata come pacchetto per il trasporto degli aggiornamenti tra *Competitor*, **OnboardComputer** e **CompetitionComputer**. La risorsa è un tipo record denominato **Competitor_Stats**, contenente i seguenti dati:

- **Time : FLOAT;**
  l'istante a cui fa riferimento l'aggiornamento
- **Checkpoint : INTEGER;**
  il checkpoint che introduceva il tratto che è stato attraversato (completato) all'istante TIME

- **LastCheckInSect : BOOLEAN;**
  se true, il checkpoint è l'ultimo di un settore
- **FirstCheckInSect : BOOLEAN;**
  su treu, il checkpoint è il primo del settore
- **Sector : INTEGER;**
  il settore a cui appartiene il checkpoint
- **Lap : INTEGER;**
  la lap in cui a cui l'aggiornamento fa riferimento
- **GasLevel : FLOAT;**
  il livello di gas presente nel serbatoio al momento in cui il tratto è stato attraversato (completato)
- **TyreUsury : PERCENTAGE;**
  la percentuale di usura gomme al momento in cui il tratto è stato attraversato (completato)
- **BestLapNum : INTEGER;**
  la miglior lap fatta dal concorrente dall'inizio della gara all'istante TIME
- **BestLaptime : FLOAT;**
  il tempo della miglior lap fatta dal concorrente dall'inizio della gara all'istante TIME
- **BestSectorTimes : FLOAT_ARRAY(1..3);**
  ogni indice dell'array indica un settore (quindi indice 1 indica il settore 1). Detto ciò, l'array contiene il miglior tempo fatto dal concorrente per ogni settore dall'inizio della gara all'istante TIME
- **MaxSpeed : FLOAT;**
  la massima velocità raggiunta dall'inizio della gara all'istante TIME
- **PathLength : FLOAT;**
  la lunghezza della traiettoria scelta per attraversare il tratto

Ora vediamo piu dettagliatamente le due sottocomponenti accennate precedentemente:

- **OnboardComputer**:
  è un computer dedicato ad ogni singolo concorrente, denominato **OnboardComputer**. Ogni concorrente ne mantiene un'istanza che utilizza per aggiornare le statistiche di checkpoint in checkpoint. Più precisamente, ogni concorrente possiede un'istanza di **Computer**, un record che

colleziona le statistiche del singolo concorrente e le informazioni statiche sulla configurazione della competizione.
Ogni **Computer** mantiene inoltre un riferimento ad una risorsa che nel diagramma è **Synch_Info_For_Box**. Tale risorsa viene utilizzata per mantenere la lista delle informazioni necessarie al box. È più che altro un meccanismo di ottimizzazione per avere gli aggiornamenti sul singolo competitor immediatamente disponibili quando il *Box* ne richiede.

Il *Competitor* utilizza il metodo AddInfo di **OnBoardComputer** per inviare le informazioni relative al tratto appena attraversato. Da quando tali informazioni sono sottomesse, vengono effettuati i seguenti passaggi:

- se la fine del tratto coincide con la fine del settore, viene verificato il tempo impiegato per attraversare tale settore (facendo riferimento anche ai dati passati) e se migliore di quello precedentemente salvato (in **Computer**), viene aggiornato quello vecchio.
- se la fine del tratto coincide con la fine del settore vengono anche aggiornate le informazioni in **Synch_Info_For_Box**. Si ricorda infatti che il *Box* riceve le informazioni aggiornate alla fine di ogni settore.
- se la fine del tratto corrisponde con la fine del giro, vengono aggiornato il miglior tempo di giro come fatto per i settori.
- una volta effettuati tutti i controlli, le informazioni vengono impacchettate e inviate a **CompetitionComputer** per ulteriori controlli e per essere salvate.

- **CompetitionComputer**:
  è invece il computer dedicato al calcolo delle statistiche globali, ovvero riguardanti tutti i partecipanti. Qualunque informazione che riguardi uno o più concorrenti è da richiedere a questa entità. Il **CompetitionComputer** si appoggia a due risorse per l'archiviazione e l'organizzazione dei dati:

  All_Competitor_Stats_Collection: la risorsa è destinata a mantenere la storia degli aggiornamenti di tutti i concorrenti. Ad ogni competitor è dedicato un array di **Synch_Competitor_Stats_Handler**. Questa entità serve ad racchiudere un **Competitor_Stats** nel corpo di una risorsa protetta. L'array è inizializzato con capacità pari a N° Lap * N° Checkpoint, poichè le informazioni vengono aggiunte ad ogni checkpoint. Ogni posizione dell'array quindi fa riferimento ad un checkpoint e l'array è intrinsecamente ordinato per tempo crescente. **Synch_Competitor_Stats_Handler** mette a disposizione un entry

di get (Get_All) che si apre solo nel momento in cui la risorsa è inizializzata. Questo permette di mettere in attesa i client che richiederanno informazioni non ancora disponibili.

SOCT_Array: questo array è la parte più alta di una struttura utilizzata per il supporto alla creazione della classifica. Nel punto più basso c'è **Classification_Table**, unità finalizzata a raccogliere i tempi di lap dei concorrenti. A gestire questa tabella virtuale c'è **Synch_Ordered_Classification_Table**, che permette di mantenere la tabella ordinata in base ai tempi e offre un set di metodi per il reperimento dei dati. Infine **SOCT_Array** (SOCT sta per Synch Ordered Classification Table) è un array in cui ogni posizione rimanda ad alla tabella della classifica della lap corrispondente all'indice.

Tramite il metodo Add_Stat, il **Computer** di **OnboardComputer** invia i pacchetti con gli aggiornamenti. Prima di salvare definitivamente un aggiornamento, viene verificato se fa riferimento ad un checkpoint di fine lap. In tal caso viene prima aggiornata la tabella della classifica corrispondente alla lap appena percorsa con l'istante di tempo segnato. Successivamente l'aggiornamento viene salvato nell'array di riferimento del concorrente, aprendo così la risorsa a chiunque la richieda o la stesse già richiedendo.

Quanto descritto costituisce le fondamenta di **CompetitionComputer**. Il metodi offerti navigano queste strutture per ottenere i dati richiesti:

procedure Get_StatsByTime
(Competitor_ID : INTEGER;
Time : FLOAT;
Stats_In : out COMPETITOR_STATS_POINT);
fornisce il primo aggiornamento con tempo maggiore o uguale all'istante dato;

procedure Get_StatsBySect(Competitor_ID : INTEGER;
Sector : INTEGER; Lap : INTEGER;
Stats_In : out COMPETITOR_STATS_POINT);
fornisce le statistiche del concorrente COMPETITOR_ID inerenti alla fine del settore richiesto nella lap richiesta;

procedure Get_StatsByCheck(Competitor_ID : INTEGER;
Checkpoint : INTEGER; Lap : INTEGER;
Stats_In : out COMPETITOR_STATS_POINT);
fornisce le statistiche del concorrente COMPETITOR_ID inerenti al checkpoint e lap richiesti;

procedure Get_BestLap(TimeInstant : FLOAT;
LapTime : out FLOAT; LapNum : out INTEGER;
Competitor_ID : out INTEGER);
fornisce il miglior giro, il tempo di tale giro e il concorrente che fatto il record;

procedure Get_BestSectorTimes(TimeInstant : FLOAT;
Times : out FLOAT_ARRAY;
Competitor_IDs : out INTEGER_ARRAY;
Laps : out INTEGER_ARRAY);
fornisce il miglior tempo per ogni settore con i concorrenti che hanno fatto il record.

procedure Get_LapClassific(Lap : INTEGER;
TimeInstant : FLOAT;
CompetitorID_InClassific : out INTEGER_ARRAY_POINT;
Times_InClassific : out FLOAT_ARRAY_POINT;
LappedCompetitors_ID : out INTEGER_ARRAY_POINT;
LappedCompetitor_CurrentLap : out INTEGER_ARRAY_POINT);
dato l'istante di tempo in input, il metodo fornisce la classifica più aggiornata con i tempi per quell'istante, compresi i concorrenti doppiati in ordine di posizione e la lap che stanno percorrendo. I concorrenti che invece non sono doppiati ma devono ancora finire la lap a cui la classifica si riferisce non vengono inclusi nella lista.

Il **Competition_Monitor**, (come vedremo poi più in dettaglio è una sottocomponente di *Monitor*) fa riferimento al **CompetitionComputer** per ottenere le informazioni di competizione.

4.1.5 Monitor

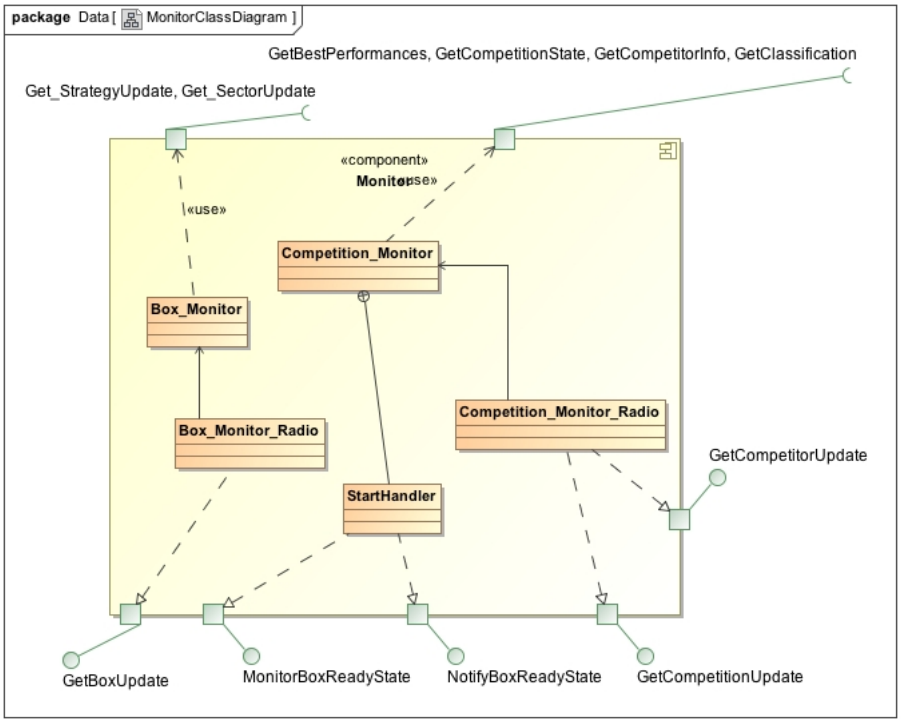

Figura 18: Class diagram - Monitor

Monitor si suddivide in due sottocomponenti: **Box_Monitor** e **Competition_Monitor**. La prima è dedicata alle informazioni esposte dal *Box*, l'altra a quelle esposte dalla *Competition*.
Box_Monitor produce i dati interrogando la componente *Box*, precisamente **Box_Data**, unità che vedremo in seguito, tramite il metodo Get_Info, grazie al quale si ottiene un oggetto che contiene le informazioni inerenti all'ultimo settore completato dal concorrente (comprese le medie di consumo) e, se disponibile, la strategia calcolata dal *Box* per la lap successiva.

Competition_Monitor si appoggia a *Stats* per reperire le informazioni richieste. I metodi che espone sono:

procedure Get_CompetitionInfo(TimeInstant : FLOAT; ClassificationTimes : out Common.FLOAT_ARRAY_POINT; XMLInfo : out Unbounded_String.Unbounded_String);
la procedura inizializza la stringa con le informazioni di competizione in

formato XML. Tali informazioni riguardano il posizionamento dei concorrenti all'istante dato (quale tratto stanno percorrendo e in che lap), le migliori performance fino all'istante dato e la classifica aggiornata all'ultima lap in corso, con tempi e concorrenti doppiati. Il metodo si appoggia ai metodi forniti da **Competition_Stats** per reperire le inormazioni necessarie.

procedure Get_CompetitorInfo(lap : INTEGER; sector : INTEGER ; id : INTEGER; time : out FLOAT; updString : out Unbounded_String.Unbounded_String);
la procedura fornisce le informazioni di un concorrente aggiornate al settore e lap richieste. Il metodo si appoggia a **OnBoardComputer** per reperire le informazioni date, precisamente utilizzando il metodo Get_BoxInfo utilizzando come parametro un riferimento al **Computer** del concorrente di cui si richiedono le informazioni.

Queste classi non comunicano direttamente con i loro clienti, ma comunicano tramite un intermediario: **Box_Monitor_Radio** e **Competition_Monitor_Radio**, ai quali viene demandato il compito di gestire le comunicazione distribuita tramite Polyorb (usando il protocollo CORBA).

4.1.6 Box

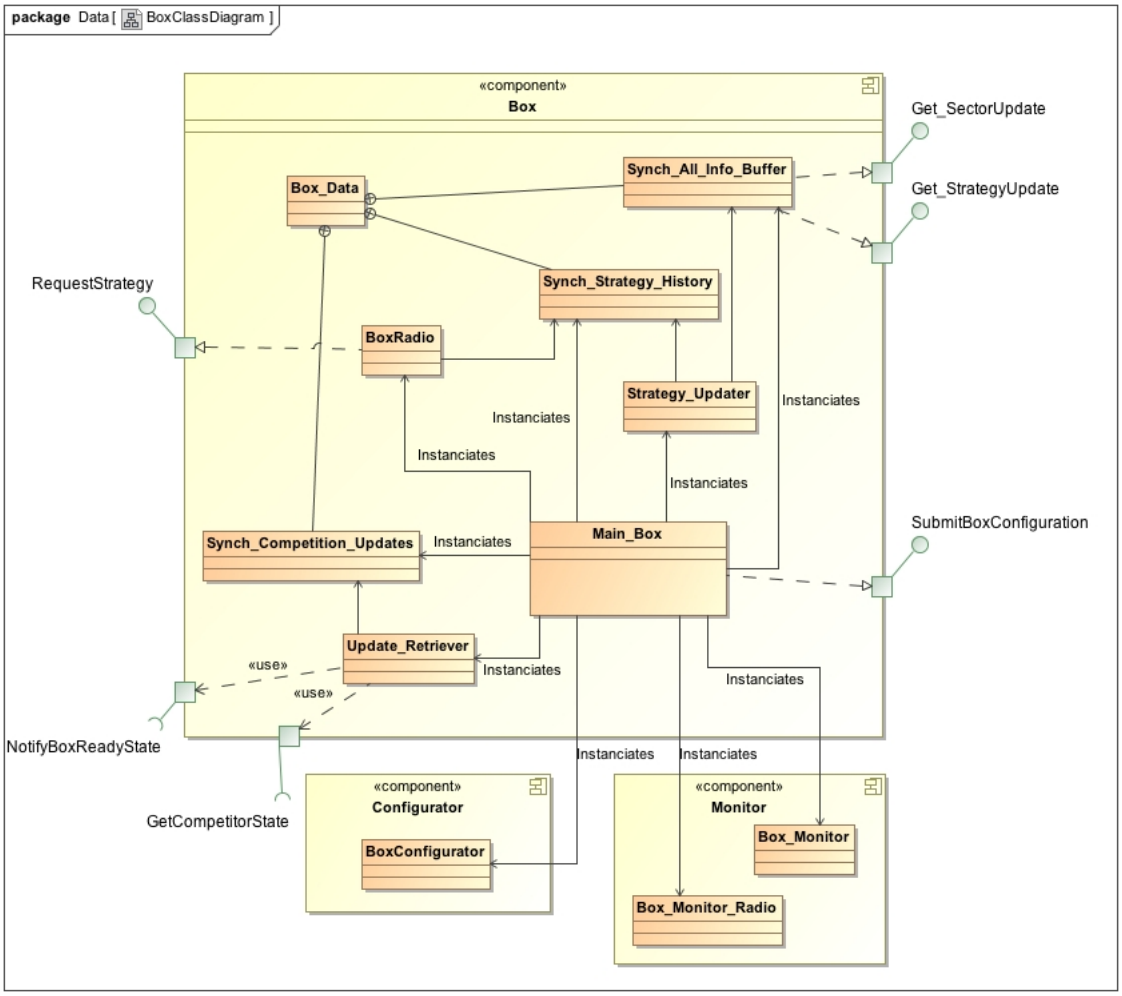

Figura 19: Class diagram - Box

Il *Box* è una componente che si occupa di assistere la gara del concorrete. A tale scopo sono stati previsti due task con le seguenti caratteristiche:

Update_Retriever:

Come il nome suggerisce, questa unit'a è finalizzata a recuperare gli aggiornamenti di gara relativi al concorrente associato al box. A tale scopo, il task si connette al **Competition_Monitor** per ottenere un pacchetto di informazioni aggiornato alla fine di ogni settore. Tali informazioni vengono convertite in un formato compatibile con il sistema (vengono ricevute in

XML e convertite in un oggetto **Competition_Update**). Fatto ciò, gli aggiornamenti vengono inseriti nel buffer **Synch_Competition_Updates**. Il task **Strategy_Updater** potrà così leggere gli aggiornamenti non appena disponibili (essendo tale buffer condiviso fra i due task).
Il task richiede le informazioni per lap e settore. Se un'informazione non è ancora stata prodotta, il thread verrà messo in attesa.

Strategy_Updated:
L'entità rappresenta una parte dell'"intelligenza artificiale" del sistema. Raccoglie gli aggiornamenti di gara dal buffer **Synch_Competition_Updates** man mano che essi vengono aggiunti. Tali aggiornamenti vengono di volta in volta usati per aggiornare le medie sui consumi e aggiornare la strategia di gara. L'aggiornamento della strategia avviene alla fine di ogni secondo settore. Una volta ottenuto l'aggiornamento del secondo settore, vengono usate le informazioni relative al 3° settore del giro precedente e quelle relative al 1° e 2° settore del giro corrente per computare una nuova strategia. Una volta che la strategia è stata calcolata viene inserita nel buffer **Strategy_History**, da dove diventa disponibile nel caso il concorrente la richieda.

A supportare la comunicazione dei dati fra entità attive vi sono 3 risorse protette contenute nel package **Box_Data**:

Synch_Competition_Updates:
Supporta la comunicazione fra **Update_Retriever** e **Strategy_History**. Contiene le informazioni di settore del concorrente raccolte dal task **Update_Retriever**. **Strategy_History** reperisce le informazioni in ordine di inserimento.

Synch_Strategy_History:
Supporta la comunicazione fra **Strategy_History** e **BoxRadio**. Il primo aggiunge una nuova strategia alla fine di ogni secondo settore al buffer. Il secondo è un tramite fra *Competitor* e *Box* e permette al **TaskCompetitor** di richiedere una strategia nuova alla fine di ogni lap tramite il metodo Get_Strategy. Se la strategia è presente, verrà tornata. Altrimenti il thread che si occupa di gestire la richiesta viene messo in attesa.

Synch_All_Info_Buffer:
Supporta la comunicazione fra *Box* e **Box_Monitor**.Questa risorse è desitnata a contenere tutte le informazioni legate ai box. Oltre alle informazioni grezze di settore sul concorrente, la risorsa colleziona anche le medie calcolate dai box e le strategie ogni qualvolta ne siano disponibili

di nuove. Le informazioni sono ordinate cronologicamente e associate ad un indice. L'indice va da 1 a $3 * LapTotali$, poichè i settori sono tre. Se un informazione relativa ad un settore e ad una lap non è disponibile, il richiedente viene messo in attesa fino a che l'aggiornamento non risulti disponibile.

Si può constatare che a concorrere per le informazioni collezionate nel *Box* non vi sono solo unità interne alla componente, ma anche thread che agiscono da altre componenti. Tali thread sono il **TaskCompetitor** lato *Competition* e l'interfaccia di visualizzazione delle informazioni del box lato *Screen*. Il primo richiede una nuova strategia alla fine di ogni lap tramite **CompetitorRadio** (lato *Competition*) che si mette in contatto con **BoxRadio** che comunica direttamente con **Synch_Strategy_History** per recuperare la strategia richiesta o essere messo in attesa.
Il secondo (*Screen*) richiede tutte le informazioni dall'indice 1 all'indice massimo contenute in **Synch_All_Info_Buffer** tramite la parte del *Monitor* destinata ai box. Le informazioni tornate possono riguardare solo un aggiornamento di settore o anche di strategia, in base a quanto disponibile per quell'indice.

Infine il **Main_Box** è destinato ad inizializzare e configurare le risorse protette condivise e i task e istanziare gli oggetti CORBA necessari alla comunicazione fra componenti.

4.1.7 Configurator

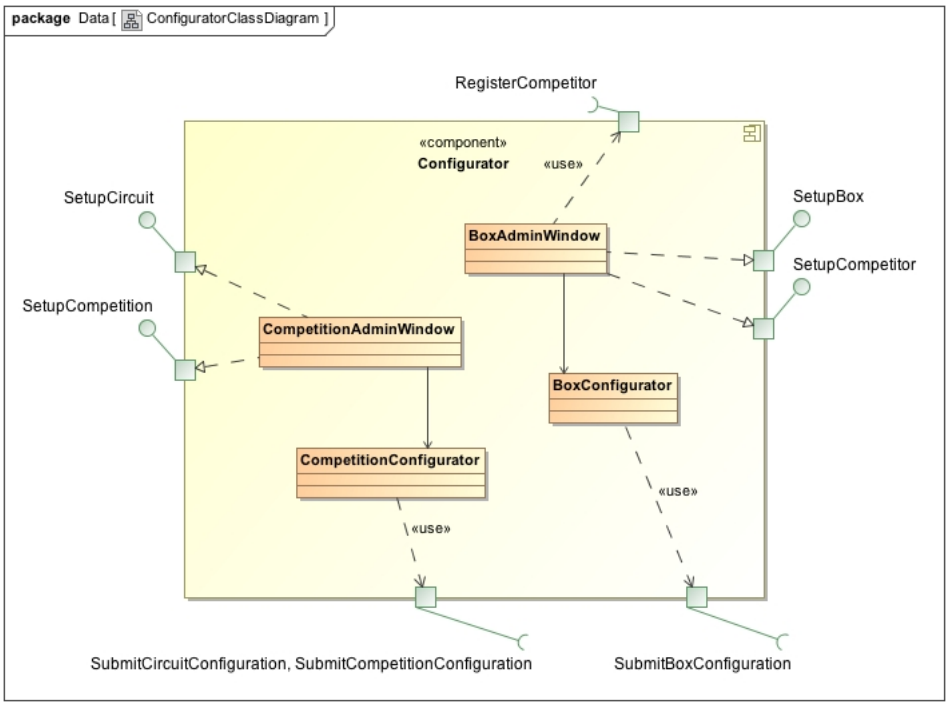

Figura 20: Class diagram - Configurator

La componente, come già accennato, gestisce la configurazione delle componenti *Box* e *Competition*. Può essere vista come suddivisa a sua volta in due ulteriori sottocomponenti, una destinata all'inserimento dati e l'altra destinata alla comunicazione dei dati alla logica sottostante. Vediamo infatti nel diagramma due classi **CompetitionAdminWindow** e **BoxAdminWindow** che offrono all'utente un'interfaccia di configurazione (scritta in Java) per *Competition* e *Box*; queste due classi sono legate ai rispettivi "configurator", ovvero classi che permettono l'intercomunicazione fra linguaggi diversi, di modo quindi che i dati inseriti nelle interfacce utente possano essere trasferiti alle componenti sottostanti per la configurazione e inizializzazione. In dettaglio:

CompetitionAdminWindow:

Utilizzata per impostare i parametri di gara, ovvero:

- numero lap
- numero concorrenti

- file da cui leggere il circuito

Una volta impostati i parametri essi verranno sottomessi via **CompetitionConfigurator** alla *Competition* per la configurazione e l'inizializzazione.

BoxAdminWindow:

Utilizzata per impostare le caratteristiche statiche del concorrente associato al box (fare riferimento al capitolo 3.1.2 per informazioni su tali parametri), dettagli riguardanti il box stesso, ovvero la strategia (fare riferimento al capitolo 3.1.5 per ulteriori dettagli) e la configurazione iniziale dell'auto, ovvero tipo di gomme e quantità di benzina iniziale. Queste informazioni vengono poi utilizzate per registrare il concorrente alla competizione (tramite il **RegistrationHandler** della componente *Competition*). Una volta registrato il concorrente si potranno ottenere il resto delle informazioni necessarie ad inizializzare il box, ovvero numero di giri e lunghezza circuito (per esempio). È quindi ora possibile inviare i parametri di configurazione al *Box* tramite il **BoxConfigurator**.

4.1.8 Screen

Figura 21: Class diagram - Screen

4.2 Analisi della concorrenza

4.2.1 Interazione Competitor - Circuit

La strategia adottata per permettere ai concorrenti di percorrere la pista è stata descritta ad alto livello nel capitolo 3.2.13. Tale strategia è indipendente dal linguaggio o dall'implementazione, quindi il codice scritto ha semplicemente tradotto la teoria in pratica. Gli aspetti legati all'implementazione a cui si è dovuto prestare attenzione invece sono descritti di seguito:

Accesso simultaneo ai checkpoint
: Per garantire che il checkpoint sia acceduto in modo mutamente esclusivo fra task, è stato sufficiente inserirlo in una risorsa protetta: **Synch_Checkpoint**. Tale risorsa offre i metodi necessari ai **TaskCompetitor** per segnalare il proprio arrivo sulla coda, "scrivere" il tempo di arrivo limite ecc. in modo da evitare race condition.

Prima posizione raggiunta
: Come si è visto, quando un concorrente arriva ad un checkpoint deve segnalare il suo arrivo effettivo e, una volta raggiunta la prima posizione nella coda del checkpoint, iniziare a valutare il path da percorrere. Per mettere

in pratica il sistema progettato, è stato necessario avvalersi di un'ulteriore risorsa protetta chiamata **Waiting_Block**. Ne viene istanziata una lista in ogni **Synch_Checkpoint** (una lista lunga quanto il numero di concorrenti). Ogni elemento della lista è associato ad un concorrente. Quando un concorrente necessita di attendere su una qualunque condizione, viene messo in attesa sul suo rispettivo **Waiting_Block**. Quando la condizione si verifica, il thread che ha causato il cambio di condizione invoca il metodo Notify nel **Waitin_Block**, che risveglia il thread che era in attesa. La condizione per cui viene utilizzata questa struttura è il raggiungimento della prima posizione nella coda del checkpoint. Il seguente diagramma descrive la sequenza di eventi che portano il **TaskCompetitor** da segnalare il suo arrivo al checkpoint all'ottenere i path tra cui scegliere per effettuare l'attraversamento del tratto.

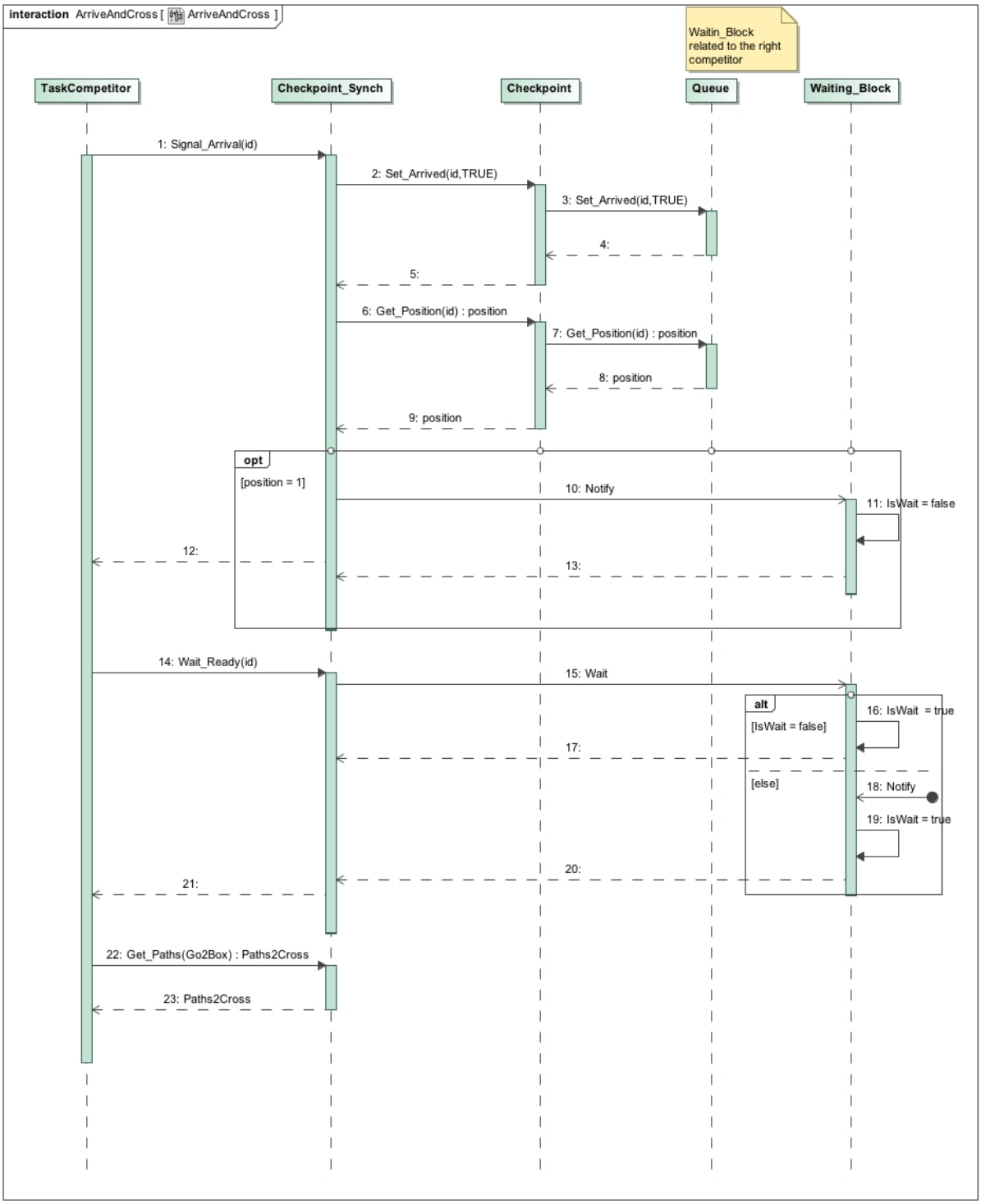

Figura 22: Sequence Diagram - Arrivo al checkpoint e recupero lista traiettorie

Prima di passare alla spiegazione del diagramma è necessario premettere in quali casi un concorrente può cambiare stato nella coda del **Synch_Checkpoint**:

- CAMBIO POSIZIONE: può avvenire quando un qualunque concorrente segna un nuovo tempo minimo (o effettivo) di arrivo con Set_ArrivalTime o quando un concorrente viene rimosso dal checkpoint con il metodo Remove_Competitor;
- DA "IN ARRIVO" AD "ARRIVATO" E VICEVERSA: avviene su invocazione del metodo Signal_Arrival, quando il concorrente arriva (passando da "in arrivo" ad "arrivato") o del metodo Signal_Leaving, quando il concorrente ha finito ed è pronto a lasciare il checkpoint (passando da "arrivato" a "in arrivo").

Quando il **TaskCompetitor** segnala il suo arrivo (a partire dalla chiamata **1**), viene segnato l'arrivo effettivo all'interno del checkpoint. Successivamente viene controllata la posizione del concorrente in coda (chiamate dalla **6** alla **9**). Se il concorrente è primo, è necessario segnalarlo al rispettivo **Waiting_Block**, anche se il thread non si è ancora messo in "wait" (giusto per aprire la guardia). Successivamente il thread del **TaskCompetitor** attende il suo momento con Wait_Ready. Se era già primo, il metodo ritorna subito, dando la possibilità al concorrente di ricevere la lista di path (sicuro che nessun altro in quel momento la starà valutando). Altrimenti il thread viene messo in attesa nel **Waiting_Block** relativo al concorrente. Il thread viene risvegliato nel momento in cui un altro **TaskCompetitor** modifica il suo istante di arrivo limite (facendo in modo che il concorrente in attesa venga a trovarsi nella prima posizione della coda) o quando viene rimosso dal checkpoint. Eseguendo una qualunque delle azioni accennate nella premessa, il **TaskCompetitor** verifica quindi la presenza di un concorrente effettivamente arrivato in prima posizione nella lista ed effettua una Notify sul relativo **Waiting_Block** (nel diagramma viene espresso come un evento esterno al punto **18**). In questo modo il concorrente in attesa può procedere alla valutazione.

4.2.2 Consumatori esterni - Stats

La sottocomponente **Competition_Monitor** espone diverse viste riguardanti i dati relativi alla competizione. Tali dati vengono acceduti in modo "simultaneo" da qualunque entità esterna lo richieda. Viene creato quindi un thread per ogni richiesta proveniente da un nodo distribuito (i *TVScreen* per esempio, di cui ne potrebbero esistere potenzialmente decine o più). La tecnica adottata per affrontare il problema ha alla base l'idea che tutte le statistiche vengono calcolate a partire da dati "grezzi" relativi ai singoli concorrenti. Essendo ogni dato riferito ad un istante e ad una specifica posizione nelle competizione (lap numero 2 e checkpoint 5 per esempio), è possibile calcolare qualunque statistica che sia riferita ad un istante o ad una posizione (es: i tempi di ogni concorrente alla fine della lap 4).
Ad ogni concorrente è assegnato un array (in *Stats*) in cui ogni posizione è destinata a contenere un aggiornamento. Gli aggiornamenti sono cronologicamente ordinati. È quindi possibile richiedere la posizione del concorrente nel circuito all'istante t semplicemente scorrendo l'array. Ogni informazione è contenuta in una risorsa protetta la cui lettura viene aperta solo nel momento in cui l'aggiornamento relativo viene inserito. Quindi se viene richiesta un'informazione non ancora presente, il thread richiedente semplicemente rimarrà in attesa.
Volendo ottenere lo stato di gara all'istante t sarà quindi suggiciente ciclare sui concorrenti chiedendo la loro posizione in tale istante, e quando tutti i concorrenti avranno aggiunto l'aggiornamento richiesto sarà possibile avere uno snapshot della gara in tale istante.
Per la classifica è stata aggiunta una struttura dati di supporto finalizzata a venire popolata con i dati di fine lap di ogni concorrente (**Synch_Classification_Table**).

4.2.3 Risorse a due consumatori

Ci sono infine numerosi casi in cui una risorsa protetta è acceduta dai classici consumatore e un produttore. Accesso simultaneo di due risorse attive al massimo dunque. È il caso, ad esempio, delle informazioni relative al box, **Synch_All_Info_Buffer**. Tale risorsa viene valorizzata da **Strategy_Updater** e letta dallo screen relativo al box (tramite **Box_Monitor**). Come visto nella sezione 4.1.6, il consumatore accede alla risorsa per indice tramite Get_Info. Quindi, se l'informazione di indice richiesto non è ancora stata inserita, la sentinella Ready verrà impostata a False e il thread **riaccodato** in Wait con la guardia Ready=True. Quando il produttore inserisce una nuova risorsa, verrà impostato Ready a True, di modo che l'eventuale thread in attesa possa verificare se l'informazione voluta sia stata insrita. In caso affermativo l'informazione viene ritornata. In caso negativo il thread riesegue la stessa procedure descritta

inizialmente. Chiaramente questa tecnica non potrebbe funzionare con più consumatori, poichè la guardia potrebbe venire cambiata più volte ad insaputa di un thread che stia venendo **riaccodato** da Get_Info a Wait (la **requeue** infatti prevede prerilascio). La soluzione però si presta a risolvere il problema per tutte le situazioni in cui non più di un consumatore e un produttore accedono alla stessa risorsa.

4.2.4 Conclusioni

Quanto descritto riguarda i casi più problematici di gestione dell'accesso concorrente a risorse. Il resto dei casi viene trattato con semplici risorse protette e **entry** con guardie. Non si presenta quindi il rischio di racecondition. Nemmeno starvation dovrebbe essere possibile, a meno che per qualche motivo un thread relativo ad un **TaskCompetitor** non si interrompa inaspettatamente, bloccando il processo di aggiornamento delle statistiche e quindi la produzione di snapshot di gara.

4.3 Distribuzione

Nella seguente sezione verranno spiegati dettagli fondamentali riguardanti la distribuzione nel sistema.

4.3.1 Componenti distribuite

Si è deciso di permettere la distribuzione delle seguenti componenti:

- ***Box***:
  La componente *Box* è possibile avviarla in un nodo separato rispetto alla competizione. È necessario infatti fornire il **corbaloc** della competizione per poter inizializzare tale componente. La scelta è stata suggerita da due motivazioni. Una, puramente pratica, riguarda un potenziale problema di sovraccarico computazionale. Il *Box* infatti, anche se per ora adotta una *AI* con vincoli abbastanza rilassati, potrebbe richiedere una potenza di calcolo superiore per effettuare dei calcoli estremamente complessi. Si pensi per esempio se venisse adottato un sistema ad apprendimento automatico o un algoritmo di ricerca della soluzione ottima ad albero per ogni scelta che il *Box* deve prendere. Avere la possibilità di poterlo isolare in una macchina dedicata potrebbe significare un notevole miglioramento nelle prestazioni.
  La seconda motivazione invece è legata alla realtà: i box normalmente comunicano via radio con i propri piloti. Inserire lo strato di comunicazione remota fra *Competitor* e *Box* è sembrato quindi un buon inizio per dare

maggiore credibilità alla simulazione. I problemi di comunicazione fra le radio di box e pilota si possono tradurre in problemi di rete fra i due nodi ospitanti *Box* e *Competitor*.

- ***TVScreen***:
  Le TV da cui poter osservare il procedere della gara possono essere inizializzate ovunque, a patto di essere in possesso del **corbaloc** del monitor di competizione. Il fatto che le TV (come nella realtà) potrebbero essere numerose, è parso sensato pensarle eseguite in nodi differenti da quello della competizione. Anche in questo caso per evitare il sovraccarico computazionale della macchina ospitante la competizione. Come per i box, se la TV avviata è una e a solo output testuale (come quella presentata nel progetto) il problema non si presenterebbe comunque. Ma volendone avviare molte a volendo magari in un futuro estendere le funzionalità del sistema aggiungendo un overlay grafico più interessante alle TV, tornerebbe molto utile in termini di efficienza poterle eseguire in nodi dedicati.

4.3.2 Interazione fra le componenti distribuite

Le componenti comunicano la maggior parte dei dati appoggiandosi a protocolli XML. Ogni set di informazioni scambiato fra componenti ha il suo schema dedicato. In questo modo si può avere un formato di rappresentazione dei dati standard, più facilmente manipolabile con le librerie e le conoscenze disponibili. Dati che richiedono una precisione maggiore, ovvero gli istanti temporali espressi in FLOAT (che possono crescere notevolmente all'aumentare del tempo di gara), vengono trasmessi utilizzando il tipo FLOAT fornito dal linguaggio IDL e, per sequenze di lunghezza non conosciuta a priori, utilizzando il tipo IDL definito come segue

```
typedef sequence<float> float_sequence;
```

Tale tipo viene poi manipolato in modo diverso in base al linguaggio utilizzato per la componente, rimanendo comunque entro un buon margine di precisione.

4.3.3 Misure di fault tolerance

Cavi di rete danneggiati o router intasati potrebbero provocare il malfunzionamento della comunicazione fra le componenti distribuite. Nel caso di malfunzionamento della comunicazione fra **TVScreen** e *Competition*, la competizione non viene compromessa. Nella peggiore delle ipotesi il **TVScreeb** non è in grado di ottenere i dati richiesti e ritenterà in seguito. Nel caso uno o più dei nodi ospitanti i *Box* cada o rimanga isolato, la competizione rischierebbe di essere

compromessa. In prossimità del checkpoint prima dei box infatti il concorrente effettua una richiesta remota al *Box* per l'aggiornamento della strategia. Se tale richiesta produce errore, il concorrente richierebbe di non poter procedere la gara. Si è deciso quindi di gestire l'arrivo di dati corrotti o la scomparsa di connessione semplicemente facendo mantenere al concorrente la stessa strategia adottata al giro precedente.

4.4 Inizializzazione competizione

Il seguente diagramma di sequenza spiega come avviene la sequenza di init della competizione, a partire dal main della competizione **main_competition.adb**. È necessario premettere che l'init vero e proprio non avviene eseguendo il file **main_competition.adb**. È stato infatti creato uno script (in bash) che effettua l'avvio di tale main e che successivamente inizializza l'interfaccia di configurazione (in java) della competizione.

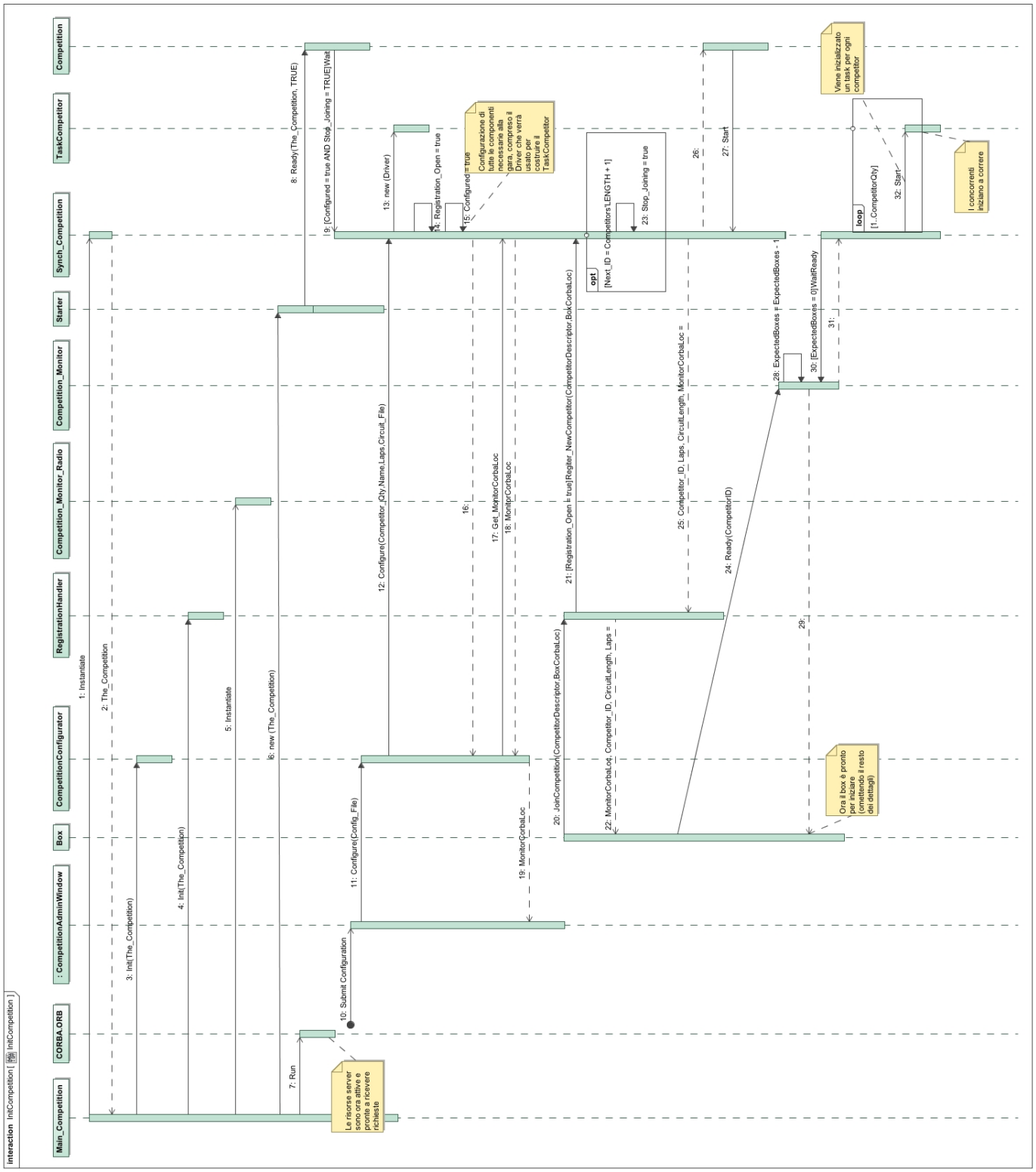

Figura 23: Sequence Diagram - Inizializzazione competizione

Una volta avviato il main della competizione, per prima cosa vengono inizializzati tutti i server, vale a dire **CompetitionConfigurator**, **RegistrationHandler Competition_Monitor_Radio**. Viene avviato un task **Starter** che rimane in attesa delli configurazione e registrazione concorrenti avvenuti per dare lo Start alla gara. Viene inoltre allocato una risorsa protetta di tipo **Synch_Competition** pensata per agevolare la fase di init. Ne viene condivisa un'istanza fra **CompetitionConfigurator**, **RegistrationHandler** e **Starter**. Tale risorsa permette di regolare i vari step dell'inizializzazione. Finchè la competizione non viene configurata (dal passo **10** al passo **16**), non sarà concesso di registrare concorrenti. Una volta configurata la competizione, viene aperta la guardia REGISTRATION_OPEN e il metodo Register_NewCompetitor potrà essere invocato (in **Synch_Competition**. L'iscrizione dei concorrenti avviene a partire dai *Box*. Nel diagramma l'azione iniziale del *Box* e la **20**. In realtà, nel nodo del box, viene prima avviato uno script che esegue il main del box seguito dal main dell'interfaccia (java) di configurazione del concorrente e del box. A configurazione avvenuta, i parametri vengono inviati al server della competizione invocando il metodo del server **Registration_Handler** Join_Competition, come scritto nel diagramma. A iscrizione avvenuta, i parametri di configurazione della competizione tornati vengono usati per inizializzare e avviare il box e relativi task grazie a cui successivamente sarà possibile mandare il messaggio di Ready alla competizione. Per ulteriori dettagli su questa parte, fare riferimento al capitolo 4.1.6 relativo al *Box*.
Quando tutti i concorrenti sono stati registrati (**23**), bisogna rimanere in attesa della conferma di avvenuta inizializzazione dei *Box*. Ciò avviene quando il thread del task **Starter** invoca il metodo Start del **Synch_Competition**, dentro cui viene messo in attesa sul **Competition_Monitor** tramite Wait_Ready. Quando tutti i *Box* hanno dato l'ok, significa che sono pronti per ricevere richieste dai rispettivi competitor e la gara può cominciare. Vengono così avviati uno ad uno (**32**) e la competizione ha inizio.

4.5 Stop competizione

Lo stop del sistema avviene su 3 livelli: stop dei task, stop delle interfacce utente e stop dei server. Vediamo in dettaglio:

Stop dei task:
i task che devono essere fermati sono i **TaskCompetitor** lato competizione e i task **Update_Retriever** e **Strategy_Updater** lato box. I primi si fermano in automatico quando le condizioni dell'auto non permettono di procedere (benzina finita o gomme troppo usurate) oppure a competizione finita (fine ultima lap). I secondi due si fermano quando

l'aggiornamento fornito dal monitor della competizione segnala uno stato dell'auto a causa di cui il concorrente non può procedere (poca benzina o molta usura gomme), oppure quando il concorrente finisce l'ultima lap (sempre grazie agli aggiornamenti);

Stop delle interfacce utente:
è sufficiente chiudere le interfacce con il tasto **x** in alto a destra;

Stop dei server:
una volta finita la gara e chiuse le interfacce utente, è sufficiente digitare "q+INVIO" nelle console dove si sono avviati i main per lanciare un segnale di terminazione ai programmi (e conseguentemente alle risorse server).

5 Analisi del supporto tecnologico

5.1 Scelta dei linguaggi

Per il progetto sono stati utilizzati principalmente due linguaggi (+1 in piccola parte):

- **ADA**: il linguaggio è stato utilizzato per sviluppare i layer sottostanti allo strato di presentazione. Si è valutato infatti, dopo le tematiche affrontate durante il corso, che sarebbe stato più agevole sviluppare un sistema multi-threaded e caratterizzato da forti vincoli di affidabilità e determinismo utilizzando tale linguaggio. Il forte sistema di tipi e le primitive a supporto della gestione dei thread e delle risorse condivise si sono infatti rivelati decisivi avere un maggior controllo sul sistema in fase di sviluppo già a compile-time. Più precisamente, il linguaggio stesso ci avrebbe (e ci ha) vincolato a scrivere del codice logicamente corretto senza troppe iterazioni di debug.

- **Java**: per lo sviluppo delle interfacce grafiche è stato invece utilizzato Java. La scelta è stata dettata da vari motivi. Primo fra tutti la competenza. È sembrato più intuitivo poter gestire un argomento come l'interfaccia utente usando tale linguaggio. Non risultando infatti critiche a livello di logica e occupando il 15% dell'intero progetto, si è pensato che per le interfacce sarebbe stato più che sufficiente. Anche per la competenza già acquisita di librerie come *awt* e *swing*.
  Seconda motivazione, la portabilità. La parte logica del sistema è stata concepita per girare su una macchina con il supporto di un numero di nodi massimo pari alla quantità di concorrenti. La TV per la visualizzazione

della gara potrebbe invece essere potenzialmente avviata su 1000 macchine diverse. È parsa quindi una scelta abbastanza furba offrire la possibilità di utilizzare tale interfaccia in qualunque computer appoggiandosi alla JVM invece che doverla ricompilare e magari riadattare per ogni macchina.

- **Bash**: questo linguaggio di scripting è stato usato per la configurazione e l'avvio del sistema.

5.2 Distribuzione e interoperabilità fra linguaggi

Glossario Bibliografia