
F1 Simulator

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

1	Progetto	4
2	Problematiche	4
2.1	Introduzione	4
2.2	Problematiche di concorrenza	4
2.2.1	Percorrenza concorrente della pista	4
2.3	Problematiche di distribuzione	6
3	Architettura ad alto livello	6
3.1	Componenti di sistema	6
3.1.1	Competition	6
3.1.2	Competitor	7
3.1.3	Circuit	8
3.1.4	Stats	9
3.1.5	Box	10
3.1.6	Monitor	11
3.1.7	Screen	11
3.1.8	Configurator	11
3.2	Interazione fra le componenti	11
3.2.1	Configurator-Competition	11
3.2.2	Competition-Competitor	12
3.2.3	Competition-Monitor	13
3.2.4	Competition-Circuit	13
3.2.5	Competition-Stats	14
3.2.6	Competitor-Stats	14
3.2.7	Competitor-Circuit	15
3.2.8	Monitor-Stats	15
3.2.9	Configurator-Box	16
3.2.10	Box-Monitor	16
3.2.11	Screen-Monitor	17
3.2.12	Competitor-Box	18
3.2.13	Strategia di simulazione	18
3.2.14	Assenza di stallo	19
4	Architettura in dettaglio	20
4.1	Diagrammi delle classi	20
4.1.1	Competition	20
4.1.2	Competitor	22
4.1.3	Circuit	23

4.1.4	Stats	27
4.1.5	Monitor	32
4.1.6	Box	34
4.1.7	Configurator	37
4.1.8	Screen	39
4.2	Analisi della concorrenza	39
4.2.1	Interazione Competitor - Circuit	39
4.2.2	Consumatori esterni - Stats	42
4.2.3	Risorse a due consumatori	42
4.2.4	Conclusioni	43
4.3	Distribuzione	43
4.3.1	Componenti distribuite	43
4.3.2	Interazione fra le componenti distribuite	44
4.3.3	Misure di fault tolerance	44
4.4	Inizializzazione competizione	45
4.5	Stop competizione	45

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 delle 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 molteplicità 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 ottenere Tr_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 una 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 aggirato 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

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.

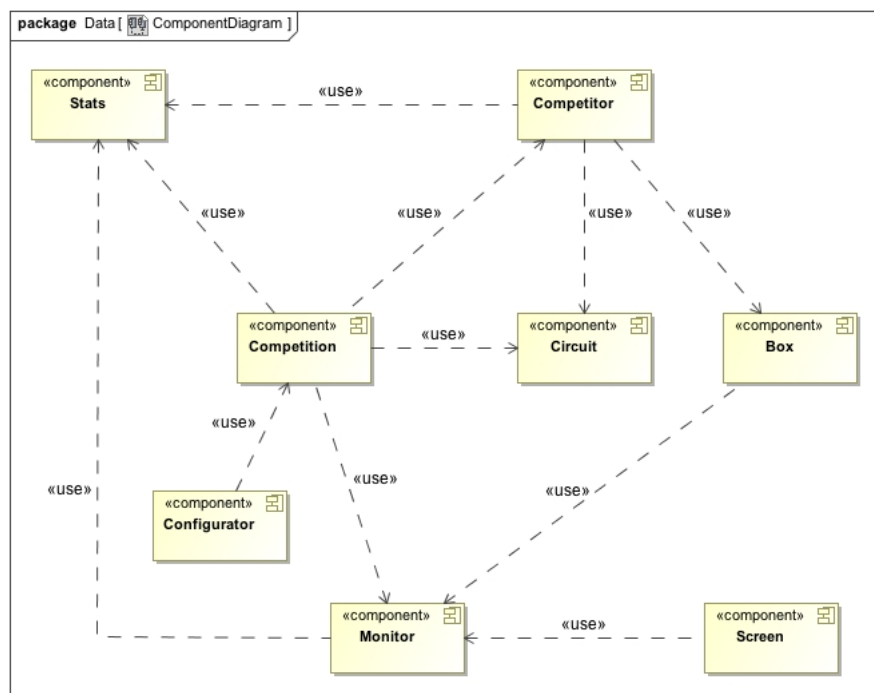


Figura 1: Diagramma delle componenti

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 condizioni dell'auto non permettano di poter correre ulteriormente;
- banalmente, continuare a correre fino alla fine delle lap prestabilite, dopodiché 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 più 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 progire 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* è 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 interazioni principali fra le componenti.

3.2.1 Configurator-Competition

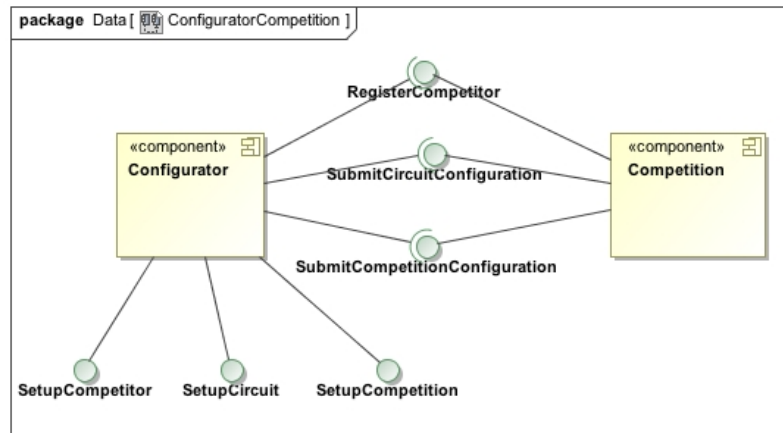


Figura 2: Interface component diagram - Configurator/Competition

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

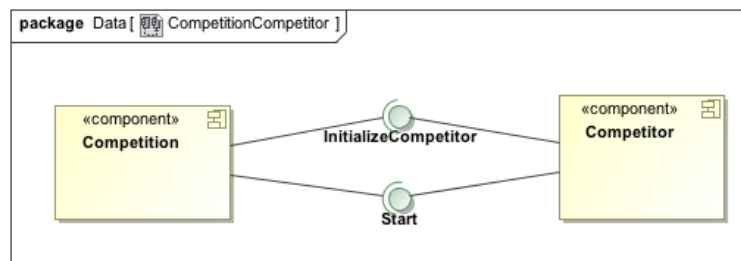


Figura 3: Interface component diagram - Competition/Competitor

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 inizializzare il competitor. Quando tutti

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

3.2.5 Competition-Stats

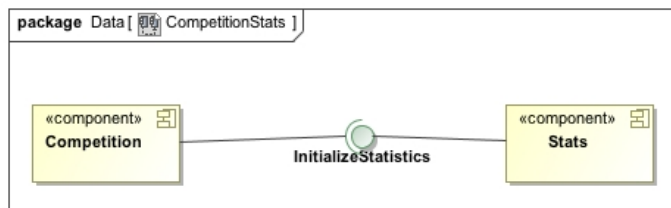


Figura 6: Interface component diagram - Competition/Stats

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

3.2.6 Competitor-Stats

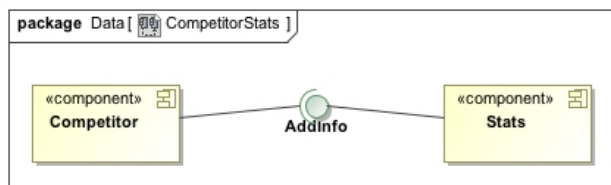


Figura 7: Interface component diagram - Competitor/Stats

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.

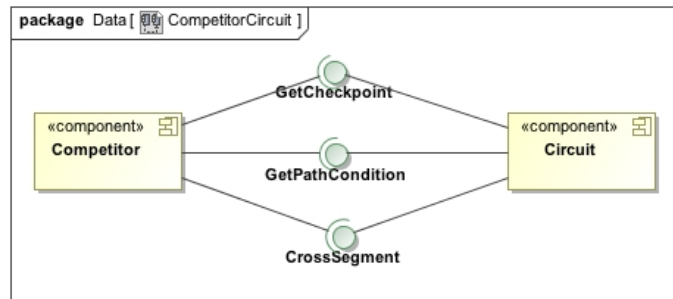


Figura 8: Interface component diagram - Competitor/Circuit

3.2.7 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

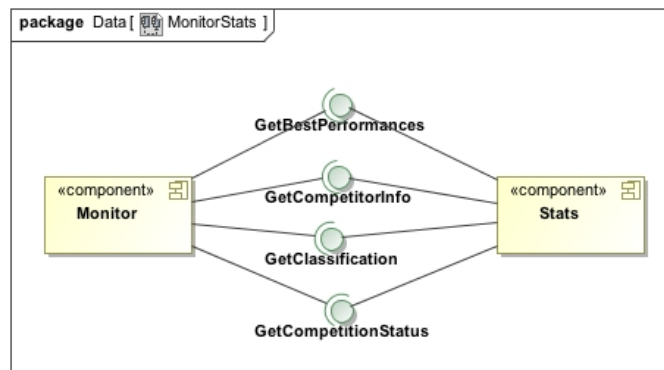


Figura 9: Interface component diagram - Monitor/Stats

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.

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 circuito in un determinato istante di tempo.

3.2.9 Configurator-Box

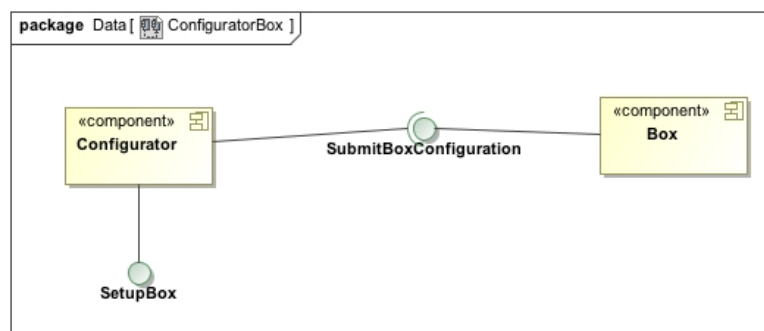


Figura 10: Interface component diagram - Configurator/Box

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

La prima interazione che la componente *Box* ha con il *Monitor* è in fase di inizializzazione della gara. La sequenza di azioni verrà esplicitata dettagliatamente in seguito, per ora basti sapere che il *Box*, una volta configurato è pronto per

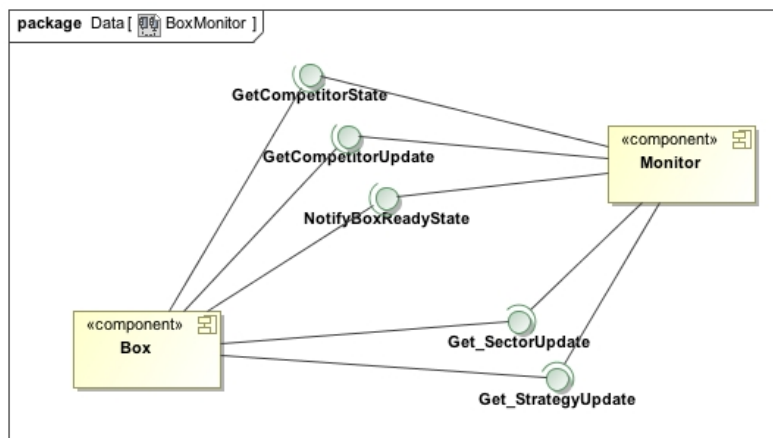


Figura 11: Interface component diagram - Box/Monitor

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

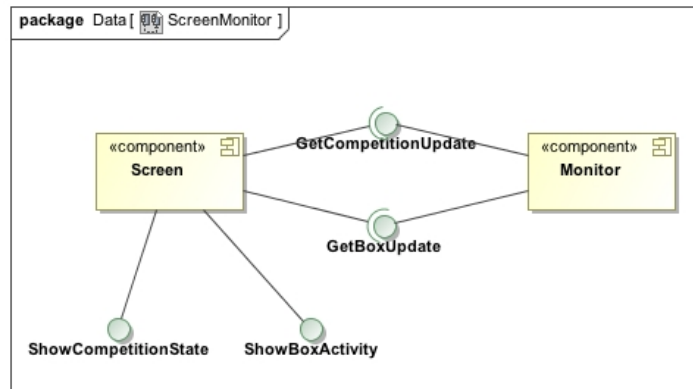


Figura 12: Interface component diagram - Screen/Monitor

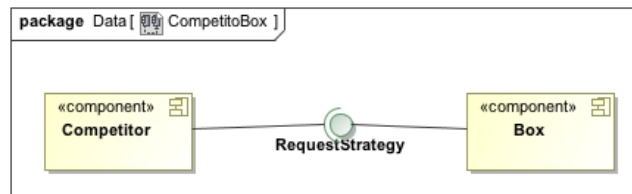


Figura 13: Interface component diagram - Competitor/Box

3.2.12 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
 - **istante di arrivo:**
 - **istante limite di arrivo previsto:**
 - **istante di liberazione traiettoria:**

3.2.14 Assenza di stallo

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

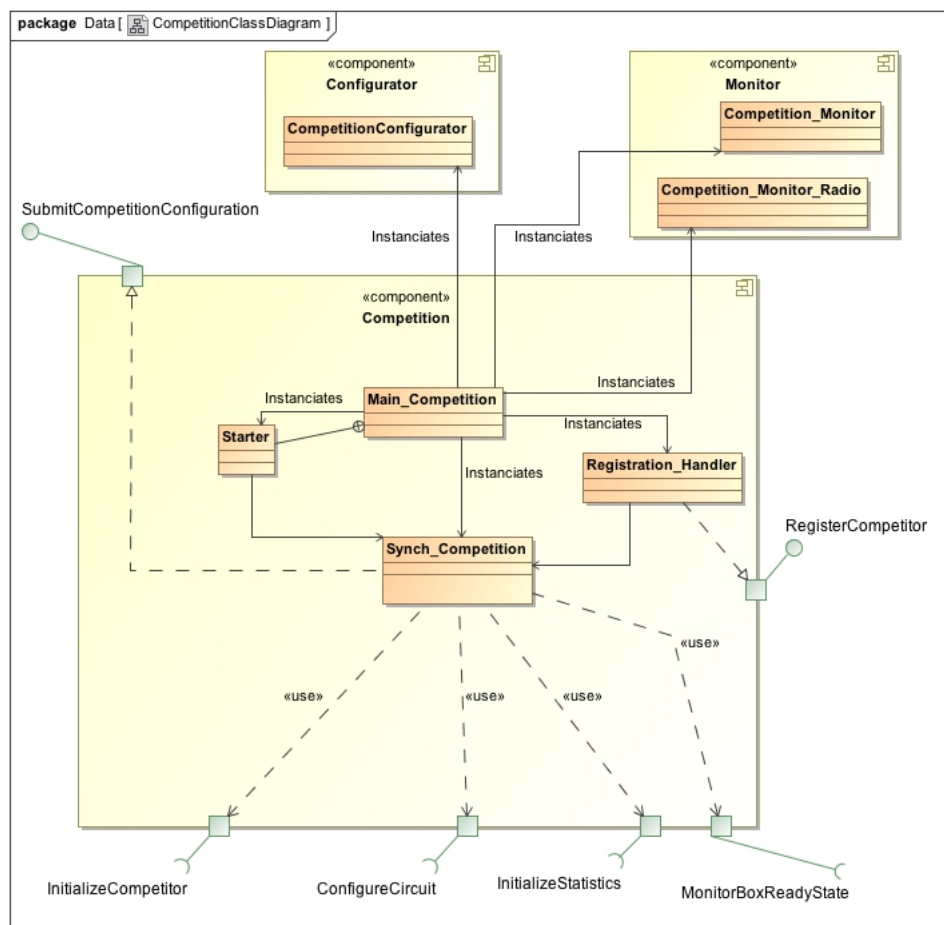


Figura 14: Class diagram - Competition

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 l'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.

4.1.3 Circuit

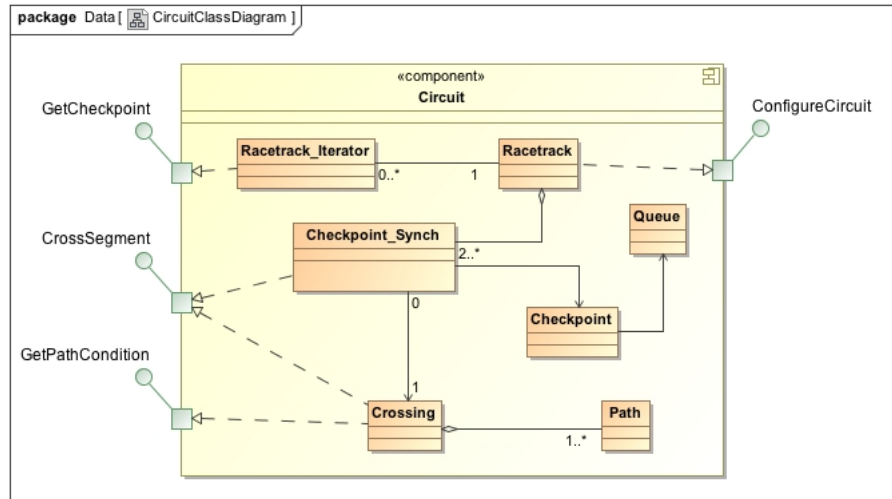


Figura 16: Class diagram - Circuit

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 vuoti dalla corsia dei box

Dati questi parametri, vengono inizializzati 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 ammissibile 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 dispone della sua copia dell'**Racetrack.Iterator**.

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 necessario 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 true, 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 più 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 $N^{\circ} \text{ Lap} * N^{\circ} \text{ Checkpoint}$, 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; Lapped-Competitors_ID : out INTEGER_ARRAY_POINT; Lapped-Competitor_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

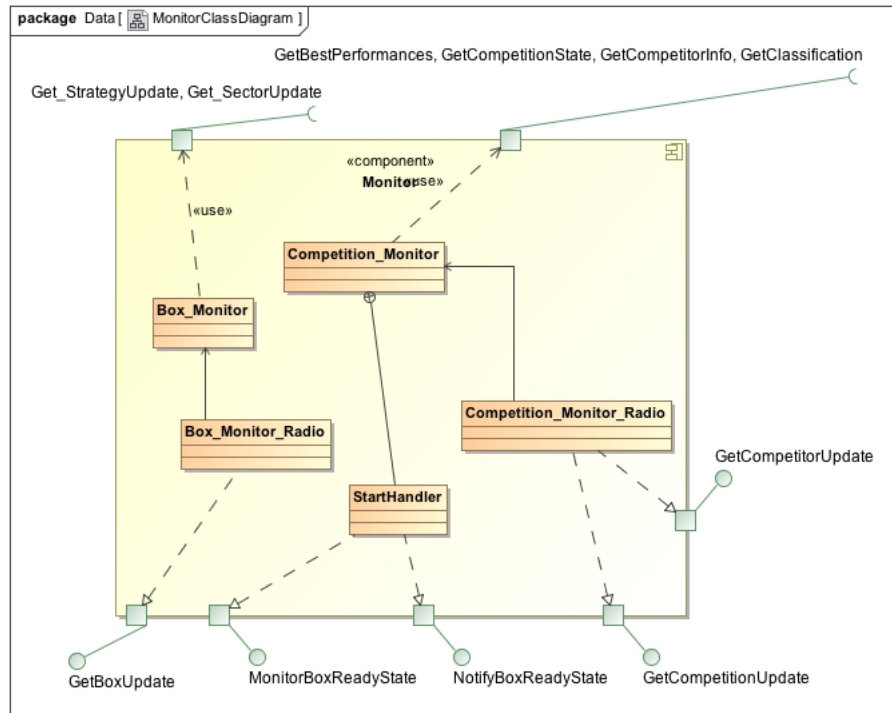


Figura 18: Class diagram - Monitor

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; Classifica-
tionTimes : 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 informazioni 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 la comunicazione distribuita tramite Polyorb (usando il protocollo CORBA).

4.1.6 Box

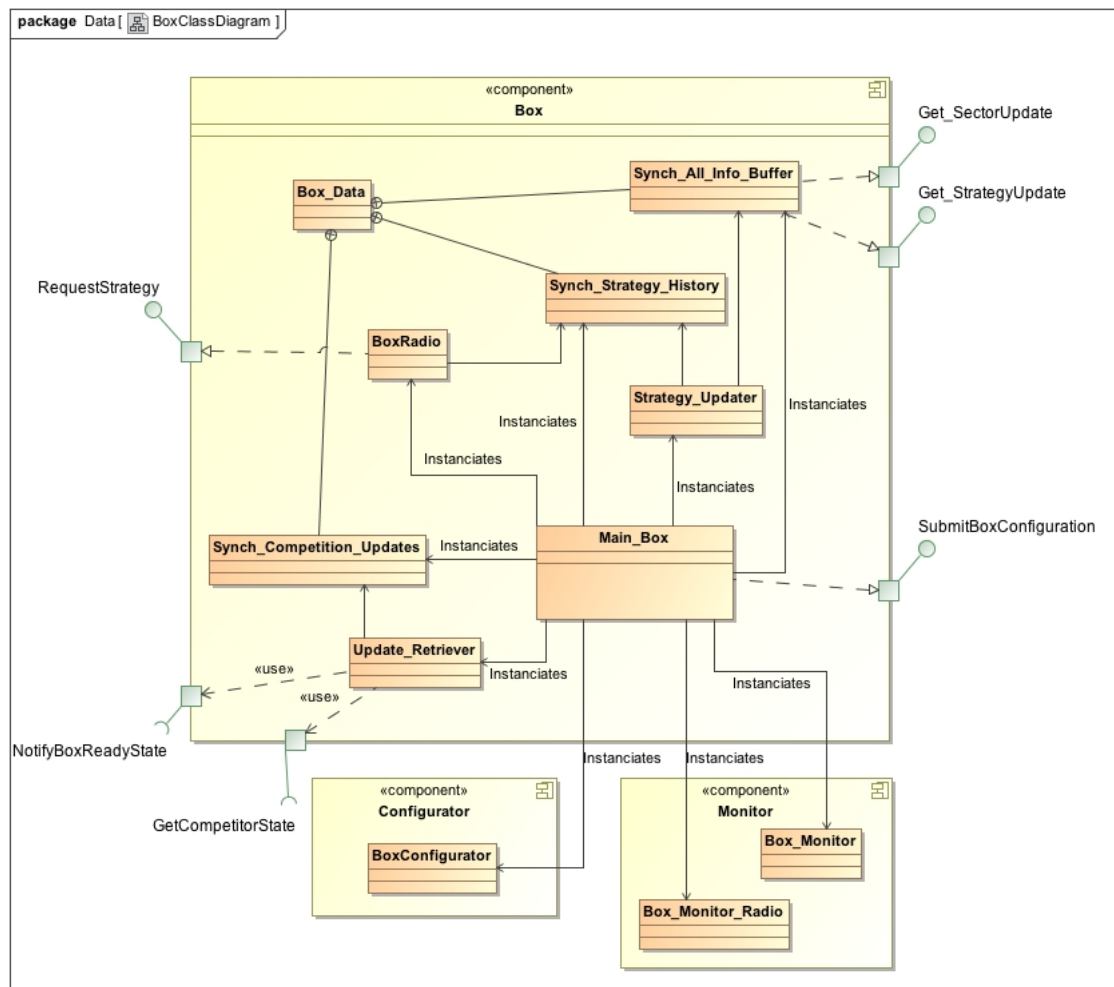


Figura 19: Class diagram - Box

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

Update_Retrieve:

Come il nome suggerisce, questa unit'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 ci'ò, gli aggiornamenti vengono inseriti nel buffer **Synch.Competition_Updates**. Il task **Strategy_Updater** potr' a cos' i 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 ' e ancora stata prodotta, il thread verr' a messo in attesa.

Strategy_Updated:

L' entit' a 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 ' e 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 entit' a 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 ' e 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 ' e presente, verr' a 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 ' e destinata 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 $3 * LapTotali$, poich'è 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 unit'è 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

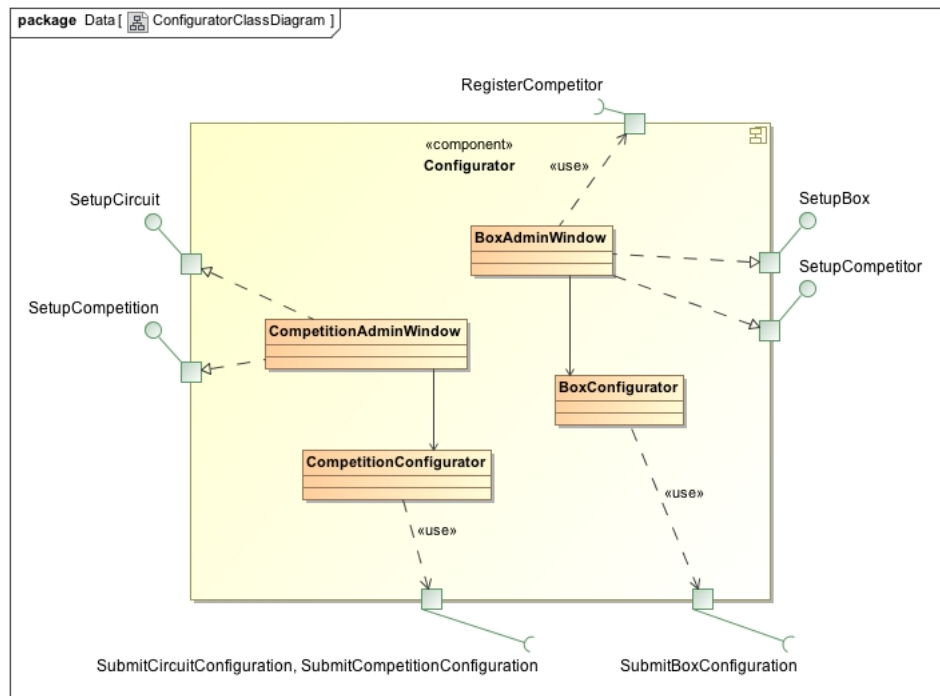


Figura 20: Class diagram - Configurator

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 quantit' 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

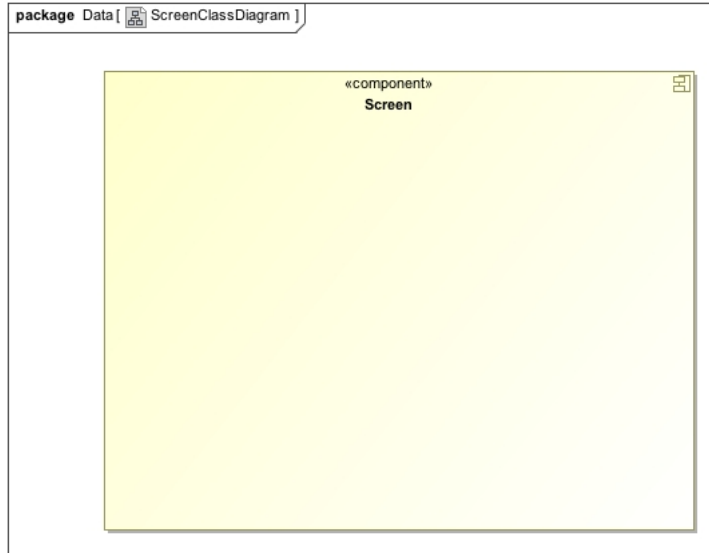


Figura 21: Class diagram - Screen

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 mutuamente 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.

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 VICEVARSA: 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 nella 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 sufficiente 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 **riaccolato** 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 inserita. In caso affermativo l’informazione viene ritornata. In caso negativo il thread riesegue la stessa procedura 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 **riaccolato** 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

maggior 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 iniziate 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 **TVScreen** 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**. E' 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.

4.5 Stop competizione

Analisi del supporto tecnologico Glossario Bibliografia

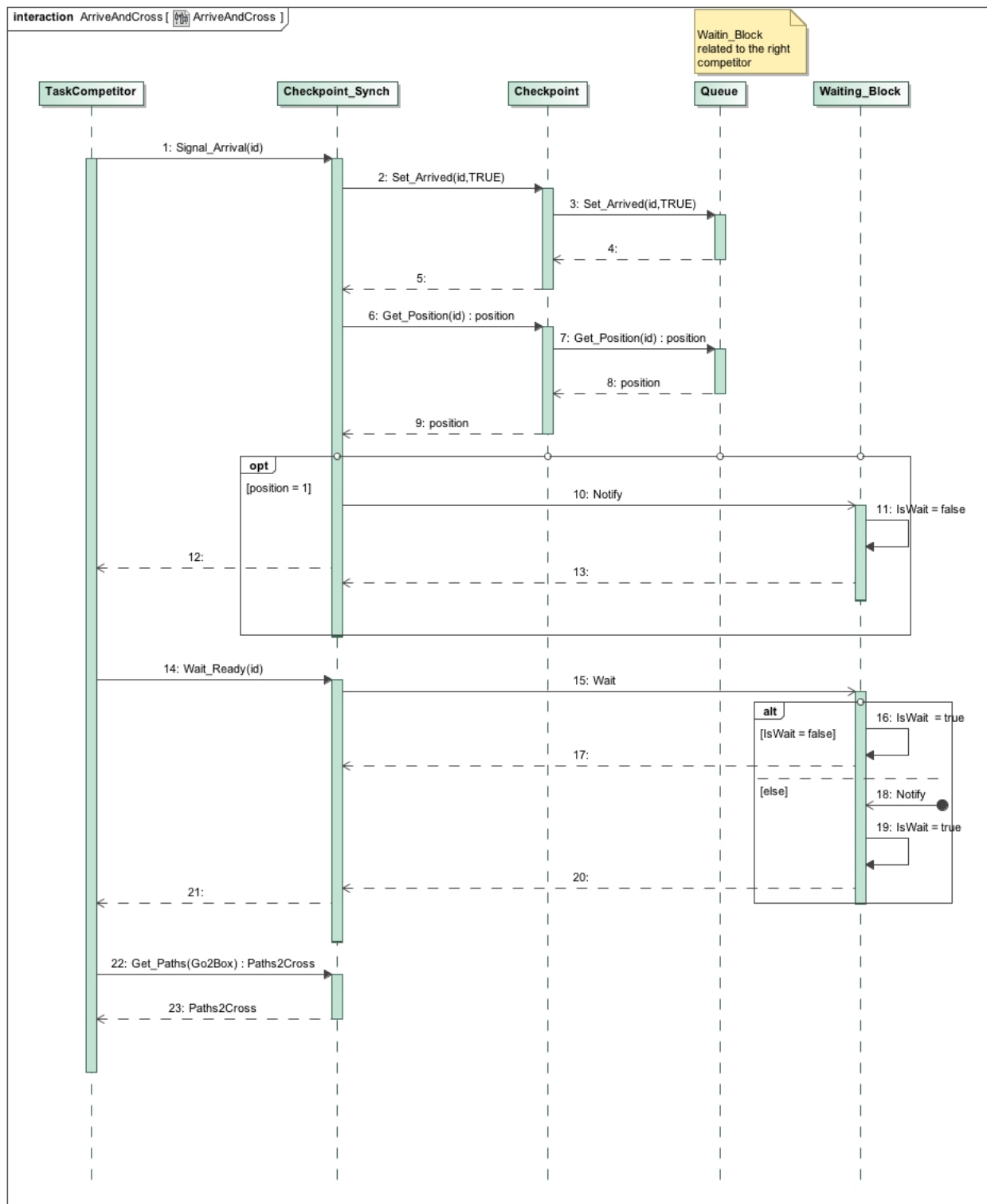


Figura 22: Sequence Diagram - Arrivo al checkpoint e recupero lista traiettorie

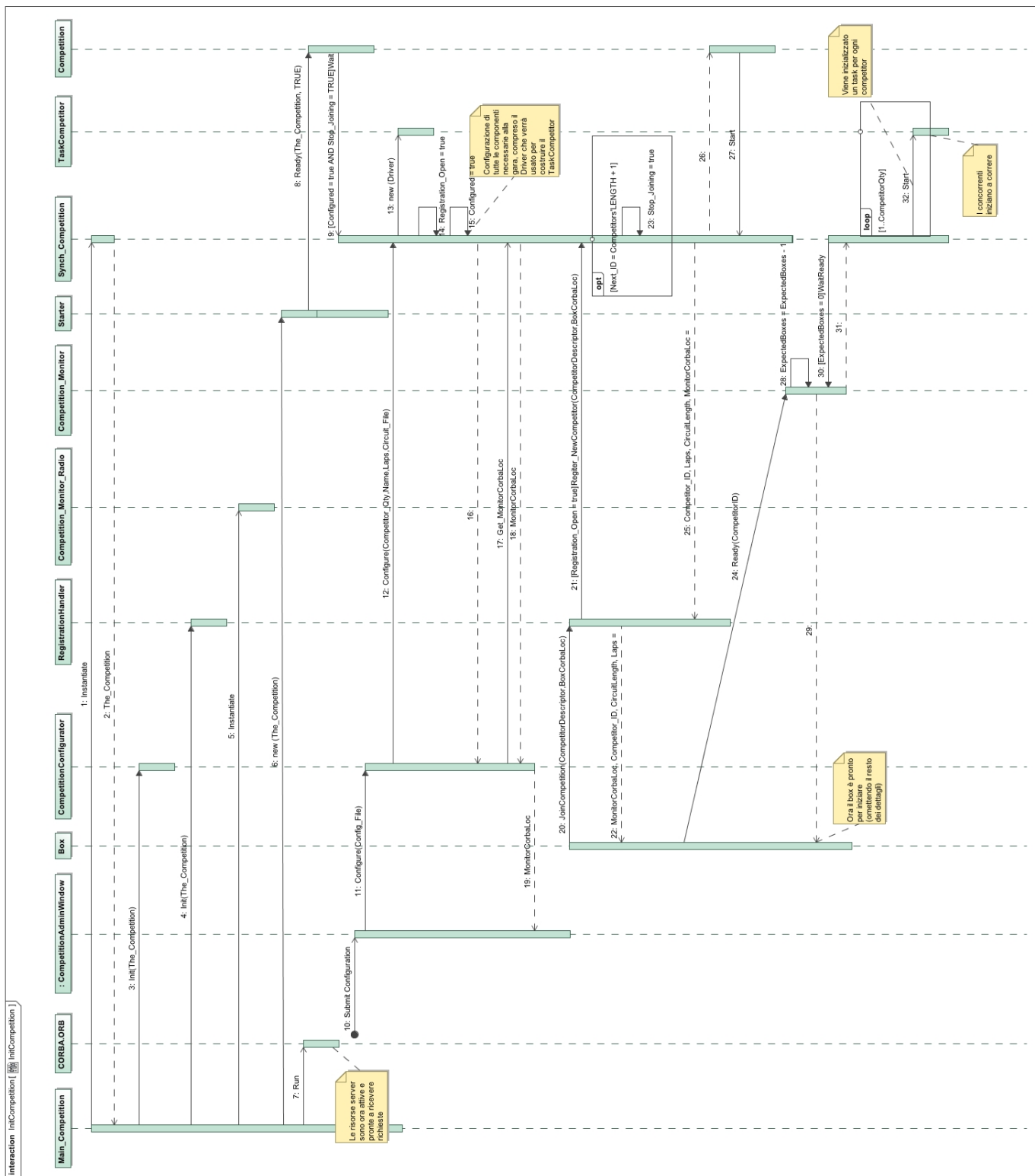


Figura 23: Sequence Diagram - Inizializzazione competizione