Simulatore di Gestione della Memoria di un Elaboratore SiGeM

http://stylosoft.altervista.org
stylosoft@gmail.com

Specifica tecnica	14 Febbraio 2008
Documento Esterno - Formale - v3.0	Specifica_tecnica.pdf

Redazione:

Alberto Zatton

Revisione:

Giordano Cariani

Approvazione:

Rubin Luca

Lista di distribuzione:

Prof. Vardanega Tullio
Prof. Palazzi Claudio
Stylosoft

Registro delle modifiche:

Versione	Data	Descrizione delle modifiche
3.0	14/02/2008	Aggiunta descrizione specifica e diagrammi
2.1	10/02/2008	Aggiunto paragrafo Design Pattern
2.0	26/01/2008	Evidenziati i termini del glossario
1.0	25/01/2008	Aggiornati i diagrammi, le descrizioni e aggiunto tracciamento
0.6	24/01/2008	Scritte le prime descrizioni dei package
0.4	24/01/2008	Aggiornate le stime di fattibilità
0.2	23/01/2008	Prima Stesura

http://stylosoft.altervista.org
stylosoft@gmail.com

Sommario

Questo documento ha il fine di illustrare le scelte tecniche, tecnologiche ed architetturali per la realizzazione del sistema SiGeM.

Indice

1 Introduzione

1.1 Scopo del documento

Tale documento analizza lo scenario applicativo e definisce l'architettura del progetto ad un alto livello di astrazione, finalizzandone gli aspetti tecnici, operativi ed implementativi affinché si possa passare ad una progettazione di dettaglio e quindi finire alla programmazione vera e propria.

1.2 Scopo del prodotto

Lo scopo del prodotto SiGeM è quello di fornire un sistema didattico destinato allo studio dei meccanismi di gestione della memoria in un elaboratore *multiprogrammato*. Tale software consentirà all'utilizzatore di simulare il comportamento di vari algoritmi di rimpiazzo delle *pagine* e *segmenti*, sulla base di dati da lui specificati, ai fini di illustrare le problematiche inerenti alla gestione della memoria. Per una descrizione più dettagliata rimandiamo al documento [AR].

1.3 Glossario

I termini in *corsivo* sono descritti per una migliore comprensione in [G], allegato a questo documento.

1.4 Riferimenti

- [AR] Analisi_dei_requisiti.pdf
- [SF] Studio_di_fattibilita.pdf
- [G] Glossario.pdf
- [PP] Piano_di_progetto.pdf
- [NP] Norme_di_progetto.pdf

1.5 Relazione con altri progetti

Al fine di ottenere una *simulazione* più completa possibile, oltre alla gestione della memoria si è scelto di includere anche la *simulazione* della schedulazione dei processi. Per riuscire a contenere il tempo di sviluppo, si è deciso di utilizzare algoritmi già sviluppati dal gruppo BlueThoth, realizzatori del progetto “SGPEMv2”. Questa scelta è stata influenzata dalla buona realizzazione di tali algoritmi, oltre al tipo di linguaggio usato (*Java*), compatibile con il nostro progetto.

2 Definizione del prodotto

2.1 Metodo e formalismo di specifica

Per rendere più comprensibile l'architettura da noi realizzata per descrivere il progetto, utilizzeremo diagrammi delle classi in linguaggio *UML*. Di questi diagrammi si fornirà poi una descrizione in dettaglio.

Questo documento è stato redatto durante al fase di analisi (rif. [PP]), pertanto presenterà un livello di astrazione alto. Successivamente, durante la fase di progettazione, verrà analizzato ogni singolo componente, specificando con più rigore il contenuto delle classi e le relazioni tra le classi stesse.

2.2 Design Pattern

2.2.1 Façade Pattern

Abbiamo deciso di utilizzare il pattern Façade perché ci permette di avere un'interfaccia diretta per la creazione della simulazione, invece di avere più interfacce complesse. Nello specifico, la classe rappresentante il Façade è il Processore. Questi conosce le classi specializzate nei vari compiti e tramite un'unica chiamata al metodo creaSimulazione() crea la simulazione senza necessità per il client (nel nostro caso la classe Player) di conoscere le classi adibite ai singoli compiti nella creazione della simulazione. Le classi del sottosistema sono lo Scheduler, adibito alla gestione dei processi e il GestoreMemoria, adibito alle operazioni sulla memoria, sia RAM che Swap. Queste due classi implementano le funzionalità del sottosistema e, come da definizione del pattern, non hanno conoscenza del processore, cioè non ne hanno alcun riferimento.

2.2.2 Iterator Pattern

Abbiamo utilizzato abbondantemente questo pattern per due motivi principali. Il primo perché è estremamente semplice da integrare e ne esiste già una implementazione efficiente nelle librerie standard di Java, il secondo motivo è che ci permette di navigare all'interno la moltitudine di strutture dati con estrema comodità, disaccoppiandoci totalmente dalle diverse implementazioni delle strutture dati. Questo pattern è stato utilizzato nelle seguenti classi logiche: GestoreMemoriaPaginata, GestoreMemoriaSegmentata, Player.

2.2.3 Strategy Pattern

Sebbene sia un pattern particolare questo ci ha permesso in più situazioni di astrarre il problema in maniera efficiente. In particolare è stato utilizzato in due occasioni. Lo Scheduler che mantiene un riferimento alla PoliticaOrdinamentoProcessi, interfaccia successivamente implementata delle classi concrete che si vogliono implementare con un algoritmo differente di schedulazione. Il gestore della memoria, allo stesso modo utilizza un'interfaccia strategica che sarà implementata dalle classi concrete per organizzare le politiche di allocazione dei segmenti e rimpiazzo delle pagine.

2.3 Primo livello di decomposizione architetturale

Di seguito viene riportato il diagramma delle classi realizzato in fase di analisi, con relativa illustrazione dei vari componenti. I componenti disegnati in rosso saranno quelli sviluppati ex-novo dal nostro gruppo, mentre quelli disegnati in giallo saranno sviluppati a partire da quelli progettati dal gruppo BlueThoth (vedi 1.5).

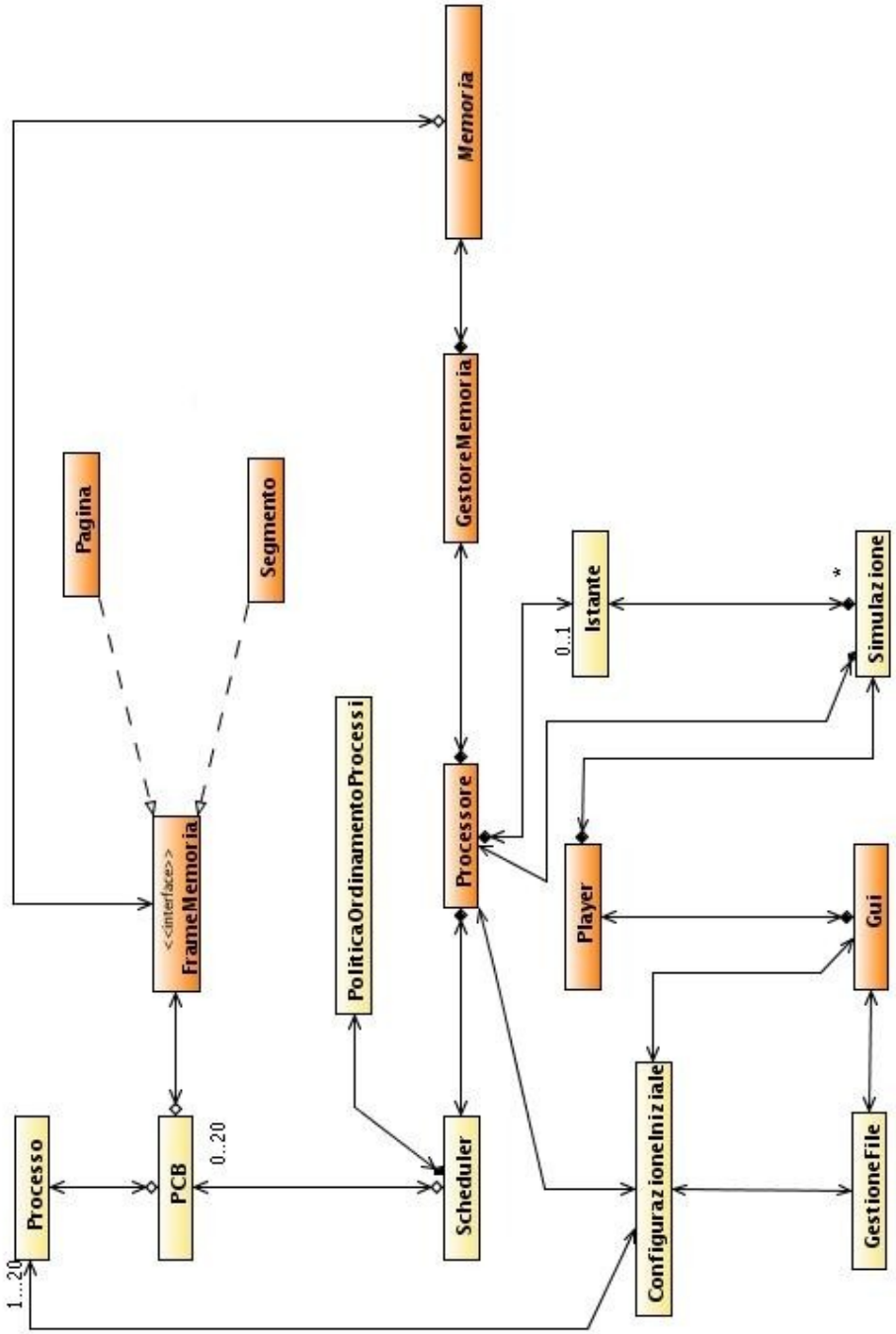

Memoria
GestoreMemoria
Pagina
Segmento
Istante
Simulazione
0..1
*
<<interface>>
FrameMemoria
PoliticaOrdinamentoProcessi
Processore
Player
Gui
Processo
PCB
0..20
Scheduler
ConfigurazioneIniziale
GestioneFile
1...20

Figura 2.1

Il cuore centrale del sistema è formata dalla coppia **Scheduler**-**GestoreMemoria**, coordinata dalla classe **Processore**: è compito dello **Scheduler** infatti scegliere il *processo* da mandare in esecuzione, in accordo con la politica di schedulazione dei processi scelta, mentre **GestoreMemoria** si occupa della gestione delle *pagine/segmenti* di cui ha bisogno il *processo*. Compito del **Processore** è infine unire le informazioni ottenute da **Scheduler** e **GestoreMemoria** per costruire un **Istante** della *simulazione*. Per **Istante** si intende una singola unità di tempo nell'evolvere della *simulazione*: deve perciò contenere tutte le informazioni che serviranno poi per calcolare le statistiche e lo stato del sistema.

Per quanto riguarda il meccanismo di schedulazione dei processi, ci siamo ispirati al lavoro effettuato in precedenza dal gruppo BlueThoth come specificato nel punto 1.5. Le classi principali coinvolte nella schedulazione sono: **Processo**, **PCB** e **PoliticaOrdinamentoProcesso**. Un'istanza di **Processo** contiene tutte le informazioni riguardanti il *processo* al momento della sua creazione. **PCB** (*Process Control Block*) ha lo scopo di salvare, per ogni *processo*, i suoi parametri durante l'evoluzione della *simulazione*. **PoliticaOrdinamentoProcesso** sarà poi realizzata come interfaccia: ogni sua implementazione rappresenterà una delle politiche di schedulazione dei *processi* specificate in [AR], paragrafo 5, tabella A7.

La gestione della memoria rappresenta l'obiettivo centrale del nostro progetto. Le classi coinvolte sono: **RAM**, **SWAP**, **PoliticaOrdinamentoMemoria**. **RAM** rappresenta la memoria principale usata dai *processi* da simulare. **SWAP** invece vuole simulare la memoria virtuale su disco fisso. Entrambe sono concretizzazioni della classe astratta **Memoria**, che ne contiene i tratti comuni. Inoltre entrambe sono costituite da un insieme di implementazioni concrete dell'interfaccia **FrameMemoria**, che sarà illustrata in seguito. **PoliticaRimpiazzoMemoria** è un'interfaccia che sarà implementata dalle diverse politiche di rimpiazzo delle pagine, illustrate in [AR] par. 5 tabella A3, o dalle diverse politiche di rimpiazzo dei segmenti, illustrate in [AR] par. 5 tabella A4. Saranno le diverse implementazioni di questa interfaccia che si occuperanno di modificare lo stato di **RAM** e **SWAP**.

Per rappresentare le diverse modalità di organizzare la memoria, usiamo un'interfaccia **FrameMemoria**. L'implementazione concreta di questa interfaccia sarà operata dalle classi **Pagina**, **Segmento** e **SegmentoPaginato**. In tutte queste classi, come attributo indispensabile, sarà presente un identificativo univoco per distinguere le varie locazioni di memoria le une dalle altre.

Simulazione si preoccuperà di collezionare e salvare l'evoluzione della simulazione. Questo sarà realizzato tramite una collezione ordinata di istanze della classe **Istante**.

Player rappresenta la classe incaricata di scorrere la collezione ordinata di istanze della classe **Istante** contenuta in **Simulazione**. Dovrà offrire tutte le funzionalità previste in [AR], 3.3.3.

ConfigurazioneIniziale è la classe che racchiude al suo interno tutti i parametri che l'utente, all'inizio della *simulazione*, inserirà per configurare l'ambiente. I parametri sono illustrati in [AR], par. 5, tabelle A1-A2-A3-A4-A7. Questi dati potranno essere salvati e caricati da file, tramite la classe **GestioneFile**.

Gui, come suggerisce il nome, rappresenta la parte grafica del progetto. Deve permettere all'utente di interfacciarsi con la parte logica, offrendo tutte le funzionalità di quest'ultima. Dovrà essere il più possibile user-friendly, in modo da permettere una rapida configurazione della *simulazione* soprattutto pensando all'utilizzo didattico durante le lezioni. Inoltre sarà adibita a mostrare a video l'evoluzione della *simulazione* e le statistiche, in maniera esaustiva e immediata.

3 Descrizione package

Di seguito viene rappresentato il diagramma delle classi raffigurante la suddivisione in package delle classi rappresentate in Figura 2.1. Questa organizzazione permette una suddivisione logica dei vari componenti del sistema.

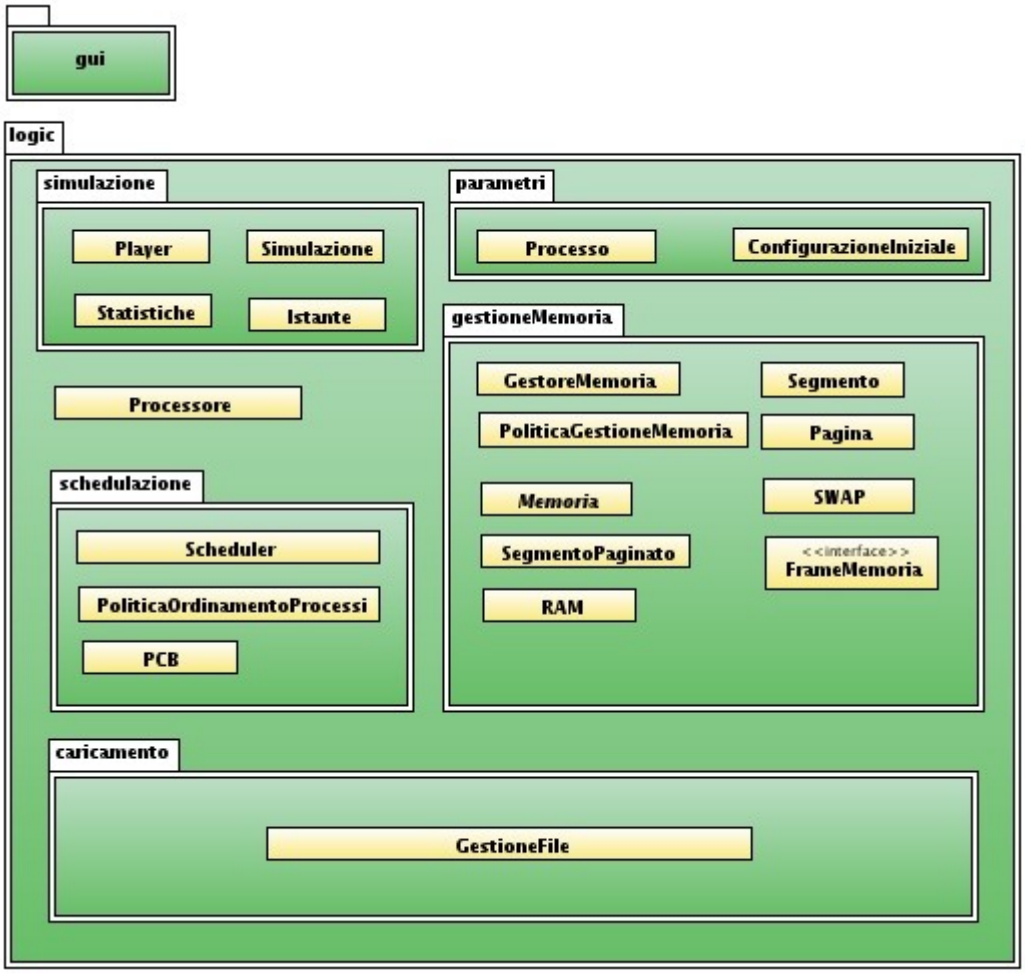

gui
logic
simulazione
Player
Simulazione
Statistiche
Istante
parametri
Processo
ConfigurazioneIniziale
gestioneMemoria
GestoreMemoria
Segmento
PoliticaGestioneMemoria
Pagina
Memoria
SWAP
SegmentoPaginato
<<interface>>
FrameMemoria
RAM
Processore
schedulazione
Scheduler
PoliticaOrdinamentoProcessi
PCB
caricamento
GestioneFile

Figura 3.1

4 Descrizione dei singoli componenti

1.1 Specifica parte logica

1.1.1 logic

1.1.1.1 Processore

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  E' la classe centrale incaricata di creare la simulazione. E' anche il façade che coordina l'attività di Scheduler e GestoreMemoria.

- **Relazioni d'uso con altre componenti:**
  Utilizza i metodi fineSimulazione, getProcessiInArrivo, eseguiAttivazione, getPCBCorrente, eseguiIncremento e getProcessiTerminati della classe Scheduler.
  Utilizza i metodi esegui e notificaProcessoTerminato della classe GestoreMemoria.
  Utilizza il metodo getRifProcesso di PCB e getId di Processo.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dalla classe Simulazione che invoca il Processore per essere creata. In quanto façade non è riferita da altre classi.

  - **Metodi:**
    - public LinkedList<Istante> creaSimulazione()
      Metodo principale della classe Processore, incaricato di creare tutta la simulazione, rappresentata da una collezione di istanti.

    - private int calcolaFault(LinkedList<Azione> istruzioni)
      Metodo che si occupa di estrarre il numero di fault di pagina da una lista di istruzioni sulla memoria.

    - private boolean controllaSwapPiena(LinkedList<Azione> istruzioni)
      Metodo il cui compito è ritornare tramite un booleano se l'area di Swap è piena o meno.

    - private boolean controllaRAMPiena(LinkedList<Azione> istruzioni)

Metodo il cui compito è ritornare tramite un booleano se la RAM è piena o meno.

- private Istante creaIstante(PCB corrente, LinkedList<Azione> istruzioni, boolean nuovoProcesso, boolean SwapPiena)

  Metodo con il compito di creare una istanza della classe Istante riguardante l'istante corrente della simulazione.

- private LinkedList<FrameMemoria> estraiFrame(PCB corrente)

  Metodo che ha il compito di estrarre i FrameMemoria necessari al processo correntemente in esecuzione nell'istante corrente della simulazione.

1.1.2 logic.caricamento

1.1.2.1 GestioneFile

- **Tipo, obiettivo e funzione della classe/interfaccia:**

  Classe pubblica e concreta; si interfaccia con i file per il funzionamento del sistema di simulazione.

- **Relazioni d'uso con altre componenti:**

  Nessuna.

- **Interfacce e relazioni di uso da altre componenti:**

  Viene usata dall'interfaccia grafica per salvare e caricare configurazioni da file specificandone il percorso.

- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public String getPercorsoFileConfigurazione():
      ritorna il percorso del file in cui verrà salvata/caricata la configurazione.

    - public void setPercorsoFileConfigurazione(String nuovoPercorso):
      imposta il percorso del file di configurazione.

    - public boolean salvaFileConfigurazione (ConfigurazioneIniziale conf) throws IOException:

salva la configurazione passata come parametro nel file.

- public ConfigurazioneIniziale caricaFileConfigurazione() throws IOException, ClassNotFoundException:
  carica la configurazione da file e la ritorna al chiamante.

1.1.3 logic.gestioneMemoria

1.1.3.1 GestoreMemoria

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe astratta che rappresenta la struttura generica del sistema di gestione della memoria. Riunisce le caratteristiche comuni ed essenziali per tutte le possibili classi ereditate. Non implementa alcun metodo direttamente, poiché le operazioni che dovrà compiere e le strutture dati che dovrà manipolare sono logicamente discrepanti. L'obbiettivo di questa classe è fornire una lista di azioni che rappresenta la sequenza logica delle operazioni che effettuo in memoria.
- **Relazioni d'uso con altre componenti:**
  poiché non ha campi dati propri ne metodi implementati, essa non è collegata direttamente alcuna componente.

- **Interfacce e relazioni di uso da altre componenti:**
  Questa classe, o le istanze delle sue sottoclassi, sono utilizzate soltanto dalla classe Processore.

- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public abstract void notificaProcessoTerminato(int id):
      Metodo astratto che notifica, tramite l'identificativo del processo, quando esso è terminato, per liberare la memoria che esso utilizzava.

    - public abstract LinkedList<Azione> esegui(LinkedList<FrameMemoria> ListaFrame, int UT):
      Metodo astratto che esegue il calcolo delle operazioni in memoria da effettuare affinché il processo pronto possa eseguire.

1.1.3.2 *GestoreMemoriaPaginata*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe concreta che eredita direttamente dalla classe GestoreMemoria e ne implementa i metodi astratti. Si occupa interamente ed esclusivamente della gestione della memoria paginata, ossia quando l'utente richiede una simulazione di un sistema a memoria virtuale paginata. Questa classe fornisce al Processore una lista di azioni, che descrive precisamente tutte le operazioni che dovranno esser eseguite in memoria nel corso dell'esecuzione della simulazione.
- **Relazioni d'uso con altre componenti:**
  Utilizza tramite l'interfaccia IRimpiazzo, un'istanza della politica di rimpiazzo delle pagine decisa in fase di configurazione del sistema.
  Utilizza inoltre un'istanza della classe RAMPaginata e una della classe SwapPaginata tramite due riferimenti. Legge informazioni dalla classe ConfigurazioneIniziale. Utilizza strutture dati standard dalla libreria “java.util” in particolare LinkedList, Vector e Iterator. Costruisce istanze di AzionePagina.

- **Interfacce e relazioni di uso da altre componenti:**
  Questa classe, una sua istanza per volta, è utilizzata soltanto dalla classe Processore, tramite il metodo esegui.

- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public abstract void notificaProcessoTerminato(int id):
      Metodo concreto che notifica, tramite l'identificativo del processo, quando esso è terminato, per liberare le pagine che esso utilizzava.

    - public abstract LinkedList<Azione> esegui(LinkedList<FrameMemoria> ListaFrame, int UT):
      Metodo pubblico concreto che esegue le operazioni sulle strutture di memoria. In particolare prende come ingressi una lista di pagine le quali dovranno esser caricate in memoria RAM nonché l'unità di tempo di esecuzione. Questo metodo costruirà e ritornerà alla classe Processore la lista delle azioni che ha compiuto per portare la memoria allo stato attuale.
    - private int inserisci(MemoriaPaginata M, FrameMemoria F, int UT) throws MemoriaEsaurita:

Metodo privato che inserisce una pagina F nella memoria M, nell'istante UT. Questo metodo potrà sollevare l'eccezione MemoriaEsaurita quando l'inserimento avviene nella memoria di Swap ed essa sarà piena. L'eccezione verrà quindi gestita dal chiamante cioè il metodo esegui. Questo metodo ritorna la posizione di inserimento.

- private FrameMemoria rimuovi(Memoria M, FrameMemoria F):
  Metodo privato che rimuove una pagina F dalla memoria M, e ritorna contemporaneamente la pagina rimossa, che sarà diversa da F nel caso in cui dovrò rimpiazzare una pagina dalla RAM.

1.1.3.3 GestoreMemoriaSegmentata

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica e concreta. Questa classe eredita direttamente dalla classe GestoreMemoria e quindi ne implementa i metodi astratti. Manipola la memoria secondo la tecnica della segmentazione. Restituisce le operazioni che ad ogni istante della simulazione dovranno essere eseguite nel sistema a segmentazione affinchè ogni processo possa eseguire.

- **Relazioni d'uso con altre componenti:**
  Utilizza tramite l'interfaccia IAllocazione, un'istanza della politica di allocazione dei segmenti decisa in fase di configurazione del sistema.
  Utilizza inoltre un'istanza della classe RAMSegmentata e una della classe SwapSegmentata tramite due riferimenti. Legge informazioni dalla classe ConfigurazioneIniziale. Utilizza strutture dati standard dalla libreria “java.util” in particolare LinkedList, Vector e Iterator. Costruisce istanze di AzioneSegmento.

- **Interfacce e relazioni di uso da altre componenti:**
  E' utilizzata dalla classe Processore tramite il metodo pubblico “esegui”.

- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public void notificaProcessoTerminato(int id):
      Metodo pubblico chiamato dal Processore per notificare la terminazione di un processo. Chiama la cancellazione dalla memoria di tutte le risorse che il processo utilizzava tramite il metodo liberaMemoria(id) di Memoria. Si richiede l'identificativo del processo terminato.

- private FrameMemoria RimuoviFrame(Vector<FrameMemoria> Frames):
  Metodo privato di utilità interna alla classe che rimuove un segmento dalla RAM quando essa è piena per far posto ad un altro segmento. La rimozione avviene secondo una politica FIFO. Il parametro Frames dovrà contenere la lista di Segmenti presenti in memoria. Il metodo ritornerà un riferimento al Segmento rimosso.
- private FrameMemoria Inserisci(MemoriaSegmentata M, FrameMemoria F) throws MemoriaEsaurita:
  Metodo privato di utilità interna alla classe che inserisce il Segmento F nella memoria M. Può rilanciare l'eccezione MemoriaEsaurita nel caso in cui l'aggiunta del segmento in memoria Swap preveda il superamento della quantità di memoria Swap disponibile. L'inserimento in RAM avviene secondo la politica di allocazione configurata. Il metodo ritorna il riferimento allo spazio di memoria in cui si è inserito il segmento.
- private FrameMemoria Rimuovi(MemoriaSegmentata M, FrameMemoria F):
  Metodo privato di utilità interna alla classe che rimuove il segmento F dalla memoria M nel caso essa sia Swap, in caso contrario richiama il metodo RimuoviFrame. Ritorna il frame effettivamente rimosso.
- public LinkedList<Azione> esegui(LinkedList<FrameMemoria> ListaSegmenti, int UT):
  Metodo pubblico di fondamentale importanza. E' il metodo principale per tutto il sistema. E' chiamato dal Processore passandogli la lista di segmenti da caricare in quell'istante. Restituisce una LinkedList di AzioneSegmento che rappresenta la sequenza ordinata di azioni effettuate nella memoria.

1.1.3.4 Azione

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica, astratta. Questa classe rappresenta un'azione generica effettuata in memoria.
- **Relazioni d'uso con altre componenti:**
  Mantiene soltanto un riferimento generico a FrameMemoria coinvolto nell'azione.
- **Interfacce e relazioni di uso da altre componenti**:
  E' utilizzata indirettamente dal GestoreMemoria per il fatto che è classe astratta e quindi non istanziabile.
- **Attività svolte e dati trattati:**

- **Metodi:**
  - public int getAzione():
    Ritorna il contenuto del campo dati TipoAzione.
  - public FrameMemoria getFrame():
    Ritorna il riferimento al FrameMemoria.

1.1.3.5 AzionePagina

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica, concreta. Questa classe eredita da Azione generica effettuata in memoria e ne specializza l'uso per una memoria paginata.
- **Relazioni d'uso con altre componenti:**
  Nessuna oltre a quelle della classe padre.
- **Interfacce e relazioni di uso da altre componenti**:
  E' utilizzata direttamente ed esclusivamente dal GestoreMemoriaPaginata.

1.1.3.6 AzioneSegmento

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica, concreta. Questa classe eredita da Azione generica effettuata in memoria e ne specializza l'uso per una memoria segmentata.
- **Relazioni d'uso con altre componenti:**
  Nessuna oltre a quelle della classe padre.
- **Interfacce e relazioni di uso da altre componenti**:
  E' utilizzata direttamente ed esclusivamente dal GestoreMemoriaSegmentata.

1.1.3.7 IAllocazione

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Interfaccia pubblica che definisce la struttura comune a tutte le politiche di allocazione dei segmenti.
- **Relazioni d'uso con altre componenti:**
  Nessuna.
- **Interfacce e relazioni di uso da altre componenti**:
  E' utilizzata esclusivamente dal GestoreMemoriaSegmentata.
- **Attività svolte e dati trattati:**

  - **Metodi:**

- public FrameMemoria Alloca (FrameMemoria F, Vector<FrameMemoria> Liberi):
  Sarà implementato da ogni politica di allocazione. Il metodo dovrà restituire il FrameMemoria libero che secondo la politica rispecchia meglio le caratteristiche del frame passatogli F. Liberi è un vettore il quale contiene tutti i FrameMemoria liberi all'interno della memoria RAM.

1.1.3.8 *IRimpiazzo*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Interfaccia pubblica che definisce la struttura comune a tutte le politiche di rimpiazzo delle pagine.
- **Relazioni d'uso con altre componenti:**
  Nessuna.
- **Interfacce e relazioni di uso da altre componenti**:
  E' utilizzata esclusivamente dal GestoreMemoriaPaginata.
- **Attività svolte e dati trattati:**
  - **Metodi:**
    - public void InserisciEntry(FrameMemoria F, int Posizione, int UT, boolean M):
      Implementato dalle politiche di rimpiazzo servirà per aggiornare i dati derivati dall'inserimento in RAM di una pagina.
    - public void LiberaEntry(int Posizione):
      Resetta i dati in fase di rimozione di una pagina.
    - public FrameMemoria SelezionaEntry():
      Seleziona la pagina da rimuovere secondo la politica prescelta.
    - public void AggiornaEntry(int Posizione, boolean M):
      Aggiorna i dati dopo un accesso alla pagina da parte di un processo.
    - public void AggiornaEntries( ):
      Aggiorna i dati di tutte le pagine in memoria quando il gestore della memoria lo richiede ai fini del corretto funzionamento degli algoritmi di rimpiazzo.

1.1.3.9 *FirstFit,BestFit,NextFit,QuickFit,WorstFit*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classi pubbliche concrete che implementano l'interfaccia IAllocazione. In particolare implementano il metodo Alloca secondo la politica dalla quale

prende il nome la classe, rispettivamente:
FirstFit: Restituisce un riferimento al primo spazio di memoria disponibile a contenere il segmento.
BestFit: Restituisce un riferimento allo spazio di memoria le cui dimensioni sono il più possibile vicine al segmento da inserire.
NextFit: Come FirstFit soltanto inizia la ricerca dall'ultimo elemento visitato.
QuickFit: Come BestFit salvo diversa implementazione.
WorstFit: Restituisce un riferimento allo spazio di dimensione maggiore.

- **Relazioni d'uso con altre componenti:**
  Utilizzano la classe Vector standard dalla libreria java.util.
- **Interfacce e relazioni di uso da altre componenti**:
  Sono utilizzate esclusivamente dal GestoreMemoriaSegmentata in fase di allocazione dei segmenti. .
- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public FrameMemoria Alloca (FrameMemoria F, Vector<FrameMemoria> Liberi):
      Implementa l'algoritmo di allocazione secondo quanto decritto nell'obbiettivo della classe, restituendo un riferimento allo spazio sul quale effettuare l'allocazione. Come parametri richiede un riferimento F al segmento da inserire e un Vector di riferimenti agli spazi liberi in memoria.

1.1.3.10 RAMPaginata

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe che nel nostro sistema modella la RAM sotto forma di collezione di pagine. E' quindi l'implementazione della RAM che verrà usata quando l'utente sceglie come tecnica di gestione della memoria la paginazione. La classe offre tutti i metodi necessari per l'inserimento, la rimozione e la ricerca delle pagine.

- **Relazioni d'uso con altre componenti:**
  Eredita direttamente da MemoriaPaginata, classe base per il tipo di memorie modellate come collezioni di pagine. Ovviamente usa la classe Pagina e i suoi metodi, anche per gestire le pagine in memoria.

- **Interfacce e relazioni di uso da altre componenti:**
  La classe viene usata esclusivamente dalla classe GestoreMemoriaPaginata.

- **Attività svolte e dati trattati:**

  - Metodi:
    - public int aggiungi(FrameMemoria pag) throws MemoriaEsaurita:
      Metodo che aggiunge una pagina nella RAM. Se ha successo, ritorna un int che rappresenta l'id della "cella" in cui ha inserito la pagina, quindi l'indice del Vector memoria in cui il riferimento è stato inserito; in caso di fallimento solleva un'eccezione che deve essere opportunamente gestita. Richiede un parametro di tipo FrameMemoria, il riferimento alla pagina da aggiungere in RAM.

    - public boolean cerca(FrameMemoria pag):
      Metodo che cerca se una pagina è già presente in RAM: serve per verificare se scatta un page fault o meno. Chiede come parametro il riferimento alla pagina.

    - public int indiceDi(FrameMemoria pag):
      Metodo che, dato un riferimento di tipo FrameMemoria, ritorna un int rappresentante la sua posizione all'interno della RAM. Utile in qualche politica di rimpiazzo delle pagine.

    - public void liberaMemoria(int idProcesso):
      Metodo che marca come eliminabili le pagine riferite da un processo che finito la sua esecuzione. Durante il prossimo inserimento di una pagina, queste pagine saranno considerate memoria libera, quindi si potranno inserire nuove pagine senza interrogare la politica di rimpiazzo. Richiede come parametro un int, l'identificativo univoco del processo che ha finito la sua esecuzione.

    - public boolean rimuovi(FrameMemoria pag):
      Metodo che marca la pagina come non più in RAM, liberando spazio. Questo metodo viene usato subito dopo aver scelto la pagina da rimpiazzare: quest'ultima è quella riferita dal parametro pag.

1.1.3.11 SwapPaginata

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe che nel nostro sistema modella lo swap sotto forma di collezione di pagine. E' quindi l'implementazione dello swap che verrà usata quando

l'utente sceglie come tecnica di gestione della memoria la paginazione. La classe offre tutti i metodi necessari per l'inserimento, la rimozione e la ricerca delle pagine.

- **Relazioni d'uso con altre componenti:**
  Eredita direttamente da MemoriaPaginata, classe base per il tipo di memorie modellate come collezioni di pagine. Ovviamente usa la classe Pagina e i suoi metodi, anche per gestire le pagine in memoria.

- **Interfacce e relazioni di uso da altre componenti:**
  La classe viene usata esclusivamente dalla classe GestoreMemoriaPaginata.

- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public int aggiungi(FrameMemoria pag) throws MemoriaEsaurita:
      Metodo che aggiunge una pagina nello swap. Se la memoria non è piena, aggiunge la pagina; altrimenti lancia un'eccezione e a quel punto il programma termina (se lo swap è pieno, lo è di conseguenza anche la RAM e ciò provoca il termine del programma). Come parametro riceve il riferimento alla pagina da inserire.

    - public void liberaMemoria(int idProcesso):
      Metodo che rimuove dallo swap le pagine riferite da un processo che ha finito la sua esecuzione. Richiede come parametro un int, l'identificativo univoco del processo che ha finito la sua esecuzione.

    - public boolean rimuovi(FrameMemoria pag):
      Metodo che rimuove dallo swap la pagina riferita dal parametro pag.

1.1.3.12 RAMSegmentata

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe che nel nostro sistema modella la RAM sotto forma di collezione di segmenti. E' quindi l'implementazione della RAM che verrà usata quando l'utente sceglie come tecnica di gestione della memoria la segmentazione. La classe offre tutti i metodi necessari per l'inserimento, la rimozione e la ricerca dei segmenti.

- **Relazioni d'uso con altre componenti:**

Eredita direttamente da MemoriaSegmentata, classe base per il tipo di memorie modellate come collezioni di segmenti. Ovviamente usa la classe Segmento e i suoi metodi, anche per gestire i segmenti in memoria.

- **Interfacce e relazioni di uso da altre componenti:**
  La classe viene usata esclusivamente dalla classe GestoreMemoriaSegmentata.

- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public void aggiungi(FrameMemoria seg, FrameMemoria spazio):
      Metodo che aggiunge un segmento nella RAM. Prende come parametri il riferimento al segmento da inserire e il riferimento allo spazio scelto dalla politica di allocazione dei segmenti.

    - public boolean cerca(FrameMemoria seg):
      Metodo che cerca se un segmento è già presente in RAM: serve per verificare se scatta un page fault o meno. Chiede come parametro il riferimento al segmento.

    - public void liberaMemoria(int idProcesso):
      Metodo che elimina i segmenti non più riferiti perchè il relativo processo ha terminato la sua esecuzione. Nel caso i segmenti da eliminare siano contigui a spazi vuoti, viene creato un unico spazio di dimensione uguale alla somma delle grandezze di spazi e segmento. Prende come parametro l'identificativo univoco del processo che ha finito di eseguire.

    - public boolean rimuovi(FrameMemoria seg):
      Metodo che rimuove il segmento riferito da seg. Nel caso il segmento da eliminare sia contiguo a spazi vuoti, viene creato un unico spazio di dimensione uguale alla somma delle grandezze di spazi e segmento.

    - public Vector<FrameMemoria> getFrameLiberi():
      Metodo che costruisce e restituisce un Vector composto dagli spazi vuoti disponibili in RAM. Utile per gli algoritmi di allocazione dei segmenti.

    - public Vector<FrameMemoria> getFrameOccupati():

Metodo che costruisce e restituisce un Vector composto dai segmenti presenti in RAM. Utile per gli algoritmi di rimpiazzo dei segmenti.

- public FrameMemoria getSpazioMaggiore():
  Metodo che restituisce il segmento con lo spazio libero più grande. Utile per gli algoritmi di rimpiazzo dei segmenti.

1.1.3.13 SwapSegmentata

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe che nel nostro sistema modella lo swap sotto forma di collezione di segmenti. E' quindi l'implementazione dello swap che verrà usata quando l'utente sceglie come tecnica di gestione della memoria la segmentazione. La classe offre tutti i metodi necessari per l'inserimento, la rimozione e la ricerca dei segmenti.

- **Relazioni d'uso con altre componenti:**
  Eredita direttamente da MemoriaSegmentata, classe base per il tipo di memorie modellate come collezioni di segmenti. Ovviamente usa la classe Segmento e i suoi metodi, anche per gestire i segmenti in memoria.

- **Interfacce e relazioni di uso da altre componenti:**
  La classe viene usata esclusivamente dalla classe GestoreMemoriaSegmentata.

- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public void aggiungi(FrameMemoria seg) throws MemoriaEsaurita:
      Metodo che aggiunge un segmento nello swap. Se la memoria non è piena, aggiunge il segmento; altrimenti lancia un'eccezione e a quel punto il programma termina (se lo swap è pieno, lo è di conseguenza anche la RAM e ciò provoca il termine del programma). Come parametro riceve il riferimento al segmento da inserire.

    - public void liberaMemoria(int idProcesso):
      Metodo che rimuove dallo swap i segmenti riferiti da un processo che ha finito la sua esecuzione. Richiede come parametro un int, l'identificativo univoco del processo che ha finito la sua esecuzione.

- public boolean rimuovi(FrameMemoria seg):
  Metodo che rimuove dallo swap la pagina riferita dal parametro seg.

1.1.3.14 Memoria

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe astratta che rappresenta una generica memoria: verrà poi implementata da tutte le memorie, sia paginate che segmentate.

- **Relazioni d'uso con altre componenti:**
  La classe contiene al suo interno la struttura dati essenziale per realizzare le memorie, costituita dall'utilizzo della classe java.util.Vector istanziata con FrameMemoria.

- **Interfacce e relazioni di uso da altre componenti:**
  Questa classe è la classe base per tutte le memorie, perciò viene ereditata direttamente sia da MemoriaPaginata sia da MemoriaSegmentata. Indirettamente, viene utilizzata da tutte le successive istanziazioni delle memorie.

- **Attività svolte e dati trattati:**

  - Metodi:
    - public abstract boolean rimuovi(FrameMemoria frame):
      Metodo astratto la cui implementazione dovrà rimuovere un FrameMemoria dalla memoria. Il FrameMemoria da togliere è quello riferito dal parametro richiesto frame. Ritorna TRUE se la rimozione ha successo, FALSE altrimenti.

    - public abstract boolean cerca(FrameMemoria frame):
      Metodo astratto la cui implementazione dovrà cercare un FrameMemoria nella memoria. frame è il riferimento al FrameMemoria da cercare. Ritorna TRUE se il frame è in memoria, FALSE altrimenti.

    - public abstract void liberaMemoria(int idProcesso):
      Metodo astratto la cui implementazione dovrà rimuovere tutti i FrameMemoria riferiti da un processo che ha finito la sua esecuzione. Richiede come parametro un int che è l'identificativo univoco del processo che ha finito di eseguire.

1.1.3.15 *MemoriaPaginata*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe astratta che rappresenta le memorie che utilizzano paginazione. Verrà concretizzata poi dalle classi RAMPaginata, SwapPaginata.

- **Relazioni d'uso con altre componenti:**
  Eredita direttamente da Memoria.

- **Interfacce e relazioni di uso da altre componenti:**
  Questa classe astratta verrà implementata e usata concretamente esclusivamente da RAMPaginata e SwapPaginata.

- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public abstract int aggiungi(FrameMemoria frame) throws MemoriaEsaurita:
      Metodo astratto la cui implementazione nelle sottoclassi dovrà permettere l'inserimento di una pagina in memoria. Richiede come parametro un riferimento alla pagina da inserire. Ritorna un int rappresentante la posizione di inserimento del frame all'interno del Vector memoria. In caso di spazio insufficiente, è previsto il lancio di un'eccezione che dovrà essere opportunamente gestita.

1.1.3.16 *MemoriaSegmentata*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe astratta che rappresenta le memorie che utilizzano segmentazione. Verrà concretizzata poi dalle classi RAMSegmentata, SwapSegmentata.

- **Relazioni d'uso con altre componenti:**
  Eredita direttamente da Memoria.

- **Interfacce e relazioni di uso da altre componenti:**
  Questa classe astratta verrà implementata e usata concretamente esclusivamente da RAMSegmentata e SwapSegmentata.

- **Attività svolte e dati trattati:**

- public abstract void aggiungi(FrameMemoria frame, FrameMemoria spazio) throws MemoriaEsaurita:
  Metodo astratto la cui implementazione nelle sottoclassi dovrà permettere l'aggiunta di un segmento in memoria. Chiede come parametro il riferimento al segmento da inserire. Può lanciare l'eccezione MemoriaEsaurita nel caso in cui non ci sia più spazio per inserire il segmento.

1.1.3.17 *FrameMemoria*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Interfaccia pubblica; racchiude gli aspetti comuni di pagine e segmenti.

- **Relazioni d'uso con altre componenti:**
  Nessuna.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene implementata dalle classi: Pagina e Segmento.
  Viene utilizzata in tutte le classi che compongono e interagiscono con il gestore della memoria.

1.1.3.18 *Pagina*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica e concreta; rappresenta una pagina che può risiedere in memoria. Implementa l'interfaccia FrameMemoria.

- **Relazioni d'uso con altre componenti:**
  Nessuna.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dal gestore della memoria e da tutte le classi che interagiscono con lui.

1.1.3.19 *Segmento*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica e concreta; rappresenta un segmento che può risiedere in memoria. Implementa l'interfaccia FrameMemoria.

- **Relazioni d'uso con altre componenti:**
  Nessuna.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dal gestore della memoria e da tutte le classi che interagiscono con lui.

1.1.4 logic.parametri

1.1.4.1 ConfigurazioneIniziale

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica e concreta. Rappresenta la configurazione del sistema di simulazione. Estende la classe Serializable per permettere la serializzazione in file.

- **Relazioni d'uso con altre componenti:**
  Nessuna.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dalle classi: SwapPaginata, GestoreMemoriaPaginata, Simulazione, Processore, RAMSegmentata, Player, GestioneFile, AssociazioneProcessiJDialog, RAMPaginata, GestoreMemoriaSegmentata, SwapSegmentata.

1.1.4.2 EccezioneConfigurazioneNonValida

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica e concreta; estende la classe Exception. Rappresenta un'eccezione lanciata nel momento in cui viene creata una ConfigurazioneIniziale non valida.

- **Relazioni d'uso con altre componenti:**
  Nessuna.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dalla classe ConfigurazioneIniziale.

1.1.4.3 *Processo*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  La classe Processo rappresenta la classe base per tutti i processi. Essa contiene in sé tutti i campi dati e metodi comuni ad ogni tipo di processo.

- **Relazioni d'uso con altre componenti:**
  Utilizza il metodo getIstanteRichiesta della classe interna Accesso. Utilizza la classe FrameMemoria per impostare le richieste di accesso alla memoria.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata da ConfigurazioneIniziale, da Processore e infine da Scheduler, come lista di processi in arrivo.
  Viene usata da PCB come campo dati.

- **Attività svolte e dati trattati:**
  - **Campi dati:**
    - private java.utils.ArrayLIst<Accesso> accessi:
      Campo dati che contiene la lista degli accessi a FrameMemoria che un processo andrà a richiedere durante il suo ciclo di vita.

    - private int id:
      Campo dati che identifica univocamente ogni istanza della classe. Da inizializzare con un'apposita chiamata ad un metodo della classe Id.

    - private java.lang.String nome:
      Campo dati contenente il nome associato al processo

    - private int tempoArrivo:
      Campo dati contenente l'istante in cui un processo diventa attivo, ciò è l'istante a partire dal quale un processo può concorrere per andare in esecuzione.

    - private int tempoEsecuzione:
      Campo dati che contiene il tempo totale d'esecuzione di un processo.

1.1.4.4 *ProcessoConPriorita*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  La classe ProcessoConPriorita estende la classe Processo e rappresenta i processi che, oltre ai dati comuni ad ogni processo ed ereditati dunque dalla classe Processo, sono caratterizzati dall'avere una certa priorità, la quale non si esclude possa anche cambiare nel corso dell'evoluzione della simulazione.

- **Relazioni d'uso con altre componenti:**
  Utilizza il metodo getIstanteRichiesta della classe interna Accesso. Utilizza la classe FrameMemoria per impostare le richieste di accesso alla memoria.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata da tutte le implementazioni di PoliticaOrdinamentoProcessi con priorità.

1.1.5 logic.simulazione

1.1.5.1 *Simulazione*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica e concreta; crea e avvia un processore che crea tutti gli istanti della simulazione. Tali istanti vengono mantenuti internamente a questa classe.

- **Relazioni d'uso con altre componenti:**
  Utilizza il metodo creaSimulazione() della classe Processore per creare gli istanti della simulazione.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dalla classe Player.

- **Attività svolte e dati trattati:**
  - **Campi dati:**
    - private Processore proc:
      istanza di processore che si occupa della generazione degli istanti.

    - private LinkedList<Istante> listaIstanti:
      lista di istanti generata dal processore.

- private ConfigurazioneIniziale conf:
  parametri di configurazione per il sistema di simulazione.

- **Metodi:**
  - public LinkedList<Istante> crea():
    da il via alla creazione della simulazione: il processore può cominciare ad interagire con lo scheduler e il gestore della memoria.

  - public int numeroIstanti():
    ritorna il numero degli istanti che compongono la simulazione.

1.1.5.2 Player

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica e concreta; scorre gli istanti che compongono una simulazione, in avanti o in dietro nel tempo.

- **Relazioni d'uso con altre componenti:**
  Utilizza il metodo crea() della classe Simulazione per creare la simulazione con i relativi istanti. Vengono inoltre utilizzati i metodi get della classe Istante, per identificare eventuali eventi rilevanti nel corso della simulazione. Vengono utilizzati i metodi AggiornaStatistiche e AzzeraStatistiche della classe interna Statistiche.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dalla GUI per lo scorrimento della simulazione.

- **Attività svolte e dati trattati:**
  - **Campi dati:**
    - private Simulazione simulazioneEseguita:
      la simulazione contenente gli istanti da scorrere con il Player.

    - private LinkedList<Istante> listaIstanti:
      la lista degli istanti generati dalla simulazione.

    - private int indiceElementoCorrente:
      l'indice dell'istante corrente; utilizzato per ottimizzare lo scorrimento degli istanti.

    - private ListIterator<Istante> istanteCorrente:

Iteratore che punta all'istante corrente; viene utilizzato per scorrere la lista.

- public enum Evento{FAULT, SWITCH, FULL_RAM, FULL_SWAP, END_PROC, NEW_PROC}:
  Enumerazione per la definizione degli eventi significativi che possono essere ricercati nella lista di istanti.

- public static Statistiche stat:
  Istanza della classe Statistiche.

- **Metodi:**
  - public boolean caricaSimulazione():
    carica la simulazione e inizializza l'iteratore per lo scorrimento della simulazione; ritorna un booleano che comunica lo stato di successo o fallimento dell'operazione.

  - public Istante istantePrecedente():
    ritorna l'istante precedente a quello attuale. Se viene ritornato un riferimento nullo, significa che non ci sono istanti precedente.

  - public Istante istanteSuccessivo():
    ritorna l'istante successivo a quello attuale. Se viene ritornato un puntatore nullo, significa che non ci sono istanti successivi.

  - public LinkedList<Istante> precedenteIstanteSignificativo (Evento e):
    ritorna l'istante significativo precedente all'istante corrente. Un istante significativo, corrisponde ad uno o più eventi del tipo: fault in memoria principale RAM, context-switch, riempimento della memoria centrale, riempimento dell'area di Swap, terminazione di un processo, arrivo di uno o più processi.

  - public LinkedList<Istante> prossimoIstanteSignificativo (Evento e):
    ritorna l'istante significativo successivo all'istante corrente. Un istante significativo, corrisponde ad uno o più eventi del tipo: fault in memoria principale RAM, context-switch, riempimento della memoria centrale, riempimento dell'area di Swap, terminazione di un processo, arrivo di uno o più processi.

  - public Istante primoIstante():

Ritorna il primo istante della simulazione.

- public LinkedList<Istante> ultimoIstante():
  Ritorna una lista di istanti che porta all'ultimo istante della simulazione.

- public int numeroIstanti():
  Ritorna il numero di istanti che compongono la simulazione.

- public Statistiche getStatistiche():
  Ritorna un riferimento all'oggetto interno stat in modo che dall'esterno siano accessibili i metodi della classe Statistiche.

1.1.5.3 Statistiche

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica e concreta, interna alla classe Player; raccoglie dati statistici nel range di istanti che va dall'inizio della simulazione all'istante corrente.

- **Relazioni d'uso con altre componenti:**
  Nessuna.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dalla classe Player per aggiornare le statistiche.

- **Attività svolte e dati trattati:**
  - **Campi dati:**
    - private int utilizzoRAM:
      la quantità in KB di RAM utilizzata.

    - private int utilizzoSwap:
      la quantità in KB di Swap utilizzata.

    - private int numeroFault:
      numero di Fault avvenuti in memoria.

    - private int numeroIstantiRimanenti:
      numero di istanti rimanenti alla fine della simulazione.

1.1.5.4 *Istante*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe pubblica e concreta; rappresenta una singola unità di tempo della simulazione. Al suo interno le informazioni sono rappresentate in modo differenziale; ciò significa che ogni istante rappresenta solo le differenze rispetto all'istante precedente.

- **Relazioni d'uso con altre componenti:**
  Nessuna.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dalla classe Player la quale interroga i campi dato per scoprire eventi significativi.

- **Attività svolte e dati trattati:**
  - **Campi dati:**
    - private PCB processoInEsecuzione:
      il processo in esecuzione nel'istante corrente.
    - private PCB processoPrecedenteTerminato:
      il processo terminato nell'istante corrente, cioè che ha eseguito il suo ultimo quanto di tempo nell'istante precedente.
    - private boolean nuovoProcesso:
      indica se sono arrivati nuovi processi nella coda dei pronti.
    - private int numeroFault:
      numero di fault generati nell'istante corrente.
    - private LinkedList<Azione> cambiamentiInMemoria:
      i cambiamenti avvenuti in memoria (RAM e Swap).
    - private boolean full_RAM:
      indica se la memoria RAM si è riempita a tal punto da non poter caricare nuove pagine/segmenti.
    - private boolean full_Swap:
      indica se l'area di Swap ha raggiunto la sua capacità massima.

1.1.6 logic.schedulazione

1.1.6.1 PoliticaOrdinamentoProcessi

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Una Politica di Ordinamento dei Processi ha il compito di ordinare i processi che sono attivi in un sistema, scegliendo quale di questi deve eseguire ad ogni istante. Una politica di ordinamento viene specializzata a seconda dei s istemi in cui questa verrà applicata. Elemento comune tra le varie politiche di ordinamento, è quello di dover mantenere i processi pronti all'interno di una s truttura dati.

- **Relazioni d'uso con altre componenti:**
  Nessuna

- **Interfacce e relazioni di uso da altre componenti:**
  Nessuna (vengono utilizzate le sue implementazioni).

- **Attività svolte e dati trattati:**

  - **Metodi:**
    - public void esegui()
      Metodo che simula l'esecuzione del processo che sta detenendo la CPU. La simulazione dell'esecuzione avviene aumentando il numero di istanti eseguiti di tale processo di un certo valore passato come parametro, valore che viene calcolato dallo scheduler.

    - public java.util.ArrayList getCodaPronti()
      Metodo che ritorna un ArrayList contenente tutti i riferimenti a Processo a cui sono associati i PCB contenuti nella coda dei pronti

    - public PCB estrai()
      Rimuove il primo elemento dalla struttura dati definita nella classe che lo implementerà.

    - public void setScheduler(Scheduler_scheduler)
      Metodo che imposta il campo dato di tipo Scheduler che dovrà essere previsto da ogni politica di ordinamento concreta che implementi direttamente questa interfaccia.

- public void inserisci(PCB pcb)
  Inserisce il PCB passato per riferimento nella struttura dati della classe che implementerà questa interfaccia.

- public int size()
  Ritorna il numero di PCB correnti presenti nella struttura dati definita nella classe che andrà ad implementarlo.

1.1.6.2 PCB

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Classe che rappresenta un process control block, ovvero un particolare oggetto che indicheremo con l'acronimo PCB e che si occupa di mantenere memorizzati tutti i dati di un processo che evolvono col trascorrere della simulazione. Istanze di questa classe perciò si occupano di memorizzare i dati dinamici relativi a dei processi. Ogni processo sarà associato ad un'istanza di questa classe e ogni istanza di questa classe sarà associata ad uno ed un solo processo. Dunque dall'istanza di PCB potrà identificare in modo univoco il processo associato, risalendo ai dati dinamici che lo caratterizzano.

- **Relazioni d'uso con altre componenti:**
  Utilizza il metodo getTempoEsecuzione dell'oggetto processo per determinare gli istanti da eseguire, al momento della creazione.
  Utilizza il metodo equals dell'oggetto processo per confrontare due processi rappresentati da due PCB.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene utilizzata da ogni implementazione dell'interfaccia PoliticaOrdinamentoProcessi per mantenere la lista dei PCB rappresentanti i processi nello stato di ponto.
  Viene utilizzata dalla classe Scheduler che crea ogni istanza di PCB a partire da un processo che passa dallo stato “in arrivo” a quello “terminato”. Lo Scheduler inoltre mantiene un riferimento al processo attualmente in esecuzione e ad una lista di quelli terminati.
  Viene utilizzata dalla classe Processore, che riceve dallo Scheduler un riferimento al PCB in esecuzione, ne estrae i FrameMemoria necessari e controlla che il PCB precedentemente in esecuzione non sia terminato.
  Viene utilizzata dalla classe Istante, di cui ogni istanza memorizza il PCB in esecuzione e quello appena terminato in un dato istante di tempo della

simulazione.

1.1.6.3 *Scheduler*

- **Tipo, obiettivo e funzione della classe/interfaccia:**
  Questa classe implementa i meccanismi necessari per realizzare una simulazione discreta di processi in un elaboratore multi-programmato. Lo Scheduler viene interrogato per ogni istante dal Processore, a cui ritorna un riferimento al PCB correntemente in esecuzione, che potrà essere un riferimento nullo nel caso in cui nessun processo abbia il controllo della CPU. L'esecuzione dei vari processi avverrà in maniera specializzata a seconda della politica di ordinamento scelta.

- **Relazioni d'uso con altre componenti:**
  Utilizza il metodo getTempoArrivo della classe Processo.
  Utilizza il metodo inserisci, esegui ed estrai di PoliticaOrdinamentoProcessi.
  Utilizza il metodo incIstantiEseguiti, getIstantiDaEseguire e getRifProcesso di PCB.

- **Interfacce e relazioni di uso da altre componenti:**
  Viene usata dall'interfaccia PoliticaOrdinamentoProcessi, da tutte le sue implementazioni e dal Processore.

- **Attività svolte e dati trattati:**
  - **Campi dati:**
    - package boolean fineSimulazione:
      Indica se la simulazione è giunta al termine o meno.

    - package PCB PCBCorrente:
      Il PCBCorrente rappresenta il PCB che, all'istante definito nella variabile tempoCorrente, è nello stato di esecuzione.

    - package PoliticaOrdinamentoProcessi politicaOrdinamento:
      Questo campo dati specifica la politica di schedulazione che si intende utilizzare per lo scheduler.

    - package java.util.LinkedList processiInArrivo:
      Contiene la lista dei processi da attivare, ordinata per tempo di arrivo.

    - package java.util.ArrayList processiTerminati:

Questo campo dati contiene la coda dei processi terminati in ordine di terminazione.

- package PCB PCBCorrente:
  Il PCBCorrente rappresenta il PCB che, all'istante definito nella variabile tempoCorrente, è nello stato di esecuzione.

- package int tempoCorrente:
  E' il contatore interno dello Scheduler.

- package int[] tempoEvento:
  Questa struttura serve per memorizzare i tempi dei due eventi che possono ricorrere in questa simulazione di schedulazione: l'arrivo e la terminazione di un processo.

- package int tempoProssimoEvento:
  Salva il valore ritornato dal metodo tempoProssimoEvento().

- **Metodi:**
  - package void attivaProcesso()
    Questo metodo si occupa di attivare un nuovo processo arrivato prendendo il primo della lista dei processi in arrivo, ne crea il PCB e lo inserisce nella lista dei pronti a seconda della politica di schedulazione scelta.

  - public boolean eseguiAttivazione()
    Questo metodo invocato dal Processore si occupa di preparare il prossimo PCB in esecuzione. Ritorna true se alla fine della sua esecuzione non è disponibile un PCB in esecuzione, false altrimenti.

  - public void eseguiIncremento():
    Questo metodo viene invocato dallo scheduler solo nel caso ci sia un PCB che può eseguire e fa progredire la simulazione.

  - public boolean fineSimulazione():
    Questo metodo ritorna lo stato della simulazione.

  - public PCB getPCBCorrente:
    Ritorna il PCB attualmente nello stato di esecuzione.

- public PoliticaOrdinamentoProcessi getPoliticaOrdinamentoProcessi() :
  Ritorna la politica di ordinamento scelta nel costruttore.

- package java.util.LinkedList getProcessiInArrivo():
  Ritorna la lista dei processi che devono ancora essere attivati, ordinata per tempo di arrivo.

- public java.util.ArrayList getProcessiTerminati()
  Ritorna la lista dei processi sono terminati in ordine di terminazione.

- public int getTempoCorrente():
  Ritorna un intero che rappresenta il numero di istanti di simulazione trascorsi.

- package void incrementaTempo():
  Incrementa di un'unità di tempo il contatore del tempo corrente.

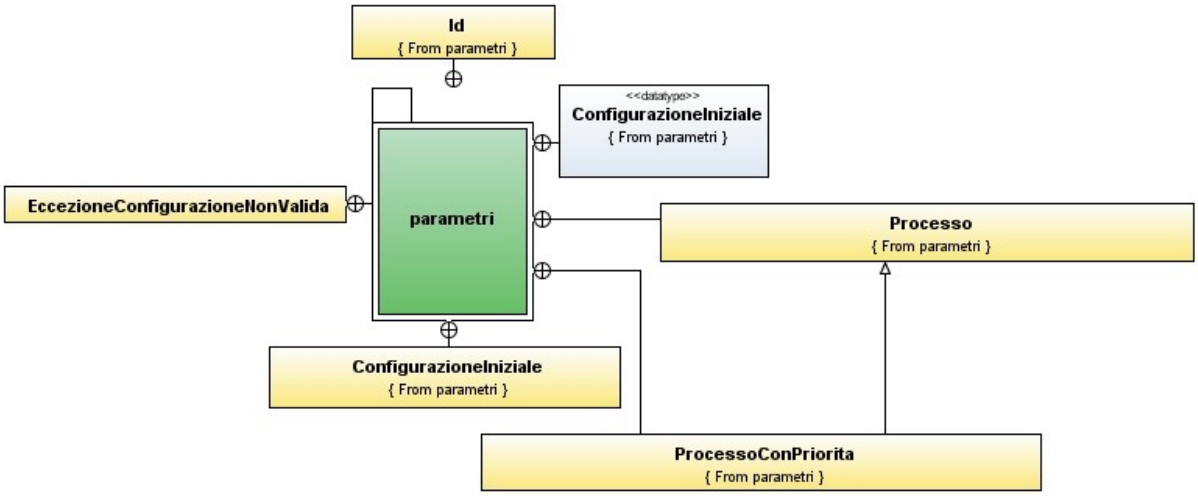
- public void incrementaTempo():
  Questo metodo incrementa il tempo corrente dello scheduler e decrementa automaticamente i tempi dei prossimi eventi di una unità.

- package void rimuoviPCBCorrente() :
  Rimuove il PCBCorrente dalla posizione di esecuzione.

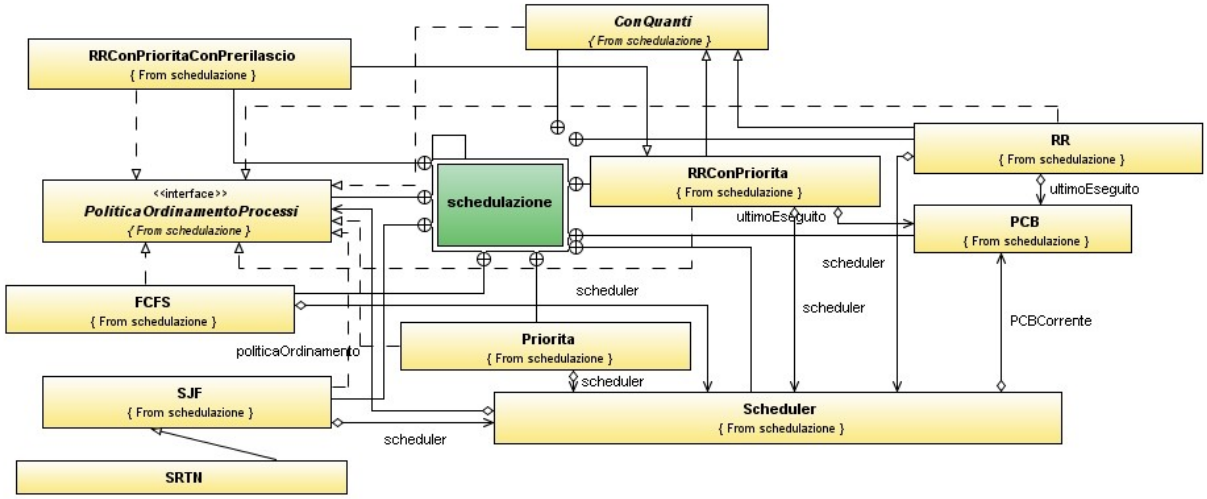
- package int tempoProssimoEvento():
  Questo metodo calcola l'intervallo tra il tempo attuale e il prossimo evento.

- package void terminaPCBCorrente() :
  Metodo invocato quando un PCB ha completato la sua esecuzione.

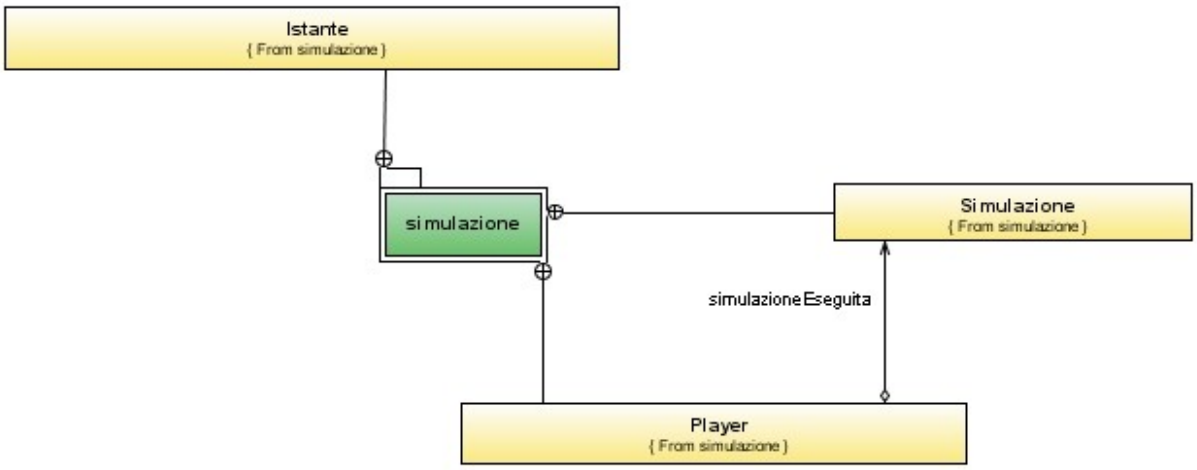
4.1 Diagramma Parametri



4.2 Diagramma Schedulazione



4.3 Diagramma Simulazione



4.4 Diagramma Gestione Memoria

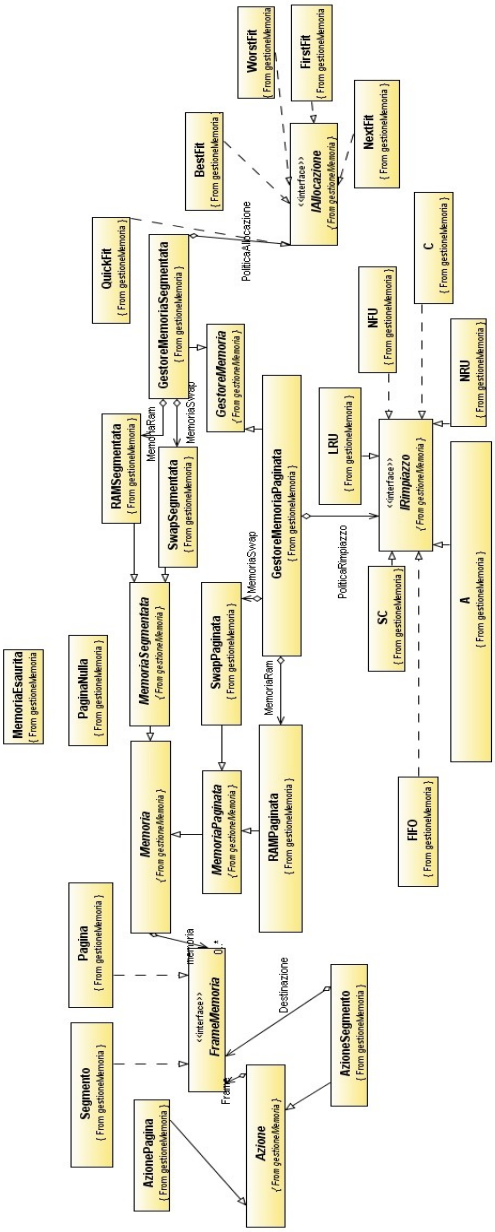


Figura 4.4

5 Diagrammi di sequenza

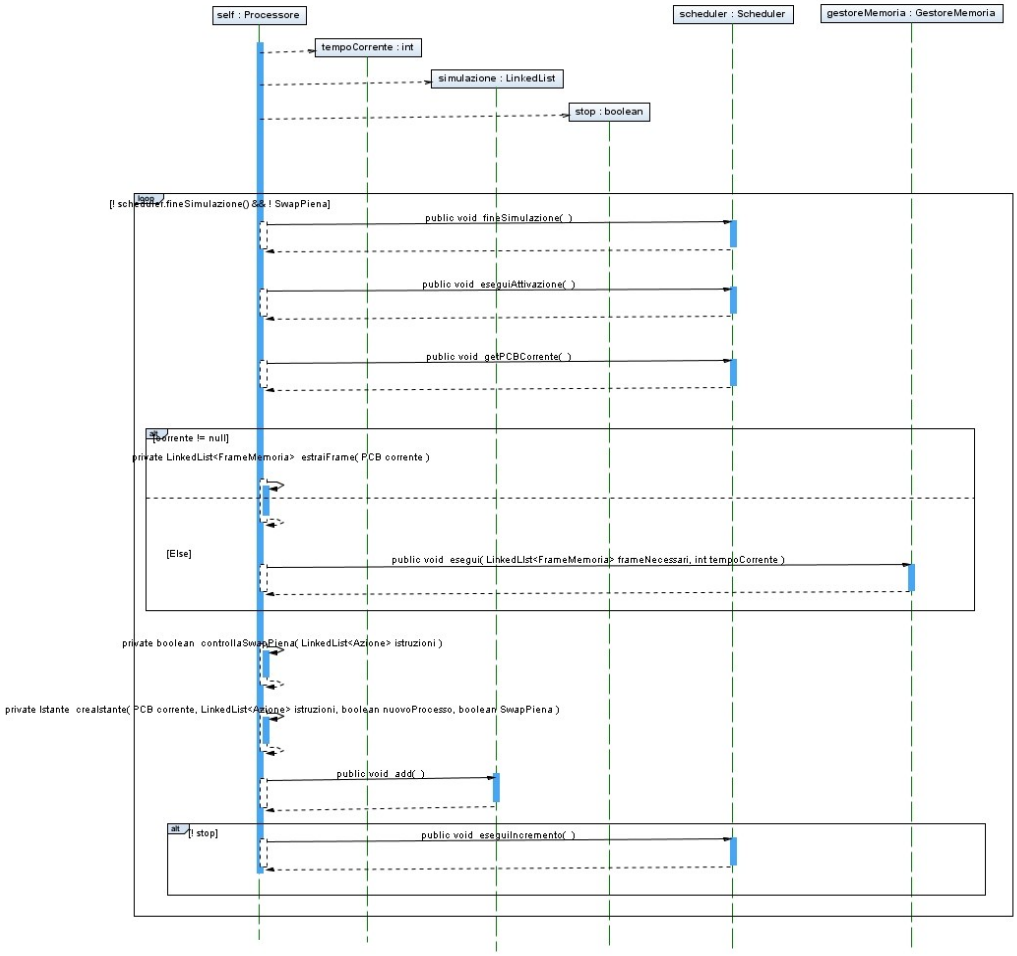


Figura 4.5

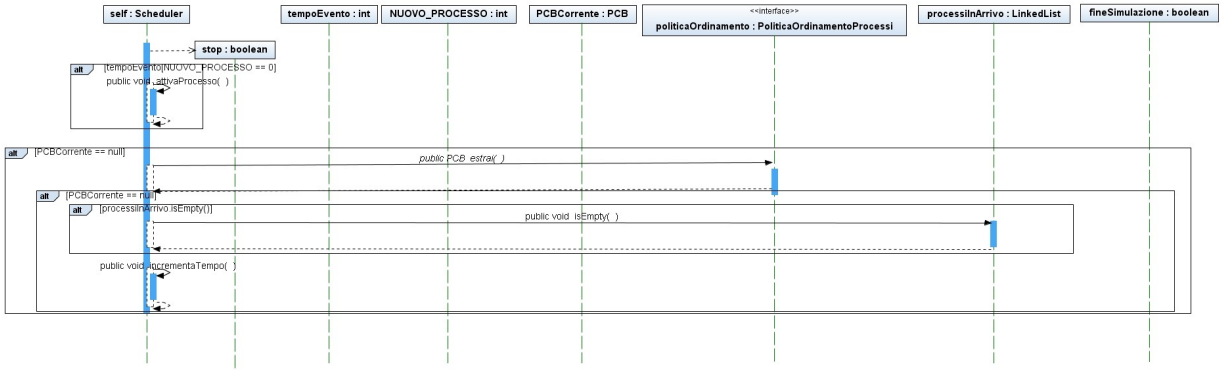


Figura 4.6

Figura 4.7

6 Stime di fattibilità e bisogno di risorse

Per l'impegno ed il tempo richiesti alla realizzazione, fare riferimento a [PP]. Per quanto riguarda le risorse hardware, umane e competenze individuali, fare riferimento a [SF].

Il progetto verrà realizzato utilizzando il linguaggio *Java* per i le seguenti motivazioni:

- tutti i componenti hanno familiarità con questo linguaggio di programmazione;
- questo linguaggio permette la portabilità del programma da una piattaforma ad un'altra, requisito implicito evidenziato in [AR] con il codice RNO08. La portabilità è stata messa in rilievo rispetto a l'ottimizzazione del codice che si può ottenere con altri linguaggi di programmazione, come ad esempio C;
- Java permette il salvataggio tramite serializzazione di oggetti, rendendo semplice la gestione dell'I/O;
- per quanto riguarda l'implementazione della parte grafica, Java mette già a disposizione librerie grafiche adatte al nostro scopo.

7 Tracciamento componenti – requisiti

Componente	Requisiti
gui	RFO11, RFO12, RFO13, RFO14, RFO15, RFO19, RFO21,

	RFO23, RFD02, RNO05, RNP01, RNO09, RNO10, RNO11
simulazione	RFO18, RFO19, RFO20, RFO23, RFD03
parametri	RFO03, RFO04, RFO05, RFO06, RFO07, RFO08, RFO09, RFO10, RFO17, RFO22, RFO24, RFD02, RNO06
gestioneMemoria	RFO01, RFO02, RFD01,RFP03
schedulazione	RFP01,RFP03
caricamento	RFO16, RFP02