Un Sistema de Simulación de Tareas sobre Recursos, en un Contexto Organizacional

Federico Rossi - Emmanuel Luján

PREINFORME

1 Información General

Fecha:	Diciembre del 2011
Tema:	Un Sistema de Simulación de Tareas sobre Recursos, en un Contexto Organizacional
Materia:	Diseño de Sistemas de Software
Carrera:	Ingeniería de Sistemas
Universidad:	Universidad Nacional del Centro de la Provincia de Bs. As.
Docentes:	Dr. Marcelo Campo Dr. Andrés Díaz Pace Dr. Alvaro Soria Dr. Luis Berdún Ing. Agustín Casamayor Ing. Juan Feldman Ing. Mauricio Arroqui
Autores:	Federico Rossi - Emmanuel Luján

2 Índice

2.1 Índice de contenido

2.2 Índice de ilustraciones

3 Objetivos

En la presente sección de plantean los objetivos generales que se desean satisfacer en el desarrollo de este proyecto:

Se pretende realizar un sistema de simulación de tareas sobre recursos en un contexto organizacional. La organización se modela mediante una red de trabajo que se compone por recursos, actores (ej. empleados) y artefactos (ej. documentos), y sus propiedades, relacionados entre ellos. Los actores realizan tareas, que son realizadas bajo el cumplimiento de ciertas condiciones sobre las propiedades de la red de trabajo, y que pueden modificar dicha red. Los actores ejecutan las tareas de forma concurrente y pueden trabajar varios sobre la misma tarea, de a uno por vez. Una tarea puede fallar, y en su lugar se debe poder ejecutar una tarea de contingencia.

Además se deben registrar los eventos de la simulación para poder tomar métricas sobre los resultados obtenidos, al final de la misma.

4 Requerimientos

4.1 Contexto

Una organización es:

- Un grupo social compuesto por personas, tareas y administración, que forman una estructura sistemática de relaciones de interacción, tendientes a producir bienes y/o servicios para satisfacer las necesidades de una comunidad dentro de un entorno y así poder satisfacer su propósito distintivo que es su misión.
- Un sistema de actividades conscientemente coordinadas formado por dos o más personas; la cooperación entre ellas es esencial para la existencia de la organización. Una organización solo existe cuando hay personas capaces de comunicarse y que están dispuestas a actuar conjuntamente para obtener un objetivo común.
- Un conjunto de cargos con reglas y normas de comportamiento que han de respetar todos sus miembros, y así generar el medio que permite la acción de una empresa. La organización es el acto de disponer y coordinar los recursos disponibles (materiales, humanos y financieros). Funciona mediante normas y bases de datos que han sido dispuestas para estos propósitos.

[WKP1]

4.2 Requerimientos funcionales

En la presente sección se describe de forma general el funcionamiento de la aplicación:

Antes de comenzar la simulación el usuario de la aplicación realiza la carga de la información por medio de la interfaz gráfica. Pueden cargarse casos de prueba previamente realizados. En esta instancia se configura un modelo compuesto principalmente por actores, artefactos, tareas y sus relaciones.

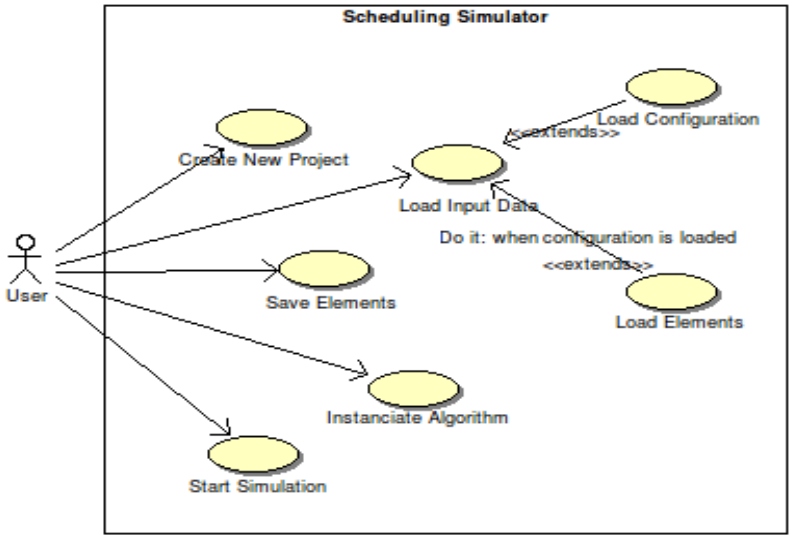


Ilustración 1: Diagrama de Casos de Uso General

Al comenzar la simulación, las tareas que deben realizarse todavía no están asignadas a sus respectivos actores. Un cierto actor (dealer actor) tiene el rol de tomar cada tarea y asignarla a su respectivo trabajador.

Cuando un actor trabajador recibe una tarea, se verifica que no se sobrepase el límite de tareas máximo que puede realizar dicho actor. En tal caso se notifica este evento como un error, no se agrega la tarea y se prosigue con la simulación.

Las tareas pueden fallar. Antes de ejecutar una tarea se verifica que la capacidad del actor corresponde a la dificultad de la tarea, si no es así la tarea falla. También, cuando una tarea es realizada por un actor se verifica que se cumplen ciertas condiciones. Por ejemplo que el actor que la realiza se categoría A. Si en algún momento, durante la realización de la tarea, no se cumple alguna condición la tarea falla.

Para los casos donde se producen fallas las tareas pueden tener asociadas una tarea de contingencia, la tarea original termina y en su lugar se ubica la de contingencia.

Un actor posee dos listas de tareas en espera para ser ejecutadas: una lista convencional, y la lista de interrupciones. La diferencia principal es la prioridad de la segunda sobre la primera. Cada vez que se elige realizar una tarea siempre se consulta primero la lista de interrupciones, y si esta se encuentra vacía se consulta la otra. Si una tarea arriba a la lista convencional el procesamiento continúa normalmente, pero si arriba a la lista de interrupciones la tarea que se estaba ejecutando pasa a su respectiva lista y selecciona una tarea de la lista de interrupciones. Hasta no acabar con las tareas de la lista de interrupciones no se trabaja en las tareas de la otra lista.

Cada lista tiene asociado un algoritmo de selección, por ejemplo la primera tarea en entrar es la primera tarea en salir (FCFS).

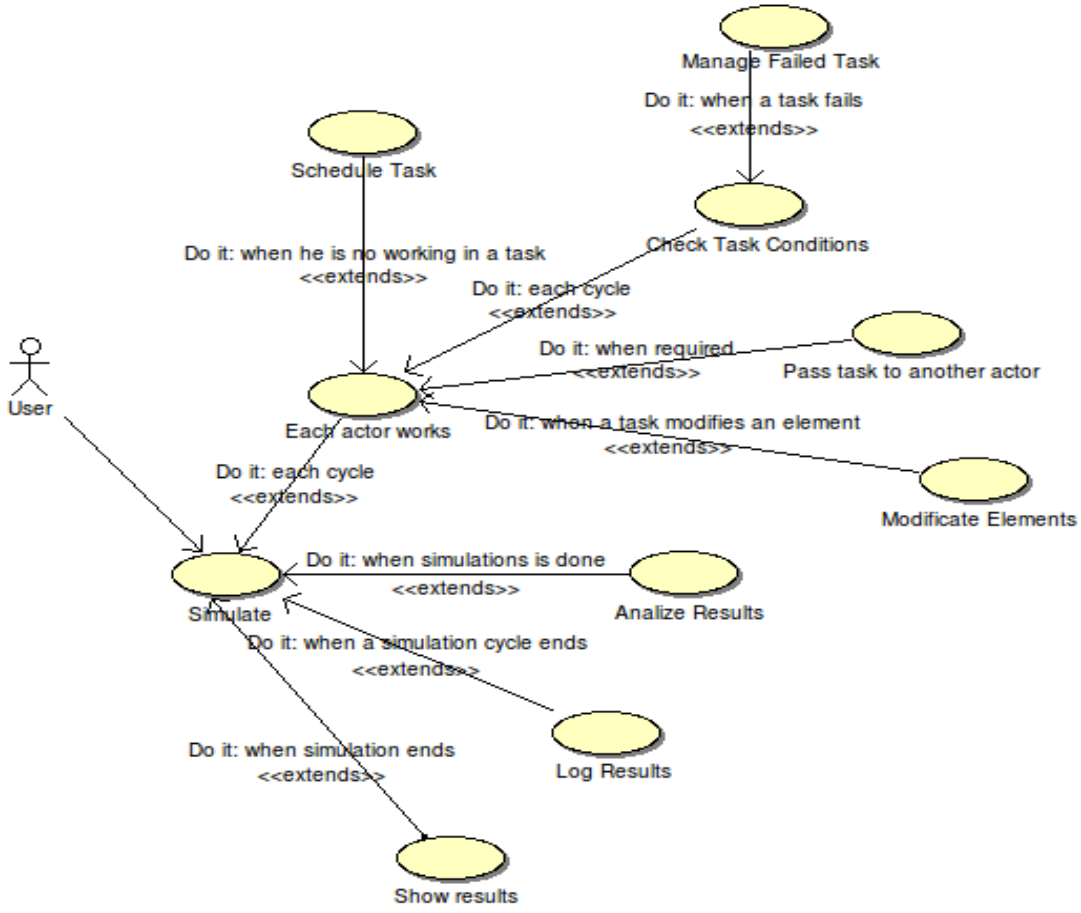
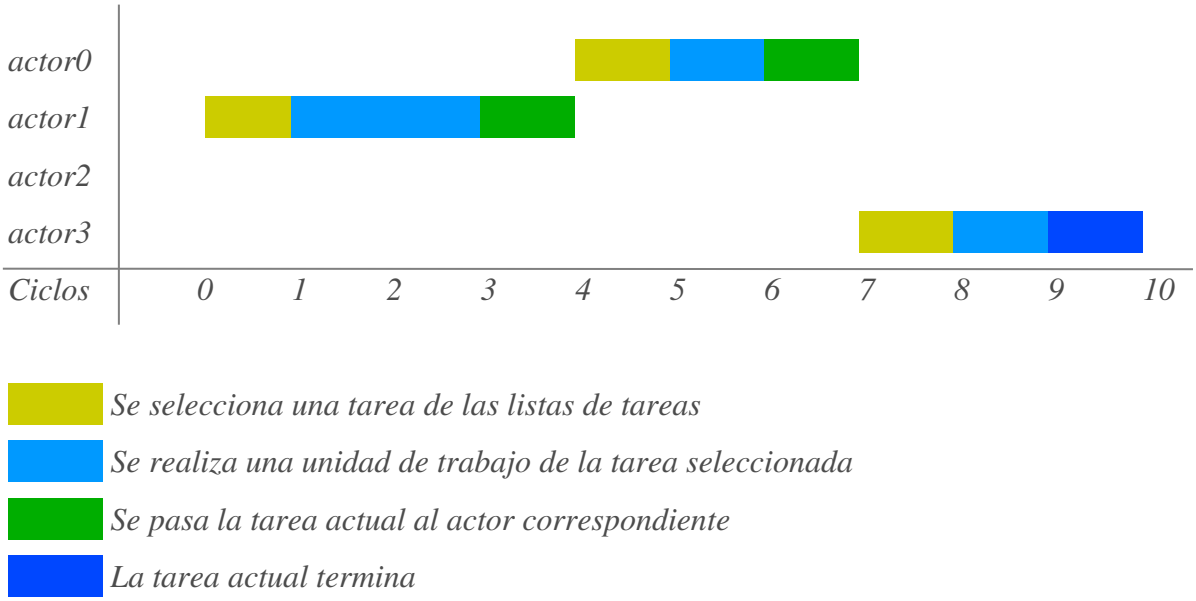


Ilustración 2: Explosión del Caso de Uso: Start Simulation

Las tareas poseen una secuencia de unidades de trabajo, cada unidad se ejecuta en un ciclo. Varios actores pueden ejecutar una tarea de a uno por vez y en el orden secuencial definido. Por ejemplo, la secuencia de una cierta *tarea0* puede ser: {*actor1, actor1, actor0, actor3*}, lo cual significa que la tarea requiere ser realizada sus primeros dos ciclos por el *actor1*, el tercer ciclo por el *actor0* y su último ciclo por el *actor3.*

En cada ciclo, cada actor puede: seleccionar una tarea de una lista, trabajar en una tarea o pasar una tarea a otro actor. Los actores realizan las tareas de forma concurrente, sin embargo esta actividad es implementada de forma secuencial. En el siguiente diagrama de Gantt se muestra gráficamente el ejemplo anterior:



4.3 Restricciones impuestas por el contexto o por los stakeholders

En la presente sección se ampliarán los objetivos anteriores en base a los requerimientos del sistema:

- Recurso
  - Es un actor (ej.: empleado) o artefacto (ej.: documento).
  - Tienen un conjunto de propiedades (tuplas nombre-valor).

- Actor
  - Puede realizar tareas.
  - Posee una capacidad o eficiencia para realizar cada tarea en particular.
  - Tiene un número limitado de tareas que debe realizar.
  - Sabe como planificar la realización de sus tareas.
  - Cada actor puede tener una manera distinta acerca de como planificar sus tareas.

- Relaciones entre recursos
  - Pueden ser
    - Entre actores.
    - Entre actores y artefactos: lectura y/o escritura de un actor hacia un artefacto.
  - Las relaciones tienen un conjunto de propiedades (tuplas nombre-valor).

- Red de trabajo
  - Grafo cuyos nodos son recursos y sus arcos son las relaciones entre los mismos.
  - Cada nodo tiene un número fijo de arcos de entrada y de salida.
  - Esta red es estática, y se pre-configura antes de iniciar la simulación.

- Tarea
  - Unidad de trabajo. Proceso o trabajo realizado por un actor.
    - Tareas convencionales.
    - Tareas de contingencia: una tarea convencional puede fallar al ser realizada (en base a la capacidad del actor y a la dificultad de la tarea), entonces se debe realizar una tarea de contingencia en su lugar. Esta situación supone el gasto de una mayor cantidad de tiempo.
  - Posee un cierto tiempo que necesita para ser realizada.
  - Posee un cierta dificultad.
  - Posee un conjunto de tareas de contingencia en caso de que falle.
  - Para ser realizada necesita del cumplimiento de pre-condiciones:
    - El cumplimiento previo de otras tareas.
    - En la red de trabajo:
      - Condiciones sobre propiedades de recursos o relaciones.
      - Condiciones sobre la existencia de relaciones o propiedades.
      - Por ejemplo: relaciones con otros actores, disponibilidad de ciertos documentos, o ciertas propiedades con ciertos valores en el actor/artefacto involucrado.
  - Luego de ser realizada produce efectos sobre la red de trabajo.
    - En las propiedades de los recursos o relaciones
    - Por ejemplo: al finalizar una tarea se modifica un documento que es necesario para la realización de otra tarea.
  - Puede tener partes críticas que deben realizarse de forma ininterrumpida. Las partes que no son críticas sí pueden interrumpirse.
  - Puede ser realizada por más de un actor, no al mismo tiempo. O sea, un actor puede derivar una tarea a otro actor.

- Log
  - El sistema debe registrar los datos de cada evento de la simulación.

- Luego de ejecutar la simulación (eventualmente varias corridas de la misma, con algunos parámetros random), es importante permitir tomar métricas. Ejemplos:
  - T: El tiempo total de la realización de las tareas.
  - Ti(a): El tiempo de inactividad de un actor.
  - Tt(t): El tiempo total de una tarea.
  - Cte: Cantidad de tareas exitosas.
  - Ctf: Cantidad de tareas fallidas.
  - Ctt: Cantidad total de tareas.
  - Proporciones como:
    - Proporción de éxitos: Cte / Ctt
    - Proporción de fallos: Ctf / Ctt
    - Proporción de velocidad: T / Tdeadline

Además:

- El sistema debe ser desarrollado en Java.
- La simulación involucra un tiempo "discreto", y no necesita "concurrencia real" en la implementación.
- La interface gráfica debe ser suficiente para facilitar el testeo del sistema.
- Se pide construir un ejemplo pequeño para poder simularlo y tomar métricas.

5 Arquitectura de Software

La arquitectura de software de un programa o sistema de computación es la estructura o estructuras de un sistema, las cuales comprenden los componentes de software, las propiedades visibles externamente de esos componentes, y las relaciones entre ellos. [SAIP]

5.1 Frameworks

Un framework es un conjunto de clases cooperativas que construyen un diseño parcial y reusable con el cual las aplicaciones dentro de un dominio son creadas a través de especializaciones. De la misma forma que una clase abstracta aúna una familia de clases con características en común, un framework aúna un conjunto de aplicaciones con características en común.

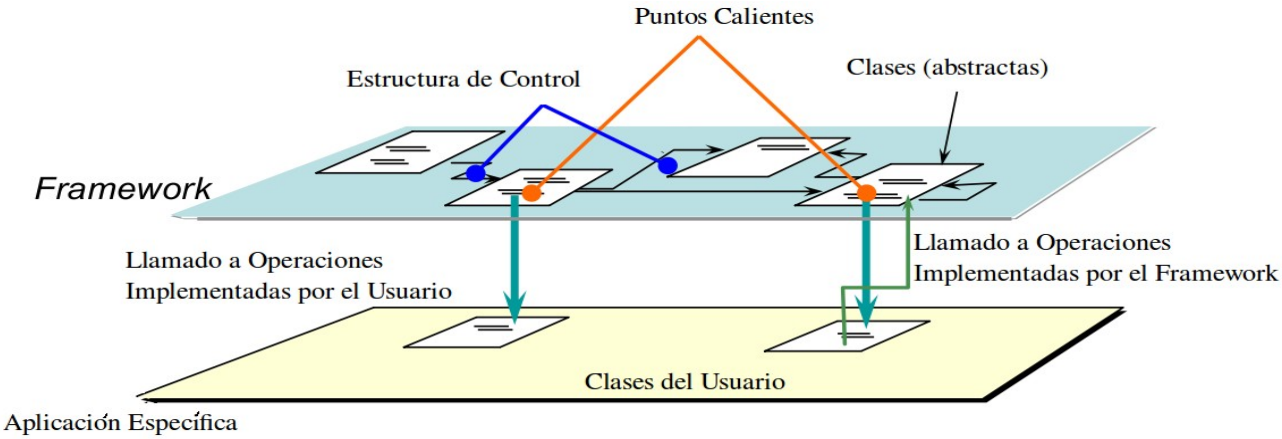


Ilustración 3: Conceptos Básicos de Frameworks

Al instanciar un framework se genera una aplicación particular. El control principal de la aplicación no se encuentra en la instanciación, sino en el framework. Como se puede ver en el gráfico de arriba las estructuras de control están en el framework. Esta propiedad se llama inversión de control. Por otro lado para instanciar un framework se necesita implementar métodos abstractos definidos dentro del framework, estos métodos se llaman puntos calientes.

A veces los métodos que necesitan ser instanciados poseen cierta estructura que deben respetar. Para solucionar esto dichos métodos (que están dentro del framework) se implementan, por lo que no son abstractos, en función de otros métodos que sí son abstractos. El método implementado parcialmente (en pos de que se respete la estructura requerida) se denomina método plantilla y los métodos que usa el método plantilla, que sí son abstractos, se llaman métodos gancho. La clase que pertenece al framework se denomina clase plantilla y la que implementa los métodos gancho se llama clase gancho.

Por último los frameworks de caja blanca son aquellos en donde en la especialización se usa herencia; mientras que en los de caja negra en la especialización se usa composición.

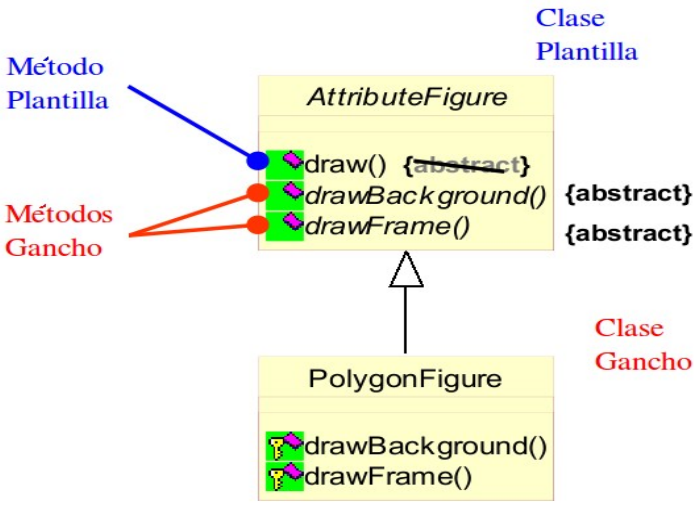


Ilustración 4: Métodos y Clases Plantilla y Gancho

[DSS07]

6 Patrón de Capas

Un patrón arquitectónico en software, también conocido como un estilo arquitectónico, consiste de unas pocas funcionalidades y reglas para combinarlos de manera que la integridad arquitectónica sea preservada. Un patrón arquitectónico está determinado por:

- Un conjunto de tipos de elementos (como un repositorio de datos o un componente que computa una función matemática).
- Una distribución topológica de elementos indicando sus interrelaciones.
- Un conjunto de restricciones semánticas.
- Un conjunto de mecanismos de interacción que determinan como los elementos se coordinan a través de la topología permitida. [SAIP]

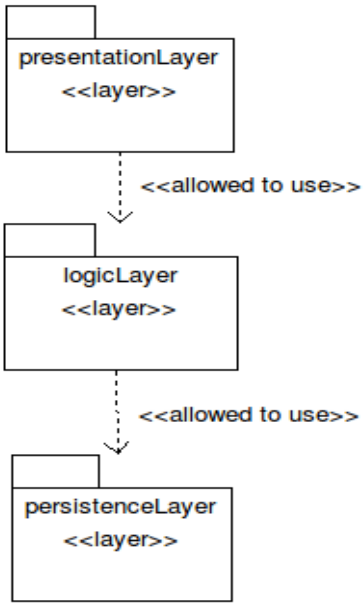


Ilustración 5: Arquitectura por Capas

Un estilo es un paquete predefinido de decisiones de diseño. [DSS11]

Un patrón muy común en la práctica es el patrón arquitectónico por capas, que ayuda a estructurar aplicaciones que pueden ser descompuestas en grupos de subtareas en los cuales cada grupo de subtareas se encuentra a un nivel de abstracción particular. [POSA]

Un sistema de capas aparece cuando las relaciones de usos en esta estructura son cuidadosamente controladas en una manera particular, en las cuales una capa es un conjunto coherente de funcionalidad relacionada. En una estructura estrictamente en capas, la capa N puede

solo usar los servicios de la capa N–1. Sin embargo, muchas variaciones de esta (y con una disminución de esta restricción estructural) ocurren en la práctica. Las capas son a menudo diseñadas como abstracciones que esconden detalles específicos de implementación debajo de las capas, creando *portabilidad y modificabilidad,* porque un cambio en una capa inferior puede ocultarse detrás de su interfaz y no afectará a las capas de arriba.

En el proyecto actual se pueden diferenciar tres capas principales. La capa superior o de presentación, muestra una interfaz gráfica con la que el usuario puede interactuar para realizar una nueva simulación, la entrada de datos, carga y guardado, y mostrado de resultados. La capa media, o capa lógica, se encarga de realizar la simulación propiamente dicha, un seguimiento o log de la misma, y el análisis de resultados. Y finalmente la capa inferior o de persistencia, se encarga de realizar el almacenamiento en disco de los datos necesarios para realizar una simulación, el seguimiento de una simulación particular y los resultados obtenidos de la misma.

7 Diseño Detallado

En esta sección se desarrollarán las capas nombradas anteriormente con los segmentos que las componen y las relaciones entre los mismos.

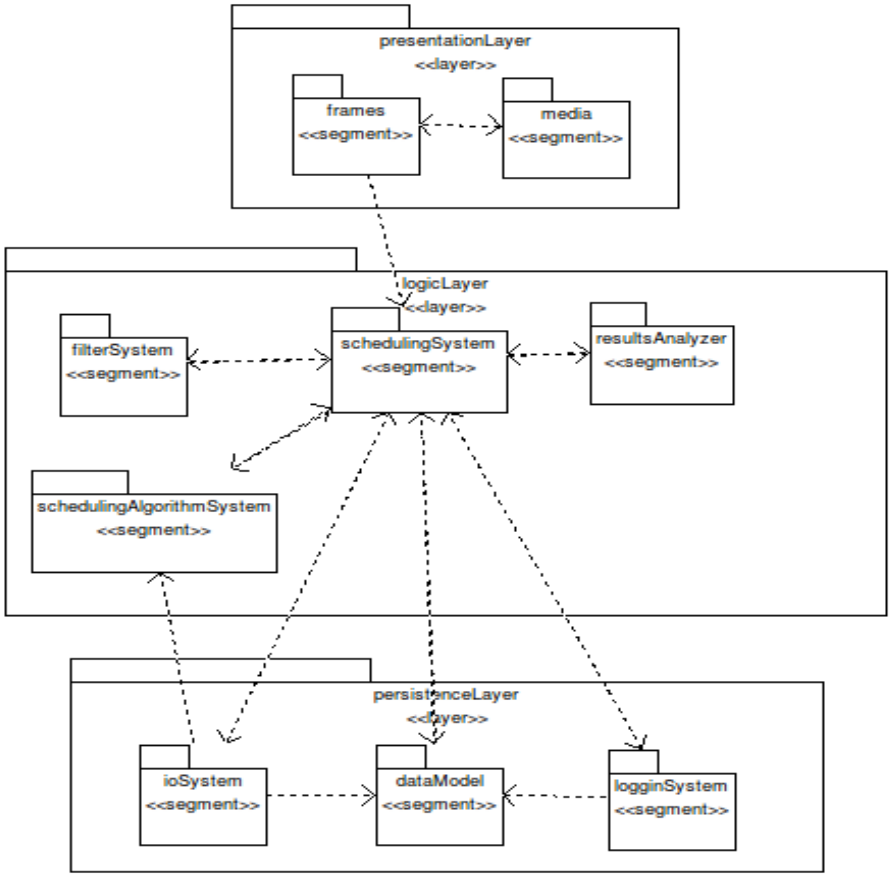

presentationLayer
<<layer>>
frames
<>
media
<>
logicLayer
<<layer>>
filterSystem
<>
schedulingSystem
<>
resultsAnalyzer
<>
schedulingAlgorithmSystem
<>
persistenceLayer
<<layer>>
ioSystem
<>
dataModel
<>
logginSystem
<>

Ilustración 6: Explosión de los paquetes de las capas

7.1 Capa de Presentación (presentationLayer)

La capa a ser descripta es la encargada de ofrecer la interfaz visual e interactiva al usuario. La interfaz gráfica de usuario o graphic user interface posee un conjunto de formas, elementos, y métodos que posibilitan la interacción de un sistema con los usuarios utilizando formas gráficas e imágenes. Con las palabras formas gráficas, se refiere a botones, ventanas, paneles, fuentes, etc; los cuales representan funciones, acciones e información. Por ejemplo, el escritorio del sistema operativo Windows, es una interfaz de usuario.

En el contexto del proceso de interacción persona-ordenador, la interfaz gráfica de usuario es el artefacto tecnológico de un sistema interactivo que posibilita, a través del uso y la representación del lenguaje visual, una interacción amigable con un sistema informático.

En el caso de la herramienta desarrollada, la interfaz gráfica le brindará al usuario la

posibilidad de que el mismo invoque comandos, actualice vistas y cargue datos para realizar el análisis de rendimiento de la estructura laboral modelada. Principalmente la misma servirá para que el usuario pueda cargar los diferentes datos de entrada y visualizar los resultados obtenidos de una forma más amigable.

La capa de presentación se comunicará simplemente con la capa de lógica del sistema para transmitir los datos de entrada del análisis y obtener consecuentemente los resultados del mismo mostrándoselos al usuario.

La capa analizada está constituida por dos paquetes, Frames y Media. El primero de ellos como su nombre lo dice, es quien contiene todos los frames de la herramienta con sus diferentes elementos. El segundo, posee recursos que brinda o necesita la herramienta. Entre tales paquetes existe un vínculo para que los diferentes frames puedan adquirir recursos en caso de que los necesiten, tomándolos del paquete Media. El paquete que entablará el vínculo de interacción con la capa lógica será el paquete Frames, que de acuerdo a los comandos que invoque el usuario se comunicará con tal capa.

7.1.1 Frames

Este paquete está constituido por un conjunto de clases que corresponden a las diferentes ventanas de la herramienta, junto a los elementos gráficos que forman las mismas. Cada una de ellas se encargará de ofrecer un tipo de función diferente y específica.

En esta descripción se irán identificando las clases del paquete y comentando su funcionalidad. La clase principal de este conjunto es la controladora central de la aplicación, SimulatorFrame, quien se encarga de administrar el panel de pestañas principal y sus acciones relacionadas. Esta clase controla el conjunto de elementos creados del modelo laboral para luego suministrárselos a la capa lógica, donde la misma realizará el análisis de la estructura creada y le devolverá los resultados obtenidos para que sean mostrados.

Dentro de la ventana que produce esta clase, existen los enlaces con el resto de las clases gráficas. En la segunda pestaña dentro de ella, se encuentra el acceso a las ventanas interactivas que se encargan de la creación de los elementos que forman el modelo laboral, actores, artefactos/recursos y tareas, correspondiéndose con las clases CreateActorFrame, CreateArtifactFrame y CreateTaskFrame. Las primeras dos nombradas ofrecen la funcionalidad para crear un Actor (Trabajador/Empleado), un Artefacto (Recurso físico) y sus correspondientes propiedades. Mientras que la tercera clase nombrada, se encarga de crear las tareas que deberán resolverse, junto a propiedades de las mismas. Dentro de su creación pueden construirse, opcionalmente, un actualizador del grafo que forma el modelo laboral y las correspondientes actualizaciones que se realizarán una vez que se resuelva la tarea a la cual pertenecen. Por ejemplo, si debe cambiarse o agregarse una propiedad nueva a un recurso una vez que la tarea creada se resuelva, deben establecerse sus parámetros. Tales parámetros se establecerán en las ventanas correspondientes a la creación del actualizador (updater) y sus actualizaciones (updates). Las clases que se involucran en tales ventanas son, CreateFilterFrame, UpdateFrame y UpdaterFrame. CreateFilterFrame, presenta la interfaz para la creación de filtros. En la creación de tareas por ejemplo, en caso de que quieran realizarse actualizaciones una vez resuelta la misma, debe

especificarse un filtro que determine si las actualizaciones pueden ser realizadas, siempre y cuando se cumpla con los parámetros especificados en el mismo. Y posteriormente dentro de la creación del actualizador (updater), se establecerá el conjunto de actualizaciones a realizar, correspondientes al UpdateFrame, y otro filtro relacionado con las mismas para que tales se activen.

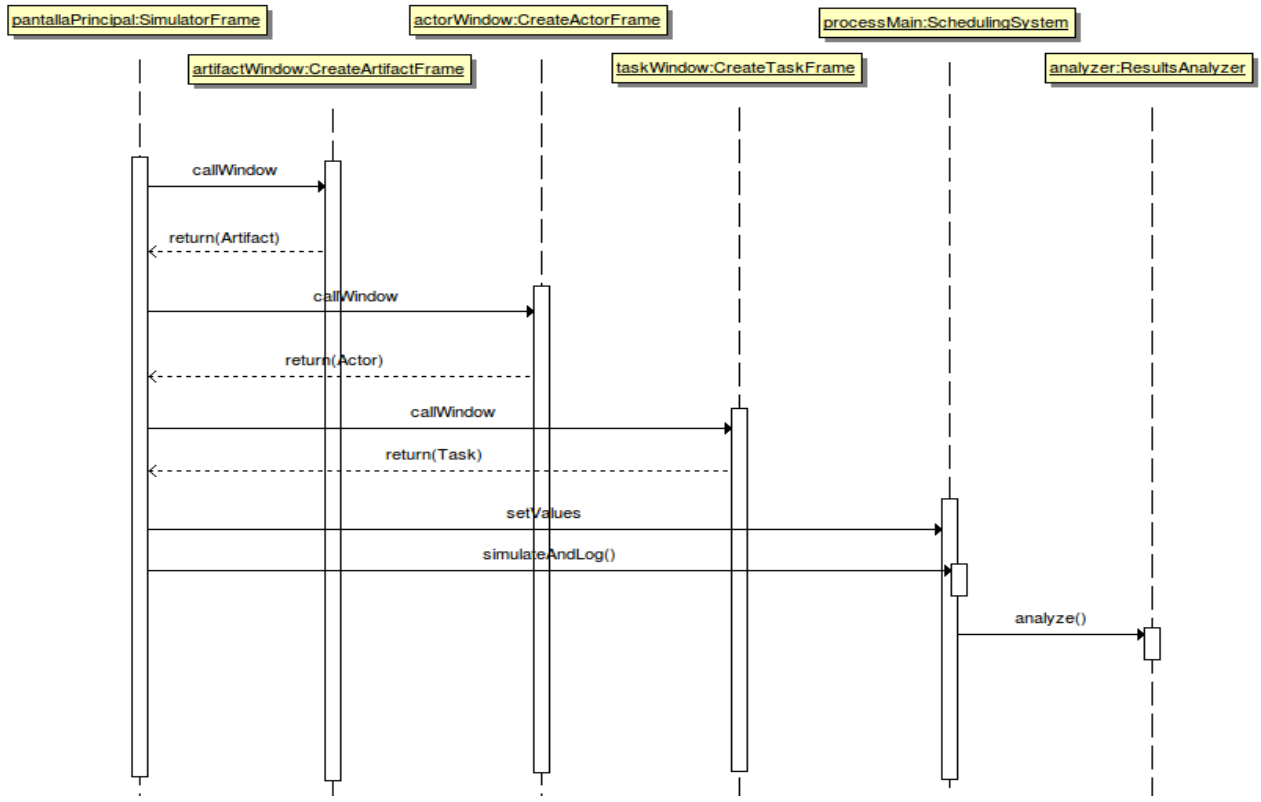


Ilustración 7: Diagrama de secuencia: interacción entre el usuario y el simulador

Una vez creados el total de elementos del modelo, pueden también crearse relaciones entre actores y/o recursos. Para crear tales, RelationFrame es la clase que ofrece tal funcionalidad. Mediante la incorporación de relaciones pueden reflejarse fielmente los vínculos que existen en una estructura laboral real, marcando relación entre trabajadores, entre un trabajador y un recurso, etc.

Por último, el paquete contiene clases encargadas también de la eliminación de elementos del modelo laboral, DeleteActorFrame, DeleteArtifactFrame, DeleteTaskFrame y clases que muestran noticias o errores reportados por la herramienta a partir de las acciones del usuario, ErrorFrame y NewsFrame. De esta forma y mediante este conjunto de clases, correspondientes al total de ventanas de la aplicación se formará la interfaz de usuario reflejada.

Un patrón de diseño usado en la mayoría de los frames es el patrón singleton. Tal patrón se ha adoptado debido a que en todo momento debe existir simplemente una sola instancia de las clases que reflejan la mayoria de los frames, salvo con los filtros y las ventanas actualizadoras (updater) que no ocurre tal caso y por ello no se adopto tal patrón para su correspondientes frames. Es importante para algunas clases que reflejan ventanas solo tener una instancia. Una variable global hace a un objeto accesible, pero no evita que se creen más de una instancia. La mejor

solución para realizar la propiedad deseada es hacer que la clase sea la responsable de gestionar una sola instancia. La clase puede asegurar que ninguna otra instancia será creada (interceptando los requerimientos de crear nuevos objetos), y puede proveer un punto de acceso a tal instancia. Debido al alto grado de acoplamiento entre las ventanas de la herramienta y la propiedad ofrecida por este patrón, se escogió su uso.

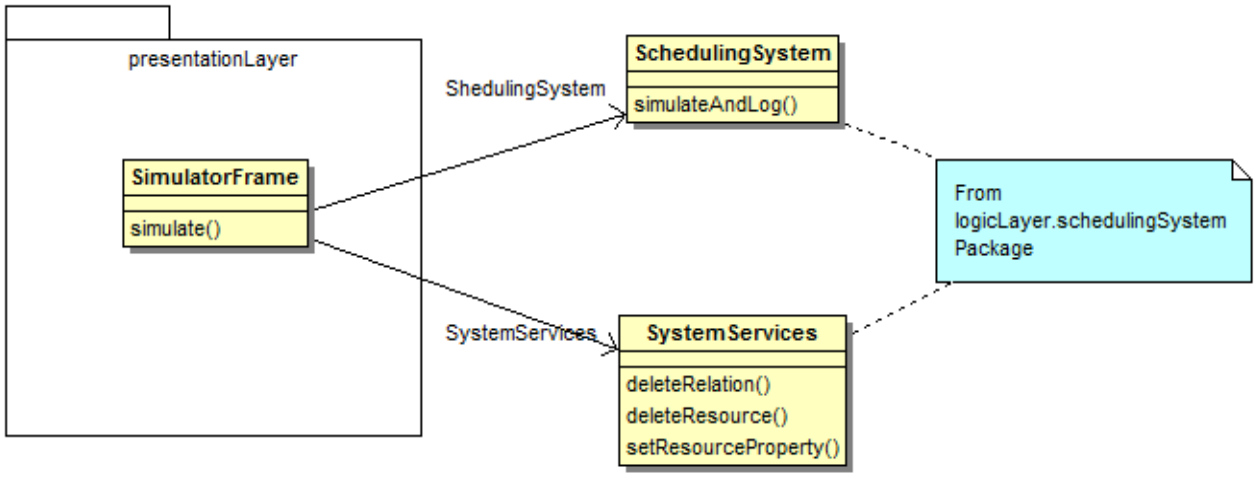


Ilustración 8: Diagrama de Clases: Interacción SimulatorFrame y Clases de Capa Lógica

7.1.2 Media

Este paquete es el encargado de contener distintos elementos útiles para la herramienta como imágenes que se incorporan en las ventanas, estilos gráficos y el manual de usuario de la aplicación.

7.2 Capa Lógica (logicLayer)

La capa lógica es la encargada de realizar la simulación y analizar los resultados de la misma. Ésta se comunica con la capa de persistencia para almacenar y recuperar los datos de una simulación y sus resultados. A continuación se detallarán cada uno de los paquetes que la forman.

7.2.1 Elementos del modelo

Un conjunto de elementos constituyen la estructura modelada por la herramienta. Los mismos serán quienes interactúen entre sí para la obtención de los resultados finales. La simulación desarrollada será una representación de está interacción, obteniendo y relacionándose con eventos que se produzcan en la misma, arrojando un resultado final de lo obtenido. En el modelo pueden identificarse diferentes elementos, actores, tareas, artefactos y relaciones entre tales.

Los artefactos representan elementos que constituyen la estructura, utilizables por los actores, como por ejemplo, un documento en excel, un ordenador, un informe, etc. Si el objetivo es la representación de una estructura laboral, adicionalmente a los artefactos pueden identificarse actores. En este caso, los actores representarán personas que pueden ser, empleados de la empresa, quienes llevarán consigo la función de resolver tareas. Utilizando estos dos elementos nombrados,

pueden identificarse e incorporarse al modelo relaciones en los mismos, que se encargarán de representar por ejemplo, que un trabajador utilice un ordenador, o una persona lea un informe, que un ordenador necesite un dvd, establecerá relaciones entre recursos, ya sean actores o artefactos.

Faltaría simplemente describir un elemento más en esta estructura, quizás el elemento más complejo de todos, la tarea. Estas serán las cuales deban ser resueltas por lo actores. Las mismas están constituidas por un grupo de características entre las cuales se encuentran dos que marcan la complejidad de la tarea y su importancia. A su vez la tarea puede estar constituida opcionalmente por un actualizador, este será el encargado que una vez resuelta la tarea, se realicen determinadas modificaciones en los recursos, si se cumplen sus filtros relacionados. Junto a estas características también se encuentra un filtro, un filtro de tarea, que determina un grupo de condiciones que deben cumplirse para que se pueda trabajar sobre la tarea. Con este grupo de características y el listado determinado de trabajadores para trabajar sobre la misma, pueden crearse las mismas, en las cuales adicionalmente pueden relacionarse tareas de contingencia que se activen por si la tarea primeria falla.

Mediante todos estos elementos nombrados y su interacción entre sí, se llevará a cabo la simulación producida por la herramienta.

7.2.2 Algoritmos de Planificación (schedulingAlgorithmSystem)

Este sistema contiene los algoritmos de planificación que puede usar el simulador. La clase SchedulignAlgorithm posee el método abstracto *schedule* que recibe como parámetro una lista de procesos y brinda como salida al proceso escogido en la planificación.

Las clases PrioritiesSA y FCFS heredan de SchedulingAlgorithm e implementan, como sus nombres lo indican, un algoritmo por prioridades y el primero en llegar es el primero en salir respectivamente. El usuario del simulador puede heredar clases de SchedulingAlgorithm para probar sus algoritmos, éste es un punto caliente de la aplicación. Mediante el uso de los diferentes tipos de algoritmos de planificación y su relación con el Actor, que será quien use los mismos, puede identificarse el patrón de diseño Strategy. El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente (Actor) puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades. Identificando tal patrón ideal para el uso y selección de los algoritmos de planeamiento.

La clase SAFactory se encarga de asociar una etiqueta a cada algoritmo. En la clase XMLIOSystem del Sistema de Entrada/Salida se utiliza esto en la creación de la lista de tareas nuevas. Por tanto cuando se agrega un algoritmo también se debe agregar una entrada en SAFactory. La etiqueta es usada en los archivos XML que guardan la información de entrada. Actualmente, debido al uso de otra clase que simplifica el almacenamiento, SerialIOSystem, SAFactory y XMLIOSystem son prescindibles.

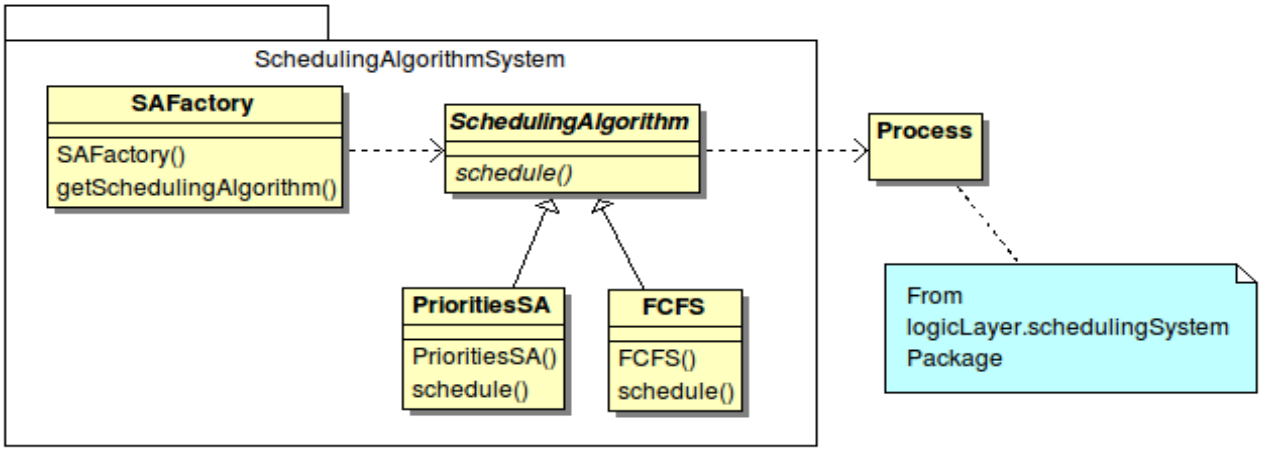


Ilustración 9: Diagrama de Clases, Algoritmos de Planificación

Para usar la clase plantilla, SchedulingAlgorithm, básicamente se debe subclasear de la misma. A manera de ejemplo se mostrará la implementación de la clase FCFS (primero en llegar primero en ser servido):

```
public class FCFS extends SchedulingAlgorithm {
        public FCFS(){}

        public Process schedule(Vector<Process> processes){
                if(processes.size()>0)
                        return processes.get(0);
                else
                        return null;
        }
}
```

7.2.3 Sistema de Planificación (schedulingSystem)

El proceso de simulación debe emular el trabajo paralelo de los actores. Para ello se realizan ciclos de simulación. En cada ciclo cada actor trabaja una fracción de tiempo. Debido a que los actores interactúan entre ellos es necesario que estén sincronizados. Para lograr ese sincronismo el trabajo realizado por cada actor en un ciclo debe tardar la misma cantidad de tiempo t0. El algoritmo desarrollado asume que en cada ciclo un actor puede realizar una de estas acciones:

- Planificar y pasar a estado activo la tarea o interrupción planificado/a.
- Interrupción por temporizador (desalojar la tarea activa y reiniciar temporizador).
- Pasar una tarea al actor que debe realizarla.
- Ejecutar una tarea (una unidad de trabajo).
- Finalizar una tarea.
- No hacer nada.

Cada tarea está compuesta por un conjunto de unidades de trabajo. Cada unidad de trabajo requiere ser ejecutada por un actor específico y tarda una cantidad de tiempo t0.

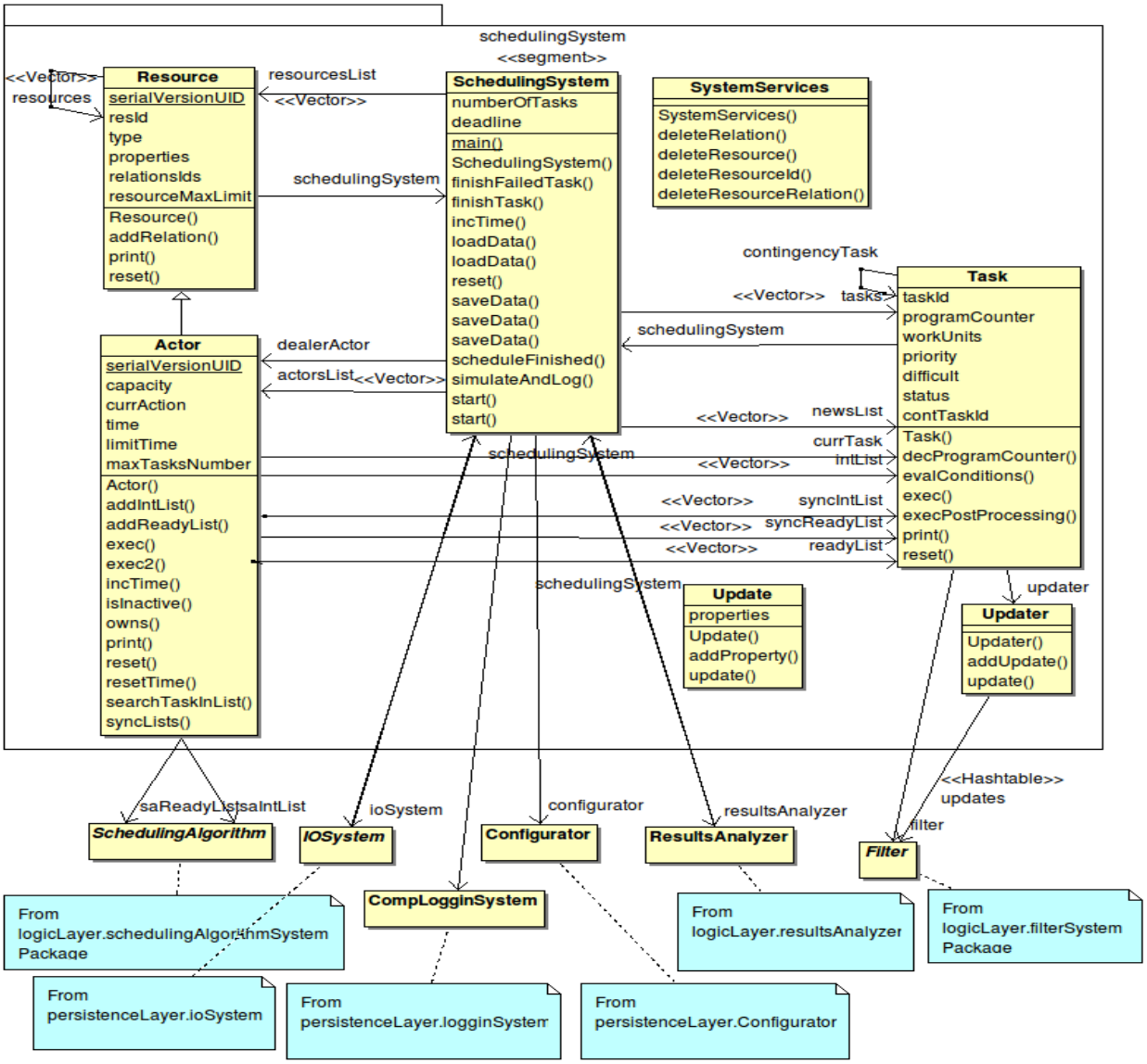

schedulingSystem
<>
Resource
serialVersionUID
resId
type
properties
relationsIds
resourceMaxLimit
Resource()
addRelation()
print()
reset()
<<Vector>>
resources
resourcesList
<<Vector>>
schedulingSystem
SchedulingSystem
numberOfTasks
deadline
main()
SchedulingSystem()
finishFailedTask()
finishTask()
incTime()
loadData()
loadData()
reset()
saveData()
saveData()
saveData()
scheduleFinished()
simulateAndLog()
start()
start()
SystemServices
SystemServices()
deleteRelation()
deleteResource()
deleteResourceId()
deleteResourceRelation()
contingencyTask
Task
<<Vector>>
tasks
taskId
programCounter
workUnits
priority
difficult
status
contTaskId
Task()
decProgramCounter()
evalConditions()
exec()
execPostProcessing()
print()
reset()
Actor
serialVersionUID
capacity
currAction
time
limitTime
maxTasksNumber
Actor()
addIntList()
addReadyList()
exec()
exec2()
incTime()
isInactive()
owns()
print()
reset()
resetTime()
searchTaskInList()
syncLists()
dealerActor
actorsList
<<Vector>>
schedulingSystem
<<Vector>>
newsList
currTask
intList
<<Vector>>
syncIntList
<<Vector>>
syncReadyList
<<Vector>>
readyList
schedulingSystem
Update
properties
Update()
addProperty()
update()
updater
Updater
Updater()
addUpdate()
update()
<<Hashtable>>
updates
filter
saReadyListsaIntList
ioSystem
configurator
resultsAnalyzer
SchedulingAlgorithm
IOSystem
Configurator
ResultsAnalyzer
Filter
From
logicLayer.schedulingAlgorithmSystem
Package
CompLogginSystem
From
logicLayer.resultsAnalyzer
From
logicLayer.filterSystem
Package
From
persistenceLayer.ioSystem
From
persistenceLayer.logginSystem
From
persistenceLayer.Configurator

Ilustración 10: Diagrama de Clases, Sistema de Simulación

La clase SchedulingSystem administra los diferentes sistemas. Es la encargada de realizar los ciclos de simulación y de sincronizar las interacciones entre los actores (por ejemplo que los pasajes de una tarea de una lista a otra se haga al fin del ciclo). La clase Resource (Recurso) posee un conjunto de propiedades dinámicas y relaciones con otros recursos. Lo cual forma un grafo de recursos que puede ser consultado y modificado durante la simulación. La clase Actor, que hereda de Recurso, se encarga de realizar el trabajo de sus tareas. La mayor parte del algoritmo de simulación se encuentra allí. La clase Task (Tarea) contiene la información referente a una tarea, como su conjunto de unidades de trabajo, su contador de programa, su tarea de contingencia y las condiciones que deben cumplirse para que la tarea se ejecute. Por otro lado, la tarea también tiene un Updater (actualizador), cuando la tarea termina, si la condición del actualizador se cumple éste

modifica el grafo de recursos gracias a la clase Update (actualización) que contiene.

El sistema de planificación no está pensado para ser modificado por el usuario del framework.

DIAGRAMA DE SECUENCIA QUE REFLEJE COMO EL SCHEDULING SYSTEM EJECUTA LA SIMULACIÓN, LLAMA AL ACTOR REPETIDAS VECES. COMO SE CHECKEAN LAS CONDICIONES DE LAS TAREAS. Y COMO SE SINCRONIZA EL PASAJE A LAS LISTAS DE LAS TAREAS

OTRO DIAGRAMA REFERENTE A LA EJECUCIÓN DEL UPDATER LUEGO DE QUE UNA TAREA TERMINA EXITOSAMENTE.

7.2.3.1 Algoritmo de Simulación

Para simular el trabajo en paralelo de los actores y sus tareas se utilizará un algoritmo iterativo. Como se dijo antes, en cada ciclo cada actor se ejecuta una fracción de tiempo t0. El algoritmo de simulación se ejecuta mientras haya tareas que deban realizarse y por cada actor. A continuación se muestra una versión en pseudocódigo y simplificada de dicho algoritmo. En azul se establecen las sentencias de control y en verde las ejecuciones:

Simular:

- *Hasta que no haya más tareas por realizar hacer:*
  - *Por cada actor hacer:*
    - *Si no hay una tarea activa*
      - *Si el temporizador ha terminado:*
        - *Se reinicia el temporizador*
      - *Sino, si la lista de interrupciones no está vacía:*
        - *Se ejecuta el algoritmo de planeamiento de interrupciones para elegir una tarea.*
        - *Se elimina dicha tarea de la lista de interrupciones.*
        - *La tarea pasa a estado activo.*
      - *Sino, si la lista de listos no está vacía:*
        - *Se ejecuta el algoritmo de planeamiento para elegir una tarea de la lista de listos.*
        - *Se elimina dicha tarea de la lista de listos.*
        - *La tarea pasa a estado activo.*
      - *Sino:*
        - *No hacer nada*
    - *Sino, si hay una tarea activa*
      - *Si el temporizador ha terminado:*
        - *Se obtiene la unidad de trabajo (que informa cuál es el próximo actor que realizará la tarea)*
        - *Si la tarea que se está realizando no es una interrupción:*
          - *Se la anexa a la lista de listos (pasa a estado listo).*
        - *Se reinicia el temporizador.*
      - *Sino, si la lista de interrupciones no está vacía:*
        - *Se ejecuta el algoritmo de planeamiento de interrupciones para elegir una tarea.*
        - *Si la tarea no es una interrupción o bien sí lo es y además tiene mayor prioridad que la tarea actual*
          - *Se elimina dicha tarea de la lista de interrupciones.*
          - *Se agrega la tarea actual a la lista de listos o a la lista de interrupciones dependiendo de lo que sea .*
          - *La tarea pasa a estado activo.*
        - *Sino*

- *Ir a Simular2*
- *Sino*
  - *Ir a Simular2*
- *Incrementar ciclo*

Simular2:

- *Si el actor actual es el actor de reparto de tareas, o la tarea falla, o no se cumplen las condiciones de la misma:*
  - *Terminar tarea (pasa a estado finalizado en la lista de tareas fallidas).*
- *Sino, si la tarea en ejecución ha llegado a su fin:*
  - *Terminar tarea (pasa a estado finalizado en la lista de tareas exitosas).*
- *Sino, si la tarea en ejecución debe continuar siendo realizada por el actor actual:*
  - *Realizar tarea.*
- *Si es una interrupción:*
  - *Se decrementa el contador de programa*
  - *Se la anexa a la lista de interrupciones del actor indicado (la tarea pasa a ser una interrupción y pasa a estado de espera).*
  - *Se desaloja la tarea en ejecución.*
- *Sino, si la tarea en ejecución debe continuar realizándose por otro actor:*
  - *Se decrementa el contador de programa*
  - *Se la anexa a la lista de listos del recurso indicado (pasa a estado de espera).*
  - *Se desaloja la tarea en ejecución.*

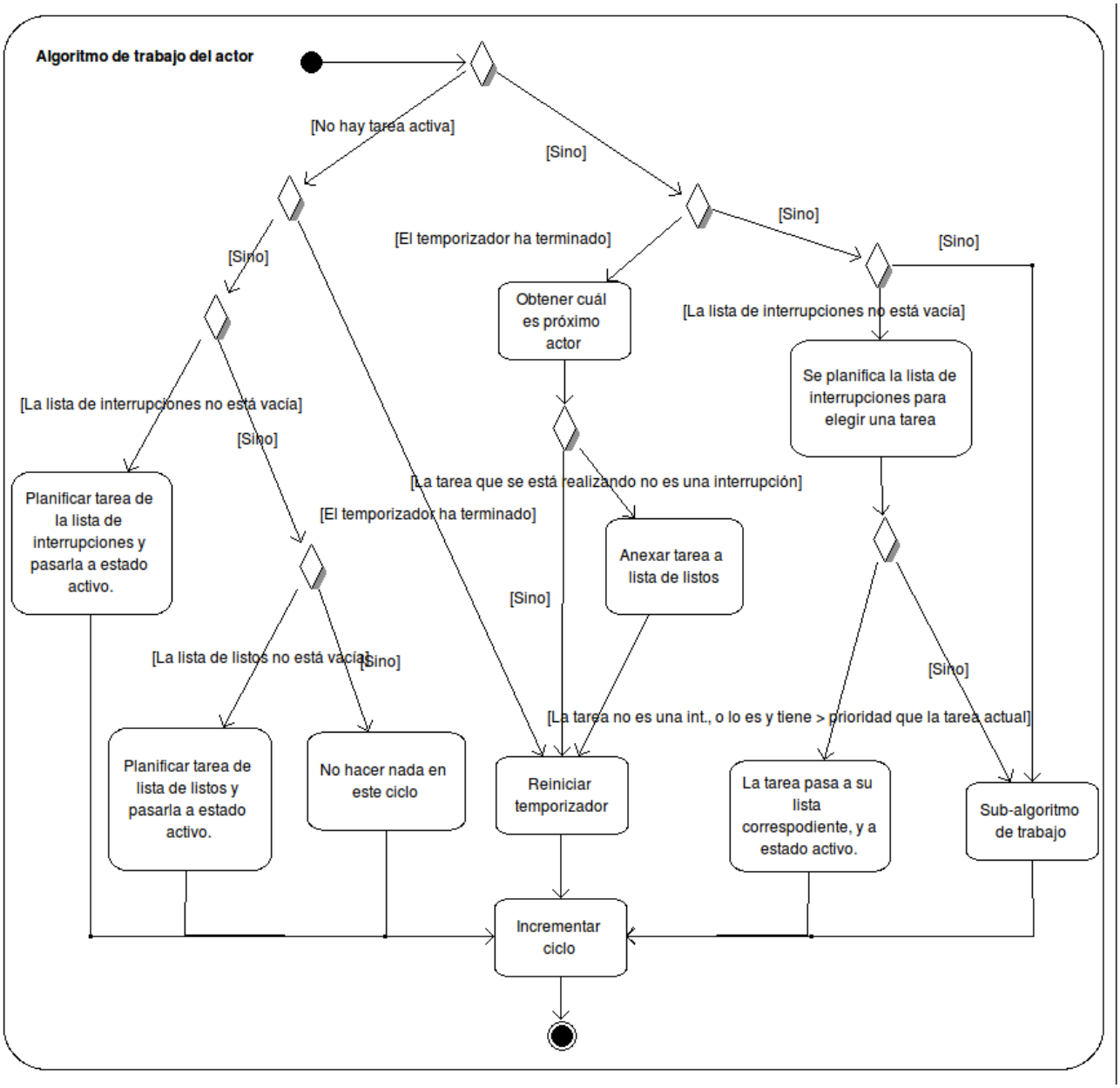


Ilustración 11: Diagrama de actividades: Algoritmo de trabajo del actor

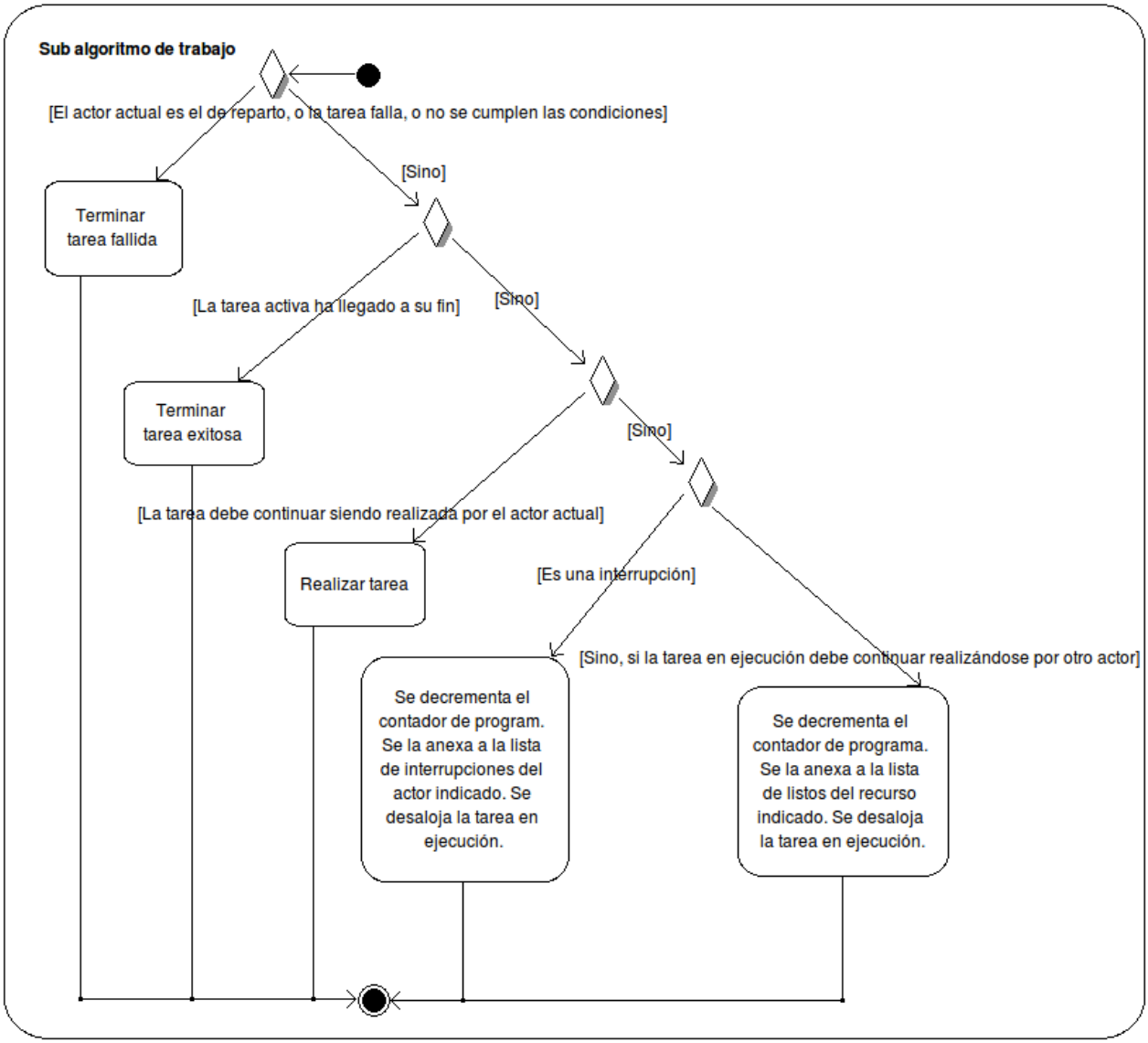


Ilustración 12: Diagrama de actividades: Sub-algoritmo de trabajo del actor

7.3 Capa de Persistencia

Es la capa encargada del almacenamiento de la información de la aplicación para ejecutar una simulación, de la información de los resultados obtenidos y del seguimiento o log de dicha simulación; y por supuesto de la carga de estos datos.

7.3.1 Analizador de Resultados (resultsAnalyzer)

Este paquete contiene la clase abstracta ResultsAnalyzer, ésta es una clase plantilla y el usuario de la aplicación debe subclasear para crear una clase que realice el análisis de los datos obtenidos en la simulación mediante una instancia del SchedulingSystem. Si no se desea realizar esto se puede usar por defecto la clase BasicAnalyzer.

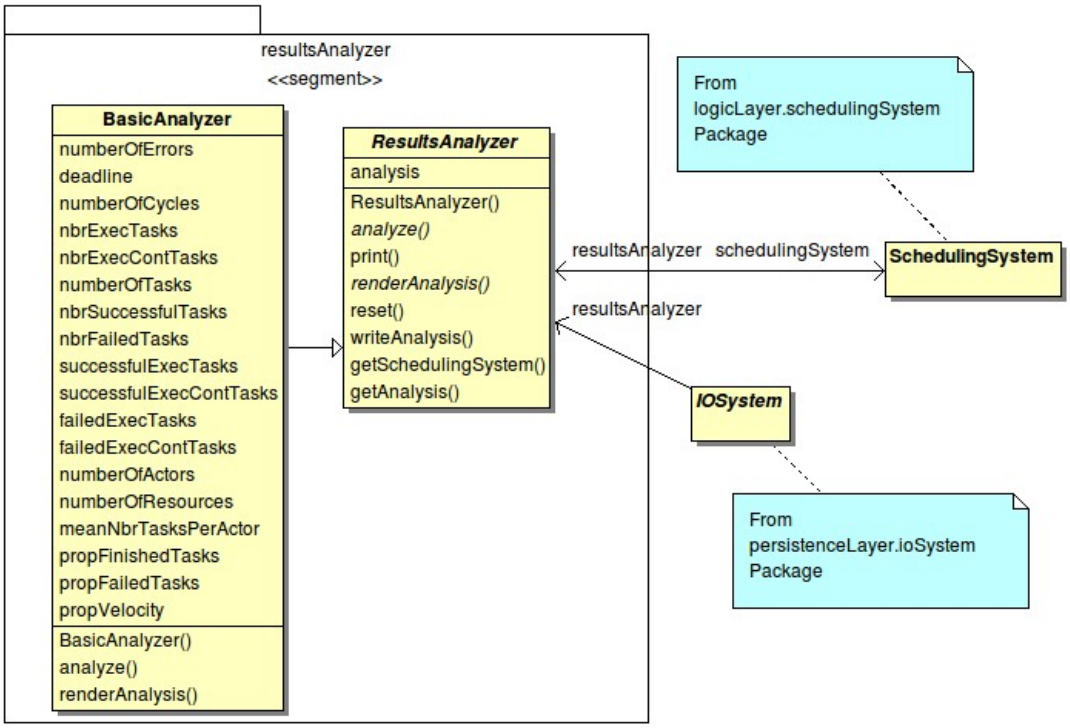

resultsAnalyzer
<>
BasicAnalyzer
numberOfErrors
deadline
numberOfCycles
nbrExecTasks
nbrExecContTasks
numberOfTasks
nbrSuccessfulTasks
nbrFailedTasks
successfulExecTasks
successfulExecContTasks
failedExecTasks
failedExecContTasks
numberOfActors
numberOfResources
meanNbrTasksPerActor
propFinishedTasks
propFailedTasks
propVelocity
BasicAnalyzer()
analyze()
renderAnalysis()
ResultsAnalyzer
analysis
ResultsAnalyzer()
analyze()
print()
renderAnalysis()
reset()
writeAnalysis()
getSchedulingSystem()
getAnalysis()
From
logicLayer.schedulingSystem
Package
resultsAnalyzer
schedulingSystem
SchedulingSystem
resultsAnalyzer
IOSystem
From
persistenceLayer.ioSystem
Package

Ilustración 13: Diagrama de Clases: Analizador de Resultados

La clase BasicAnalyzer genera un archivo con el análisis de los resultados de la simulación. Se pueden encontrar el número de errores, el tiempo que se pretende que dure la simulación y el tiempo real de la misma (medido en ciclos), el número total de tareas convencionales y de contingencia que fueron ejecutadas, el número total de tareas que resultaron satisfactorias y fallidas, la cantidad de actores y el tiempo que cada uno estuvo ocupado realizando sus tareas y planificándolas, el número de artefactos, y algunas proporciones como las de tareas exitosas, fallidas y velocidad de la simulación.

A continuación se muestra un ejemplo de un caso de simulación:

Results Analisys

E = Number of errors: 0

D = Deadline (in cycles): 80

C = Number of cycles: 11

ET = Total number of executed tasks : 1

ECT = Total number of executed contingency tasks: 0

TE = ET + ECT = Total number of executed tasks: 1

ST = S(ET) + S(ECT) = Number of successful tasks : 1

FT = F(ET) + F(ECT) = Number of failed tasks: 0

A = Number of actors: 2

Actor: actor0, busy time: 4

Actor: actor1, busy time: 4

R = Number of other resources (artifacts): 1

TPA = Mean number of tasks per actor: 0.5

Proportions:

Successful tasks : 1.0

Failed tasks : 0.0

Velocity (deadline/nbrOfCycles): 7.2727275

7.3.2 Sistema de Entrada/Salida (ioSystem)

Este sistema es usado por la clase SchedulingSystem para almacenar y cargar los datos concernientes a una simulación, esto es, los resultados obtenidos de dicha simulación, y también los datos necesarios para reproducir la simulación.

La clase IOSystem es una clase plantilla y puede ser subclaseada para implementar la forma que se quiera para almacenar los datos, incluso una base de datos.

Antiguamente se utilizaba la clase XMLIOSystem para realizar estas tareas, sin embargo, debido a que era costosa su modificación, porque el desarrollador, cada vez que agregaba un nuevo elemento a la capa lógica debía también adaptar esta clase, cuyo manejo no era fácil por trabajar con la estructura interna de archivos xml, se optó por otra opción, el almacenamiento serializado de objetos, implementado en la clase SerialIOSystem.

La clase Configurator es necesaria para obtener la dirección del archivo de entrada, salida y el nombre del proyecto.

La clase FileManager es utilizada para almacenar y cargar archivos de texto, principalmente se utiliza en la escritura de los logs.

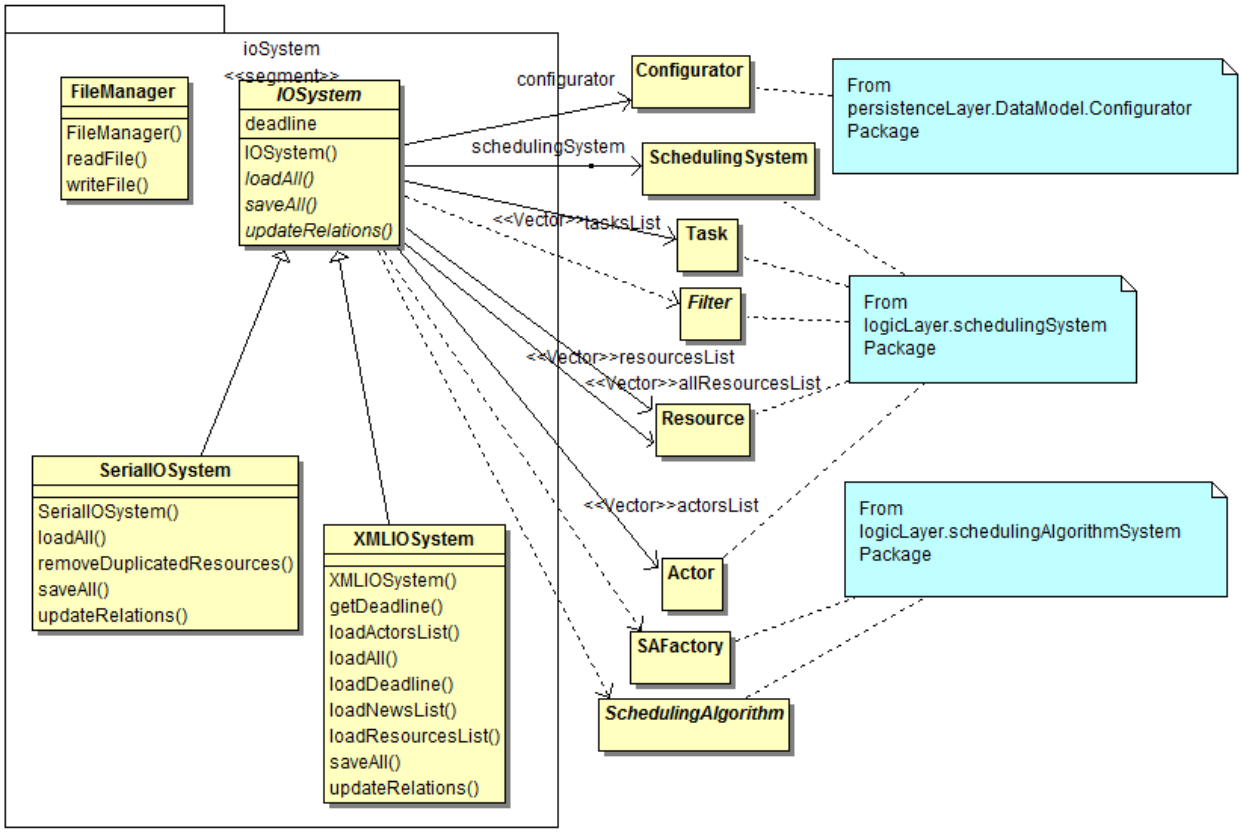

ioSystem
<>
FileManager
FileManager()
readFile()
writeFile()
IOSystem
deadline
IOSystem()
loadAll()
saveAll()
updateRelations()
configurator
Configurator
From
persistenceLayer.DataModel.Configurator
Package
schedulingSystem
SchedulingSystem
<<Vector>>tasksList
Task
Filter
From
logicLayer.schedulingSystem
Package
<<Vector>>resourcesList
<<Vector>>allResourcesList
Resource
SeriallOSystem
SeriallOSystem()
loadAll()
removeDuplicatedResources()
saveAll()
updateRelations()
XMLIOSystem
XMLIOSystem()
getDeadline()
loadActorsList()
loadAll()
loadDeadline()
loadNewsList()
loadResourcesList()
saveAll()
updateRelations()
<<Vector>>actorsList
Actor
From
logicLayer.schedulingAlgorithmSystem
Package
SAFactory
SchedulingAlgorithm

Ilustración 14: Diagrama de Clases, Sistema de Entrada/Salida

7.3.3 Sistema de Logueo (logginSystem)

Este sistema está encargado de registrar todas las actividades que se realizan en cada ciclo de simulación. El Sistema de Planificación usa a este sistema para dicho fin. La información es almacenada en memoria principal hasta que se pide que la misma pase a memoria secundaria. Para realizar esto último el sistema cuenta con más de una codificación disponible, XML y TXT. En caso de ser requerido se pueden agregar nuevos medios de almacenamiento, como una base de datos, por lo que LogginSystem resulta ser una clase plantilla.

La clase LogginSystem es abstracta, tiene implementado el método *log*, que es el que guarda la información de un ciclo en memoria principal. Tiene un método abstracto *writeLog*, que es el que almacena la información en memoria secundaria. La clase FileLogginSystem abstrae el comportamiento para almacenar información en disco en forma de archivos. Las clases XMLLogginSystem y TXTLogginSystem, como es esperado, almacenan la información en estos formatos. Por último la clase CompLogginSystem posee un conjunto de LogginSystem que le permite almacenar en disco la información en varios formatos. En este caso en particular cuando el LogginSystem ejecuta el método *writeLog* de la instancia CompLogginSystem la información que se pasa a disco es en formato XML y TXT. Está última clase mencionada y en el siguiente diagrama se podrá identificar un patrón de diseño utilizado, Composite. El patrón Composite sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la

composición recursiva y a una estructura en forma de árbol. Este patrón es de común identificación en sistemas complejos de clases, siendo identificado de un mismo modo en el sistema de la herramienta.

La clase Configurator es usada para conocer dónde se deben almacenar los resultados. La clase Resource es necesaria para obtener la información del mismo. Finalmente la clase SimulationTime tiene la información de un ciclo de simulación.

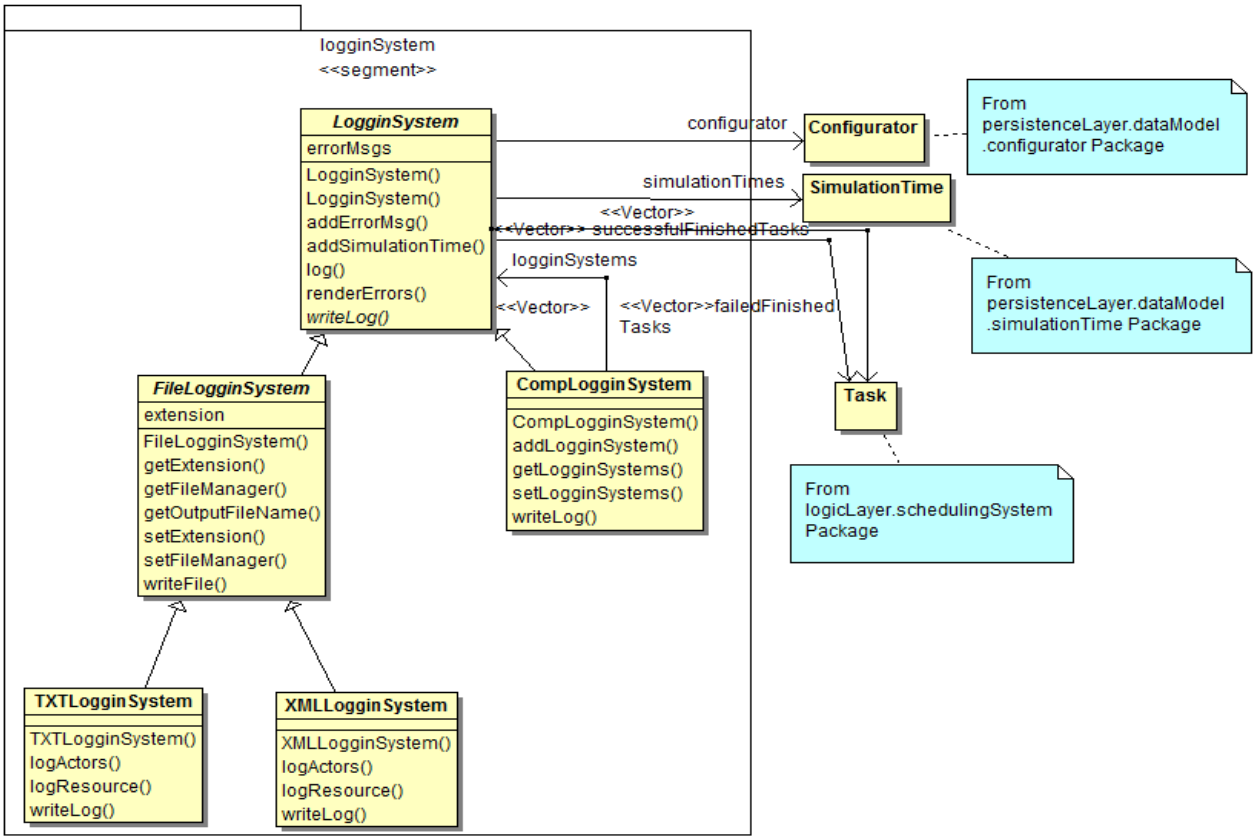

logginSystem
<>
LogginSystem
errorMsgs
LogginSystem()
LogginSystem()
addErrorMsg()
addSimulationTime()
log()
renderErrors()
writeLog()
configurator
Configurator
From
persistenceLayer.dataModel
.configurator Package
simulationTimes
<<Vector>>
SimulationTime
<<Vector>> successfulFinishedTasks
logginSystems
<<Vector>>
<<Vector>>failedFinished
Tasks
From
persistenceLayer.dataModel
.simulationTime Package
FileLogginSystem
extension
FileLogginSystem()
getExtension()
getFileManager()
getOutputFileName()
setExtension()
setFileManager()
writeFile()
CompLogginSystem
CompLogginSystem()
addLogginSystem()
getLogginSystems()
setLogginSystems()
writeLog()
Task
From
logicLayer.schedulingSystem
Package
TXTLogginSystem
TXTLogginSystem()
logActors()
logResource()
writeLog()
XMLLogginSystem
XMLLogginSystem()
logActors()
logResource()
writeLog()

Ilustración 15: Diagrama de Clases, Logueo de Resultados

Como se dijo, se generan dos archivos con distinto formato, XML y TXT, pero con la misma información. El primer archivo está pensado para si se quiere trabajar con esa información, por ejemplo para la realización de estadísticas. El segundo para ser legible si se quiere hacer una lectura de la ejecución. A continuación se mostrará un fragmento del resultado de una ejecución de en TXT:

...

Time: 21

ResourceId: dealerActor

Current Action: None

Active Task: None

Current Time: 21

Limit Time: -1

Interruption List:

Ready List:

ResourceId: actor0

Current Action: Select a task from the ready list and put that task as active. The selected task is task1

Active Task: task1

Current Time: 1

Limit Time: 5

Interruption List:

Ready List:task0

ResourceId: actor1

Current Action: Procesing active task task2

Active Task: task2

Current Time: 21

Limit Time: -1

Interruption List:

Ready List:

Time: 22

…

7.3.4 Modelo de Datos (dataModel)

El modelo de datos representa la información que será almacenada y la información de configuración. Por un lado la clase Configurator tiene la información relacionada con las direcciones de la información de entrada (tareas, actores, artefactos, etc), y de salida (reportes generados en base a la simulación). Por otro lado la clase SimulationTime contiene la información necesaria para almacenar un ciclo de simulación. Cada ciclo almacena el estado de cada Recurso, se hace necesaria la clase SimulationResource. Estas clases dependen de la clase Resource y Task.

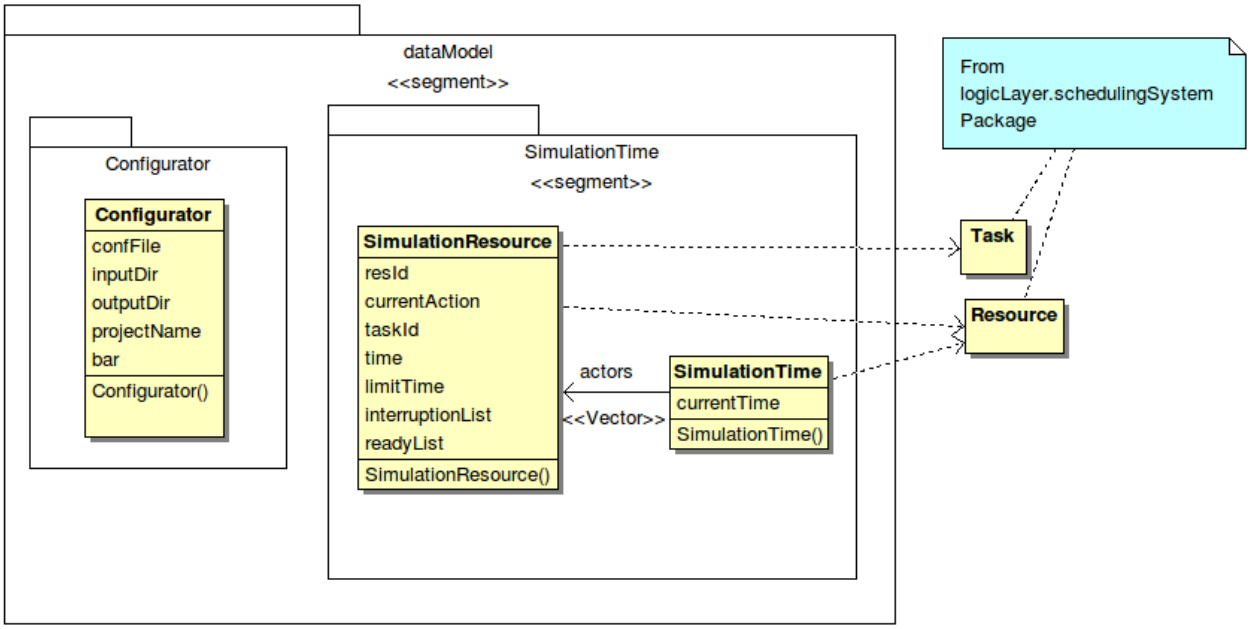

dataModel
<>
Configurator
Configurator
confFile
inputDir
outputDir
projectName
bar
Configurator()
SimulationTime
<>
SimulationResource
resId
currentAction
taskId
time
limitTime
interruptionList
readyList
SimulationResource()
actors
<<Vector>>
SimulationTime
currentTime
SimulationTime()
From
logicLayer.schedulingSystem
Package
Task
Resource

Ilustración 16: Diagrama de Clases: Modelo de Datos

7.3.4.1 Sistema de Filtros (filterSystem)

Los filtros forman parte de la capa lógica. El papel de tales es asistir tanto en la creación de tareas y su opción consecuente de actualización, como en la colaboración para realizar cambios en los nodos del grafo correspondiente a la estructura laboral. Existen una variedad de ellos, los mismos se seleccionarán de acuerdo al objetivo de filtrado que se quiera incorporar.

Entre ellos se encuentran, filtros compuestos, la unión de dos o más filtros (AndFilter) y filtros simples. En la segunda categoría se identifica un mayor conjunto de tales compuesto por, el filtro de igualdad (EqualFilter) que filtra recursos iguales, filtro de propiedades iguales (EqualPropertyFilter) filtrando aquellos elementos con propiedades del mismo tipo y valor, filtro de relaciones (ActorRelationshipFilter) filtrando/identificando relaciones de recursos (actores/artefactos), filtro por posición laboral (JobPositionFilter) encargado de filtrar actores de acuerdo a su puesto laboral y filtro por lista de propiedades (PropertyListFilter) filtrando no solo por una simple propiedad, sino por un conjunto de ellas.

La clase Filter es una clase plantilla para que el usuario del framework pueda realzar sus propios filtros de acuerdo a sus necesidades.

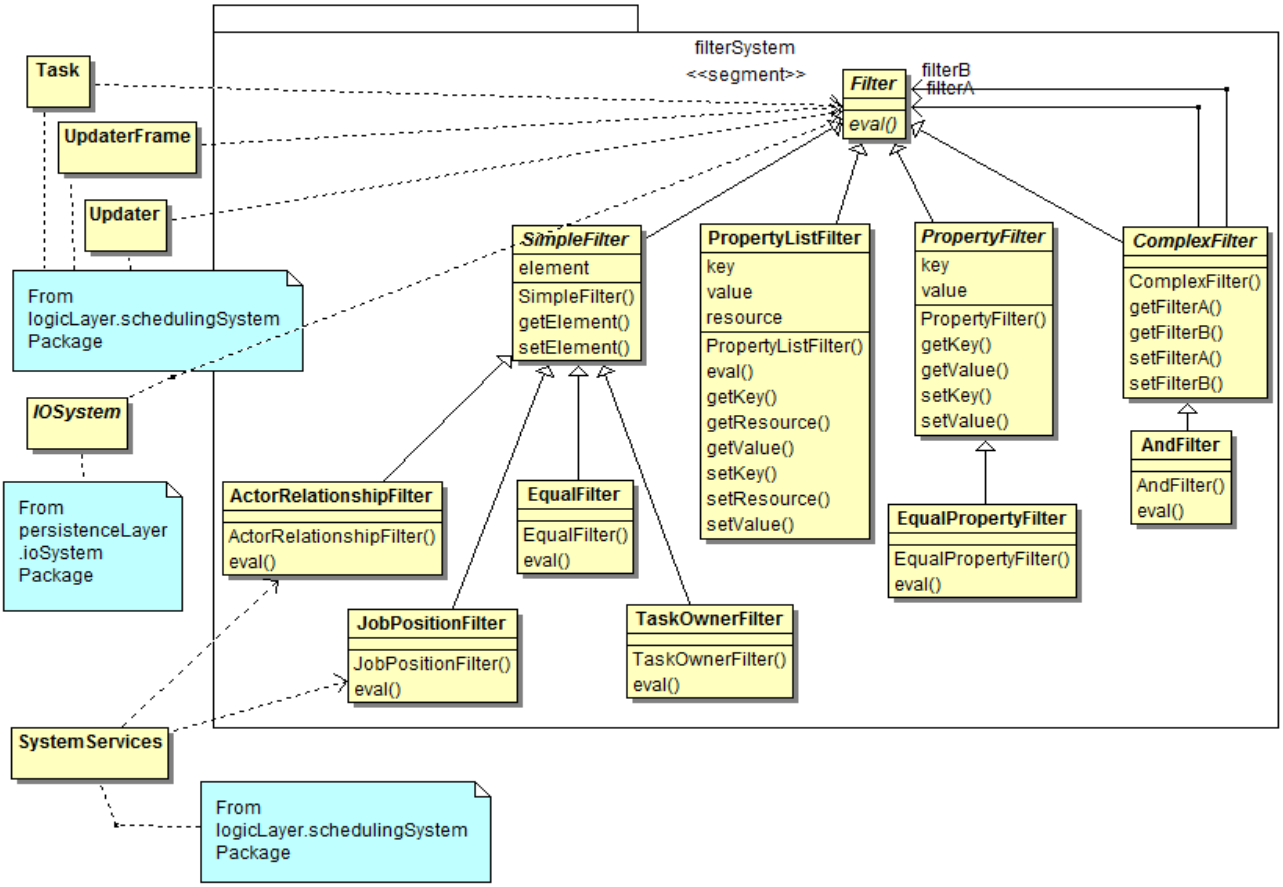

filterSystem
<>
Task
UpdaterFrame
Updater
From
logicLayer.schedulingSystem
Package
IOSystem
From
persistenceLayer
.ioSystem
Package
SystemServices
From
logicLayer.schedulingSystem
Package
Filter
eval()
filterB
filterA
SimpleFilter
element
SimpleFilter()
getElement()
setElement()
PropertyListFilter
key
value
resource
PropertyListFilter()
eval()
getKey()
getResource()
getValue()
setKey()
setResource()
setValue()
PropertyFilter
key
value
PropertyFilter()
getKey()
getValue()
setKey()
setValue()
ComplexFilter
ComplexFilter()
getFilterA()
getFilterB()
setFilterA()
setFilterB()
AndFilter
AndFilter()
eval()
ActorRelationshipFilter
ActorRelationshipFilter()
eval()
EqualFilter
EqualFilter()
eval()
EqualPropertyFilter
EqualPropertyFilter()
eval()
JobPositionFilter
JobPositionFilter()
eval()
TaskOwnerFilter
TaskOwnerFilter()
eval()

Ilustración 17: Diagrama de Clases, Sistema de Filtros

8 Tests y Profiling

8.1 Casos de test

En la ingeniería del software, los casos de prueba o test cases son un conjunto de condiciones o variables bajo las cuáles el analista determinará si el requisito de una aplicación es parcial o completamente satisfactorio. Se pueden realizar muchos casos de prueba para determinar que un requisito es completamente satisfactorio. Para determinar tal estado en la herramienta elaborada, se han incluido una serie de casos de prueba.

8.1.1 Test Case 1

En el primer caso de test, se representa un actor trabajando sobre una tarea. El resultado del correspondiente caso de prueba fue exitoso. Necesitando un número de ciclos inferior al límite para satisfacerlo, realizando la simulación en diez ciclos del total arbitrario permitido.

8.1.2 Test Case 2

El segundo caso representa un actor trabajando en dos tareas de características diferentes. En donde una de ellas posea una dificultad superior a la capacidad del actor para resolverla. El resultado consecuente de su simulación, muestra una tarea resuelta exitosamente y la otra fallida. El número de ciclos utilizado para su simulación fue de un total de once ciclos.

8.1.3 Test Case 3

En esta tercera prueba se modelan un actor, un artefacto y una tarea, sin relaciones existentes entre los mismos. La simulación realizada obtuvo como resultado la resolución de la tarea en cuestión exitosamente, en un número de nueve ciclos.

8.1.4 Test Case 4

La cuarta prueba está formada por dos actores, un artefacto y dos tareas, estableciendo en las tareas mencionadas un trabajo conjunto de los actores para trabajar en una de ellas y la restante sirviendo como tarea de contingencia de la primera nombrada. La simulación arrojo como resultado la exitosa solución de la primera tarea, sin necesidad de recurrir a la tarea de contingencia en un total de once ciclos.

8.1.5 Test Case 5

En el quinto caso de test, se observan dos actores, una tarea, donde la misma tiene una dificultad superior a la capacidad de ambos actores y debe ser resuelta por tales en conjunto. El resultado obtenido posterior a la simulación fue de la tarea resuelta de forma fallida, en un total de cinco ciclos.

8.1.6 Test Case 6

La sexta prueba realizada es representada por un actor y tres tareas, sobrecargando de

trabajo al actor, siendo el número máximo de tareas permitidas para el actor de uno, asignándole las tres tareas simultáneamente. De esta forma una vez realizada la simulación, el resultado muestra la resolución exitosa de una sola tarea, y dos de ellas fallidas, reportando los errores que ocurrieron en tales. Donde los mismos son, como se menciono, que el actor no pudo trabajar en esas dos tareas restantes debido a que el máximo de tareas permitidas en el mismo era inferior al asignado. El número total de ciclos utilizado fue de nueve.

8.1.7 Test Case 7

En el último caso de test, el caso de test más complejo, se representaron tres tareas, dos artefactos y dos actores. Adicionalmente se estableció una tarea de contingencia para una de ellas, filtros y actualizaciones en las mismas, dos artefactos donde el primero de tales tiene una relación con el segundo y el segundo una relación con dos actores, y por ultimo uno de los actores posee relación con un artefacto y con el otro actor. Con esta estructura creada se procedió a la simulación obteniendo como resultado en un número de cincuenta y siete ciclos, tres tareas resueltas exitosamente y ninguna fallida.

8.2 Profiling

En ingeniería de software el análisis de rendimiento, comúnmente llamado profiling, es la investigación del comportamiento de un programa de computadora usando información reunida del análisis dinámico del programa (cuando este esta corriendo) en oposición al análisis estático (análisis de código). La meta del análisis de rendimiento es determinar que partes del programa se pueden optimizar para ganar velocidad u optimizar el uso de memoria.

Usualmente el análisis de rendimiento tiene una gran importancia aunque (como muchas otras practicas importantes) no se realiza con la seriedad necesaria. Se suele decir que el profiling tiene que ocupar un 90% del tiempo del desarrollo de una aplicación. [WKP2]

Debido a que la herramienta usada para los diagramas de secuencia también puede proveer información del profiling se la usó para dicho fin.

8.2.1 Análisis de Memorias

Este análisis se focaliza en las instancias de las clases de la aplicación:

- Live instances (instancias vivas): el número de instancias que están vivas (por ejemplo que no fueron recogidas por el recolector de basura)
- Total instances (total de instancias): el número total de instancias de la clase.
- Active Size (bytes) (tamaño activo): el tamaño de las Live instances.
- Total Size (bytes) (tamaño total): el tamaño de Total instances
- Age (edad): número de veces que una instancia sobrevive al recolector de basura.
- Average age (edad promedio): es la suma de las edades / número de instancias.

Class Name	Package	Live Instances	Active Size (bytes)	Total Instances	Total Size (bytes)	Avg. Age
SimulationResource	persistenceLayer.dataModel.SimulationTime	1368	54720	1881	75240	0,03
SimulationTime	persistenceLayer.dataModel.SimulationTime	456	7296	675	10800	0,03
FCFS	logicLayer.schedulingAlgorithmSystem	37	592	121	1936	0,04
Task	logicLayer.schedulingSystem	28	1568	60	3360	0,07
Actor	logicLayer.schedulingSystem	22	1936	71	6248	0,07
FileManager	persistenceLayer.ioSystem	18	144	78	624	0,05
Resource	logicLayer.schedulingSystem	14	560	21	840	0,1
TXTLogginSystem	persistenceLayer.logginSystem	9	360	39	1560	0,05
XMLLogginSystem	persistenceLayer.logginSystem	9	360	39	1560	0,05
CompLogginSystem	persistenceLayer.logginSystem	9	360	39	1560	0,08
ResultsAnalyzer	logicLayer.resultsAnalyzer	9	792	39	3432	0,05
Configurator	persistenceLayer.dataModel.Configurator	8	256	32	1024	0,09
SchedulingSystem	logicLayer.schedulingSystem	8	384	32	1536	0,09
XMLIOSystem	persistenceLayer.ioSystem	7	280	25	1000	0,04
PrioritiesSA	logicLayer.schedulingAlgorithmSystem	7	112	21	336	0,05
SerialIOSystem	persistenceLayer.ioSystem	1	40	7	280	0,29

Ilustración 18: Análisis de Memorias

8.2.2 Análisis de Threads

El análisis de threads (hilos) se usa para identificar y corregir cualquier número de problemas relacionados con threads en la aplicación. Para threads de interés específico, se pueden usar herramientas de profiling para saber que tan seguido estos threads son bloqueados y por quien, y visualizar las características de los threads de la aplicación.

En el caso de este programa sólo se cuenta con un solo thread (main), sin embargo en la información a continuación se pueden ver otros que corresponden principalmente a java.lang, este paquete provee clases que son fundamentales para el diseño del lenguaje de programación Java.[JL]

Thread Name	Class Name	State	Running Time	Waiting Time	Blocked Time	Block Count
Reference Handler	java.lang.ref.Reference$ReferenceHandler	Stopped	00:01:279	00:02:131		
Finalizer	java.lang.ref.Finalizer$FinalizerThread	Stopped	00:01:280	00:02:120		
Signal Dispatcher	java.lang.Thread	Stopped	00:03:313			
main	java.lang.Thread	Stopped	00:03:296			
ReaderThread	org.eclipse.jdt.internal.junit.runner.RemoteTestRunner$ReaderThread	Stopped	00:03:166		00:00:000	1

Ilustración 19: Análisis de threads 1

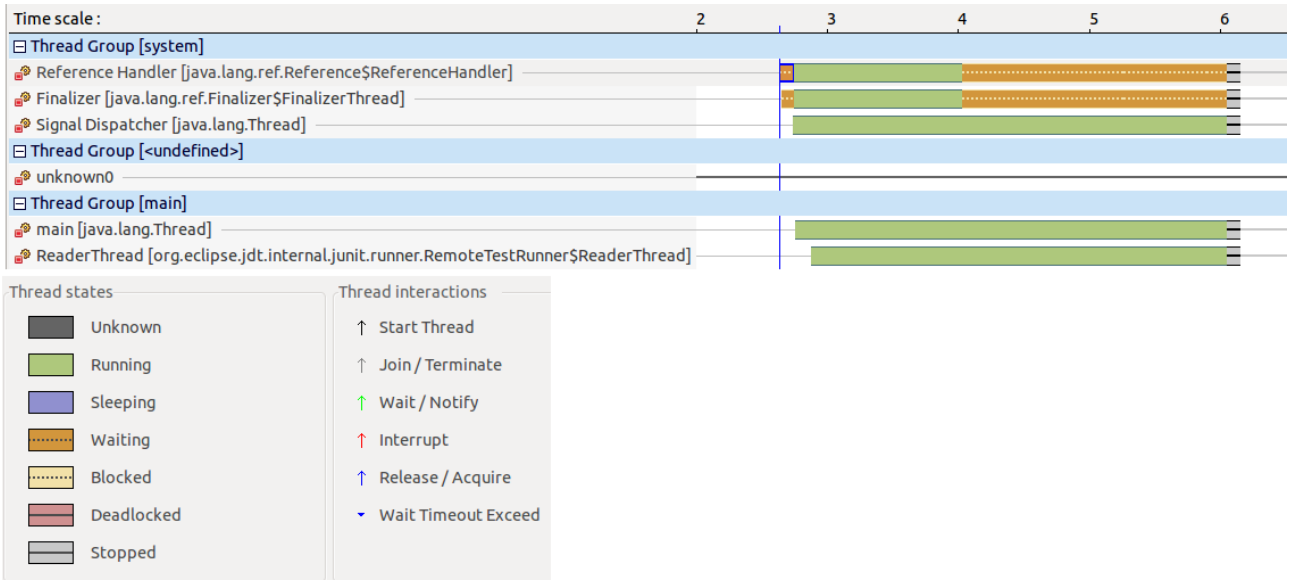


Ilustración 20: Análisis de threads 1

8.2.3 Análisis de Tiempos

El siguiente análisis de tiempos se hace por paquetes, por clases y por métodos. En este caso se mostrarán dicho análisis por paquetes y por clases, ya que por método ocuparía demasiado espacio en este informe.

- Base Time (tiempo base): la cantidad de tiempo en segundos que al método le toma ejecutarse. No se incluye el tiempo de ejecución de ninguno otro método llamado desde este método.
- Average base time (tiempo base promedio): el tiempo base promedio requerido para ejecutar este método una vez.
- Cumulative base time (tiempo base acumulado): la cantidad de tiempo en segundos que a este método le tomó ejecutarse. Incluyendo el tiempo de ejecución de cualquier otro método llamado desde este método.
- Calls (llamadas): número de veces que este método fue invocado.

Package	Base Time (seconds)	Average Base Time (seconds)	Cumulative Time (seconds)	Calls
[-] persistenceLayer.ioSystem	3,177469	0,016212	3,18937	196
[+] SerialIOSystem	3,165255	0,150726	3,186102	21
[+] IOSystem	0,010568	0,000072	0,010568	147
[+] FileManager	0,001646	0,000059	0,001646	28
[-] logicLayer.schedulingSystem	2,980895	0,000123	7,591528	24256
[+] Actor	2,487216	0,000119	2,657744	20835
[+] SchedulingSystem	0,310262	0,00033	7,591528	939
[+] Task	0,120905	0,000093	0,127344	1302
[+] Resource	0,062513	0,000053	0,062513	1180
[-] persistenceLayer.dataModel.Configurator	0,805531	0,00822	0,805531	98
[+] Configurator	0,805531	0,00822	0,805531	98
[-] persistenceLayer.dataModel.SimulationTime	0,467695	0,000172	0,621892	2712
[+] SimulationResource	0,405657	0,000171	0,559853	2376
[+] SimulationTime	0,062039	0,000185	0,621892	336
[-] persistenceLayer.logginSystem	0,095331	0,000142	0,718869	671
[+] LogginSystem	0,065002	0,000129	0,686894	503
[+] CompLogginSystem	0,014676	0,000262	0,05723	56
[+] FileLogginSystem	0,012942	0,000154	0,029353	84
[+] TXTLogginSystem	0,00139	0,000099	0,022196	14
[+] XMLLogginSystem	0,001322	0,000094	0,009868	14
[-] logicLayer.resultsAnalyzer	0,058103	0,000177	0,065352	329
[+] ResultsAnalyzer	0,058103	0,000177	0,065352	329
[-] (default package)	0,044921	0,000802	7,638236	56
[+] SchedulingSystemTest	0,044921	0,000802	7,638236	56
[-] logicLayer.schedulingAlgorithmSystem	0,007187	0,000081	0,008223	89
[+] FCFS	0,00273	0,000065	0,004706	42
[+] PrioritiesSA	0,002481	0,000131	0,003517	19
[+] SchedulingAlgorithm	0,001976	0,000071	0,001976	28
[-] junit.framework	0,001104	0,000079	0,001104	14
[+] Assert	0,001104	0,000079	0,001104	14

Ilustración 21: Análisis de Tiempos

9 Herramientas

Lenguaje:

Java

IDEs:

NetBeans IDE 7.0.1. http://netbeans.org/

Eclipse IDE. http://www.eclipse.org/

Testing:

JUnit. http://www.junit.org/

UML:

BOUML, Diagramas UML y reingeniería. http://bouml.free.fr/

Profiling y Diagramas de Secuencia:

Eclipse Test & Performance Tools Platform Project. http://www.eclipse.org/tptp/

Documentación:

LibreOffice. http://www.libreoffice.org/

MicrosoftOffice. http://office.microsoft.com/es-es/

Clientes SVN:

TortoiseSVN. http://tortoisesvn.tigris.org/

RapidSVN. http://www.rapidsvn.org/

Servidor SVN, Issues, Wiki, etc.:

Google Code. http://code.google.com/intl/es-AR/

10 Anexo

Existen diferentes puntos no descriptos anteriormente que se refieren a cómo utilizar la herramienta descripta. Si bien tales puntos son desarrollados ampliamente en el manual de usuario incorporado en la aplicación, igualmente a continuación se nombrarán de forma breve algunos de ellos para ampliar el conocimiento sobre los mismos.

Inicialmente, una vez ejecutada la aplicación el usuario tiene la opción en la pestaña “Main” de cargar un proyecto, es importante que al seleccionar el archivo correspondiente a su cargado el mismo sea una carpeta, ya que la aplicación salva los modelos producidos como un conjunto de archivos dentro de una carpeta con el nombre del modelo desarrollado. De esta manera pueden cargarse los datos de entrada, sin necesidad de ingresarlos uno por uno. De lo contrario, hay que crear los diferentes elementos pertenecientes a la estructura labora en la pestaña “Set”.

Dentro de ella, pueden crearse artefactos, actores, tareas, relaciones entre recursos (artefactos/actores) y setear el nombre del modelo laboral que se está creando. Al igual que los botones para creación existen a sus costados aquellos correspondientes para la eliminación de elementos de la estructura de acuerdo su tipo. Adicionalmente en tal pestaña se pueden observar la cantidad de elementos creados y el botón correspondiente a la ejecución de la simulación de tareas en el modelo creado.

Si bien no existe un orden especifico para crear los diferentes elementos, se aconseja crear en un principio los artefactos y actores, y posteriormente las tareas y relaciones. Al crear los artefactos deben configurarse en su ventana el número máximo de relaciones que puede tener el artefacto que se está creando y sus diferentes propiedades. Por otro lado al crear un actor, en el frame deben setearse el número máximo de tareas y de relaciones que tendrá un actor, seleccionar el algoritmo de planeamiento que se desea para elegir las tareas a trabajar, la capacidad que el actor tendrá para resolver tareas (Capacidad de trabajo), el quantum (Tiempo límite que una tarea puede ejecutarse de corrido para que no haya envejecimiento), y por último al igual que con los artefactos el seteo de propiedades extras en la tabla de las mismas.

Una vez creados este tipo de elementos pueden crearse de una forma más amena las tareas, o configurar las relaciones entre actores y artefactos que se deseen en su ventana correspondiente. A la hora de crear tareas, su creación puede decirse que es la de mayor complejidad de todas. En tal se deben configurar el número de dificultad de resolución que posee tal tarea y la prioridad que tiene la misma y marcarla en sus campos. Posteriormente se puede crear opcionalmente, un updater (actualizador). Este elemento se encarga de actualizar propiedades de algún recurso específico una vez que se concluye la tarea que se está creando. Por ejemplo, si un actor está trabajando sobre un artefacto, incorporando nueva información en el mismo, una vez que se concluya tal trabajo quizás se deba actualizar/incorporar nuevas propiedades al documento u a otro recurso por ello existirá el updater, para ocuparse de tal acción.

Es importante mencionar que se relaciona tal tarea con un filtro, “Task Filter”. Este será el encargado de determinar si se puede trabajar en la tarea, siempre y cuando se cumplan las condiciones señaladas por tal filtro.

Es importante repetir que el uso de estos elementos, updater y filtro, es opcional pero se recomienda su uso para actualizar constantemente la información de los nodos del grafo de componentes, siempre y cuando se lo requiera. Estos elementos, junto a esta funcionalidad, fueron incorporadas para poseer una opción más a la hora de mantener una alta similitud con el comportamiento de una estructura laboral real.

Al momento de crear el updater, existirán diferentes ventanas en las cuales deberemos introducirse datos extra. Al comenzar con su creación, inicialmente se observará la ventana para incorporar un filtro y un update, actualización, relacionada al mismo. Con esto podremos crear la funcionalidad respecto a realizar modificaciones en el grafo, en caso de que se cumpla tal filtro relacionado. En la creación del filtro de acuerdo al tipo del mismo que se seleccione, se mostrarán habilitados diferentes campos en los cuales se deberán introducir los valores de comparación correspondientes. En cuanto al update, es simplemente una tabla donde se especificaran las propiedades y valores a actualizar en consecuencia de cumplir con las condiciones configuradas. Con estos dos elementos mencionados, las actualizaciones incluidas serán realizadas cuando termine la tarea, siempre y cuando se cumplan las condiciones estipuladas en su filtro relacionado.

Luego de completar los campos y elementos nombrados restaría simplemente especificar el orden de los trabajadores encargados de trabajar en dicha tarea, con esto se terminarían de completar por completo sus cualidades y podrá crearse correctamente.

Con lo descripto hasta el momento en esta sección se han creado la totalidad de los elementos, pero resta describir funcionalidades y pestañas de la herramienta correspondiente por ejemplo a la configuración de tareas de contingencia. Esta funcionalidad estará dentro de la pestaña “Set Contingencia” donde simplemente se deberá seleccionar la tarea principal que tendrá como tarea de contingencia la seleccionada en el combo box inferior. La herramienta provee la funcionalidad de realizar cambios sobre lo que ya se ha creado, tal funcionalidad puede encontrarse en la pestaña “Modification”. En caso de que se quiera incorporar nuevas propiedades en algún recurso o eliminar una relación creada, se pueden realizar tales acciones desde esa pestaña.

Por último queda la última pestaña y no por ello, menos importante, que es la correspondiente a reflejar los resultados obtenidos una vez terminada la simulación. Con este conjunto de elementos gráficos y funcionalidad se conforma la interfaz y la lógica de la aplicación creada.

11 Bibliografía

[DSS07] Filminas 2007. Materia Diseño de Sistemas de Software. UNICEN.

[DSS11] Filminas 2011. Materia Diseño de Sistemas de Software. UNICEN.

https://sites.google.com/a/alumnos.exa.unicen.edu.ar/disenio/

[JL] Artículo del Paquete java.lang

http://docs.oracle.com/javase/6/docs/api/java/lang/package-summary.html

[POSA] Pattern-Oriented Software Architecture.

Volume 2: Patterns for Concurrent and Networked Objects. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sornmerlad, Michael Stal of Siemens AG, Germany.

[SO] Sistemas Operativos. 6a. Edición. Silberschatz, Galvin, Gagne.

[SAIP] Software Architecture in Practice, Second Edition.

Por Len Bass, Paul Clements, Rick Kazman. Editor: Addison Wesley

[WKP1] Organización. Artículo Wikipedia.

http://es.wikipedia.org/wiki/Organizaci%C3%B3n

[WKP2] Profiling, Artículo de Wikipedia.

http://es.wikipedia.org/wiki/Profiling