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

1 Información General....2
2 Índice....2
2.1 Índice de contenido....2
2.2 Índice de ilustraciones....3
3 Objetivos....4
4 Requerimientos....4
4.1 Contexto....4
4.2 Requerimientos funcionales....5
4.3 Restricciones impuestas por el contexto o por los stakeholders....7
5 Arquitectura de Software....9
6 Diseño Detallado....10
6.1 Capa de Presentación (presentationLayer)....10
6.1.1 Frames....11
6.1.2 Media....13
6.2 Capa Lógica (logicLayer)....13
6.2.1 Algoritmos de Planificación (schedulingAlgorithmSystem)....13
6.2.2 Paquete del Sistema de Planificación (schedulingSystem)....14
6.2.2.1 Algoritmo de Simulación....15
6.3 Capa de Persistencia....17
6.3.1 Analizador de Resultados (resultsAnalizer)....17
6.3.2 Sistema de Entrada/Salida (ioSystem)....18
6.3.3 Sistema de Logueo (logginSystem)....18

6.3.4 Paquete del Modelo de Datos (dataModel).....20
6.3.4.1 Sistema de Filtros (filterSystem).....20
7 Uso del Framework.....22
7.1 Acerca de Frameworks.....22
7.2 Crear un algoritmo de planificación.....23
7.3 Leer Datos de Salida.....23
8 Casos de test.....24
8.1 Test Case 1.....25
8.2 Test Case 2.....25
8.3 Test Case 3.....25
8.4 Test Case 4.....25
8.5 Test Case 5.....25
8.6 Test Case 6.....25
8.7 Test Case 7.....26
9 Bibliografía.....26

2.2 Índice de ilustraciones

Ilustración 1: Diagrama de Casos de Uso General.....5
Ilustración 2: Explosión del Caso de Uso: Start Simulation.....7
Ilustración 3: Arquitectura por Capas.....10
Ilustración 4: Explosión de los paquetes de las capas.....12
Ilustración 5: Diagrama de secuencia: interacción entre el usuario y el simulador.....14
Ilustración 6: Diagrama de clases. Interacción SimulatorFrame y clases de capa lógica.....15
Ilustración 7: Diagrama de Clases, Algoritmos de Planificación.....16
Ilustración 8: Diagrama de Clases, Sistema de Simulación.....17
Ilustración 9: Diagrama de Clases, Analizador de Resultados.....19
Ilustración 10: Diagrama de Clases, Sistema de Entrada/Salida.....20
Ilustración 11: Diagrama de Clases, Logueo de Resultados.....24
Ilustración 12: Diagrama de Clases, Sistema de Filtros.....25
Ilustración 13: Conceptos Básicos de Frameworks.....26
Ilustración 14: Métodos y Clases Plantilla y Gancho.....27

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.

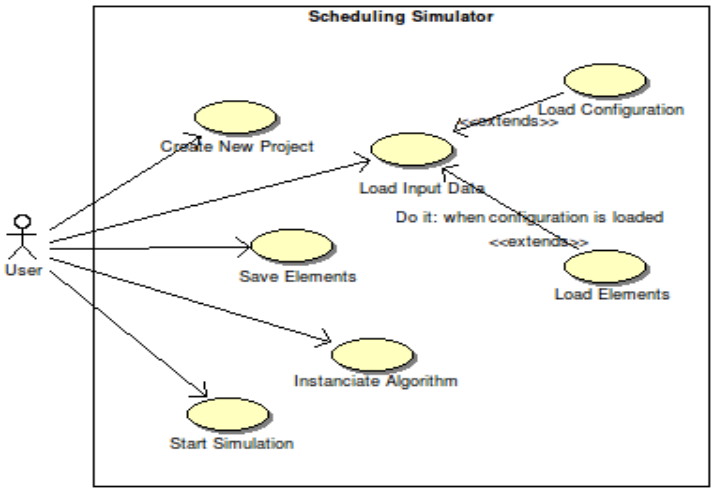


Ilustración 1: Diagrama de Casos de Uso General

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).

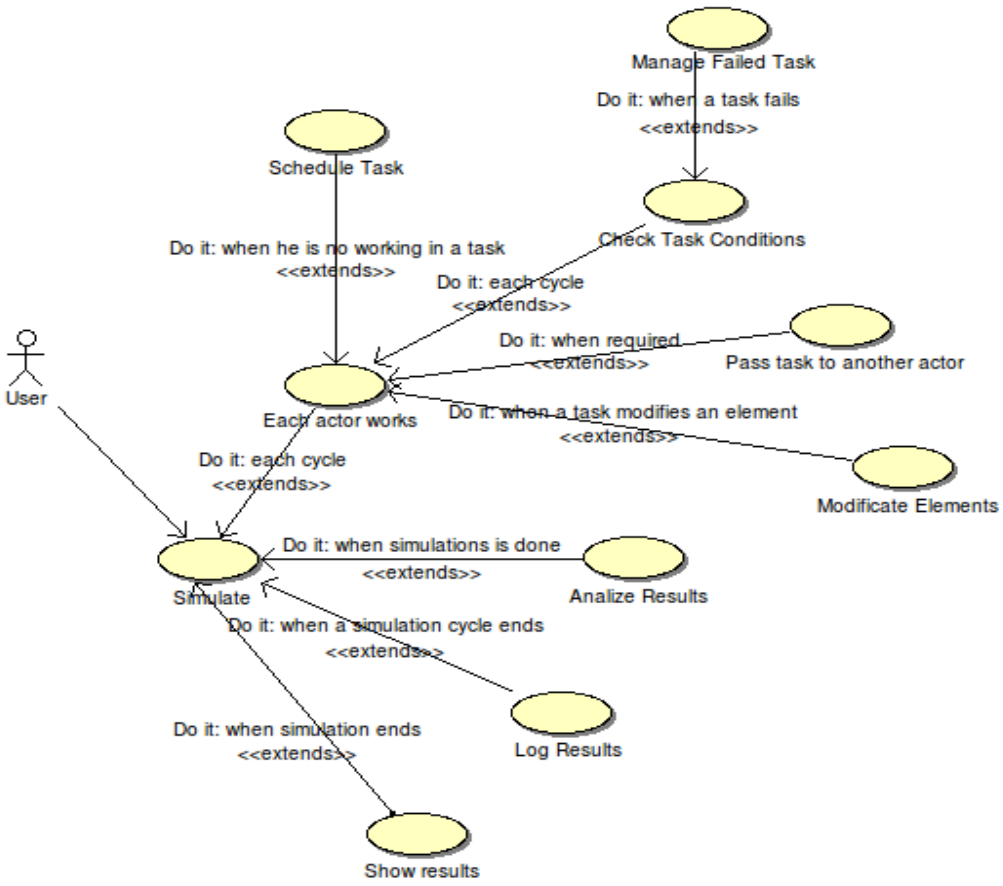
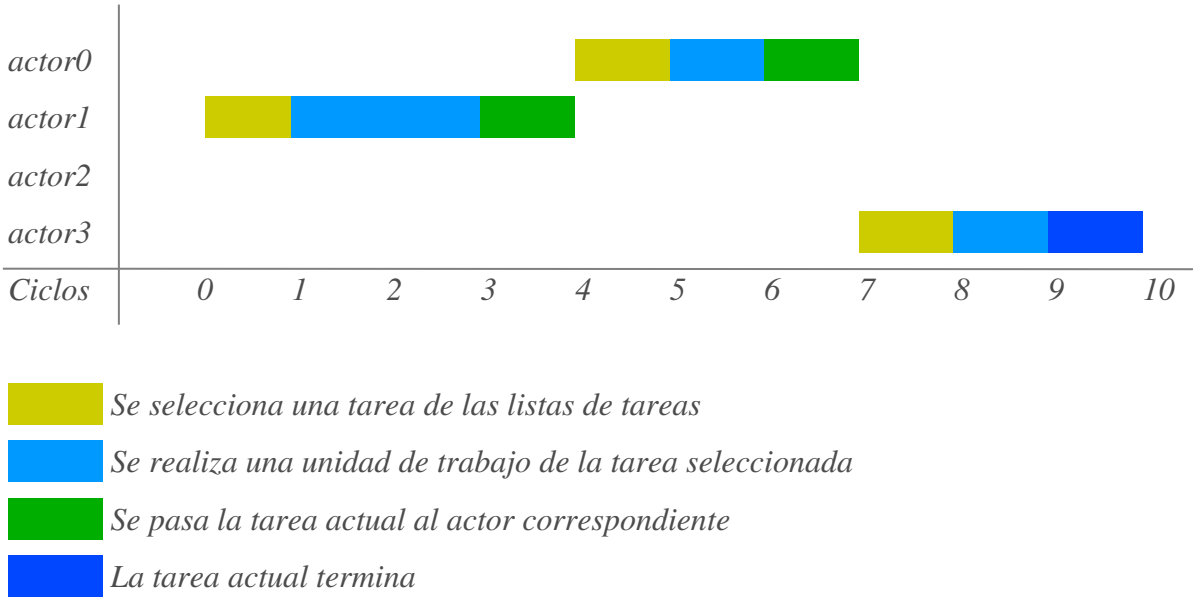


Ilustración 2: Explosión del Caso de Uso: Start Simulation

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.

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]

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.

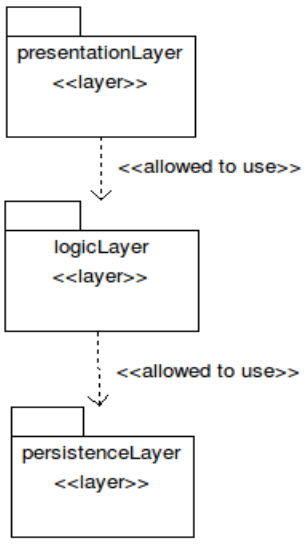


Ilustración 3: Arquitectura por Capas

En el actual proyecto 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.

6 Diseño Detallado

En esta sección se desarrollarán las capas mostradas anteriormente con los segmentos que

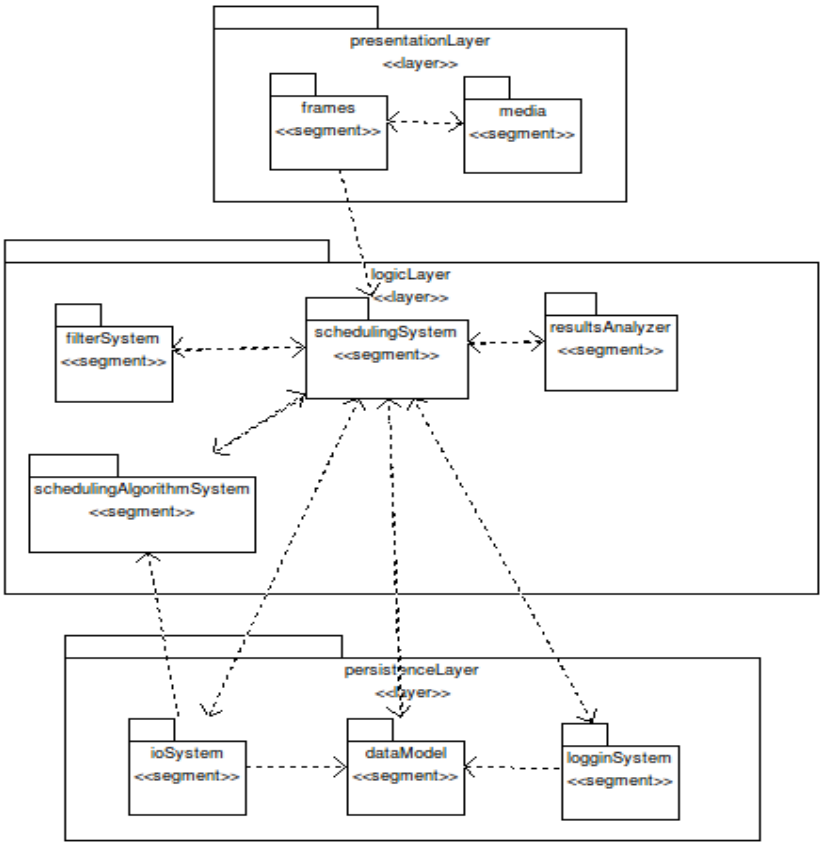

presentationLayer
<<layer>>
frames
<>
media
<>
logicLayer
<<layer>>
filterSystem
<>
schedulingSystem
<>
resultsAnalyzer
<>
schedulingAlgorithmSystem
<>
persistenceLayer
<<layer>>
ioSystem
<>
dataModel
<>
logginSystem
<>

Ilustración 4: Explosión de los paquetes de las capas

las componen y las relaciones entre los mismos.

6.1 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.

6.1.1 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.

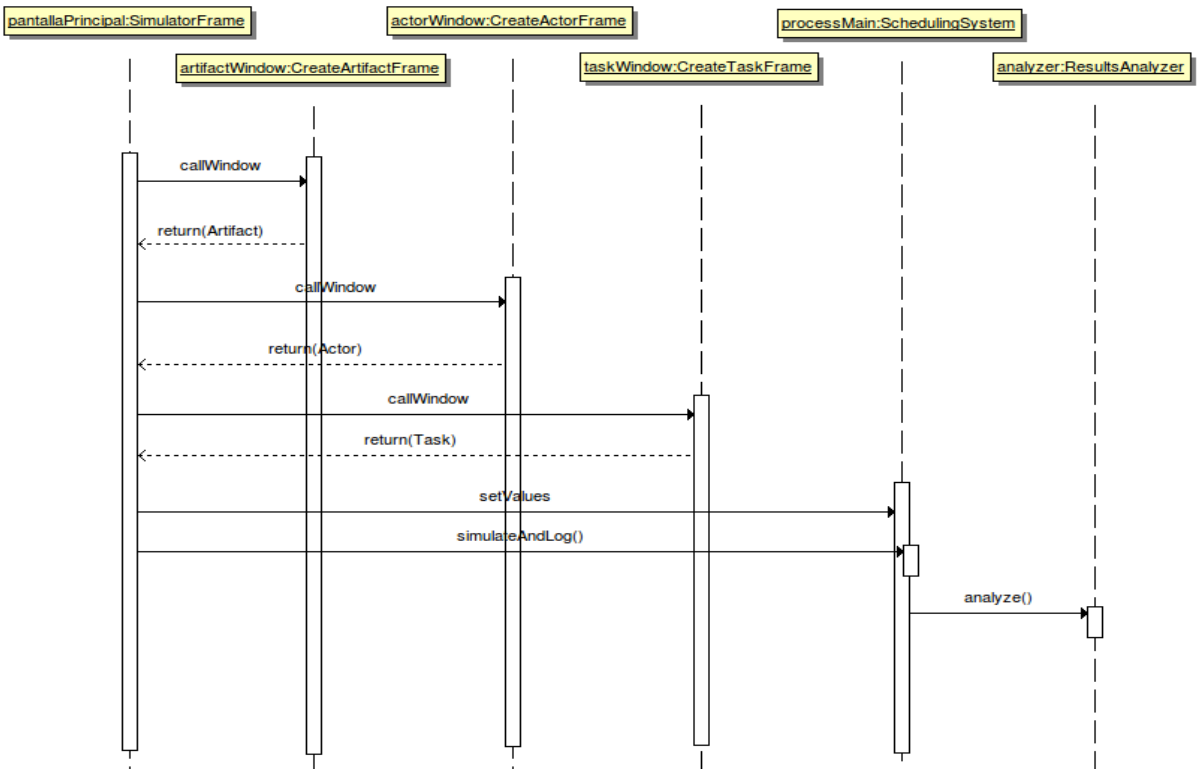


Ilustración 5: Diagrama de secuencia: interacción entre el usuario y el simulador

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 los frames, salvo con los filtros que no ocurre tal caso y por ello no se adopto tal patrón para su correspondiente frame. 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.

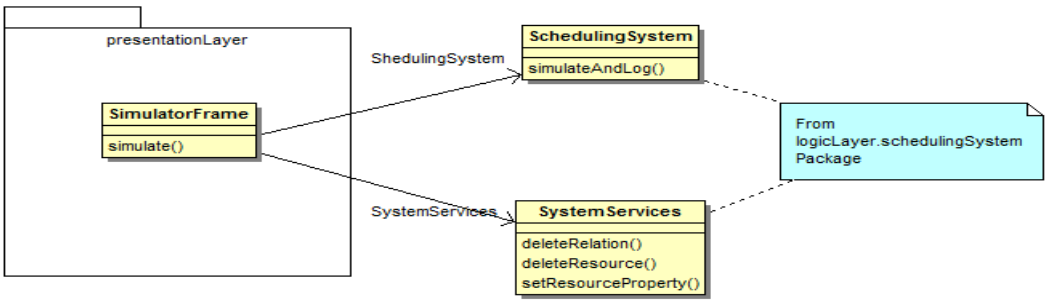


Ilustración 6: Diagrama de clases. Interacción SimulatorFrame y clases de capa lógica

6.1.2 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.

6.2 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.

6.2.1 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. 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.

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.

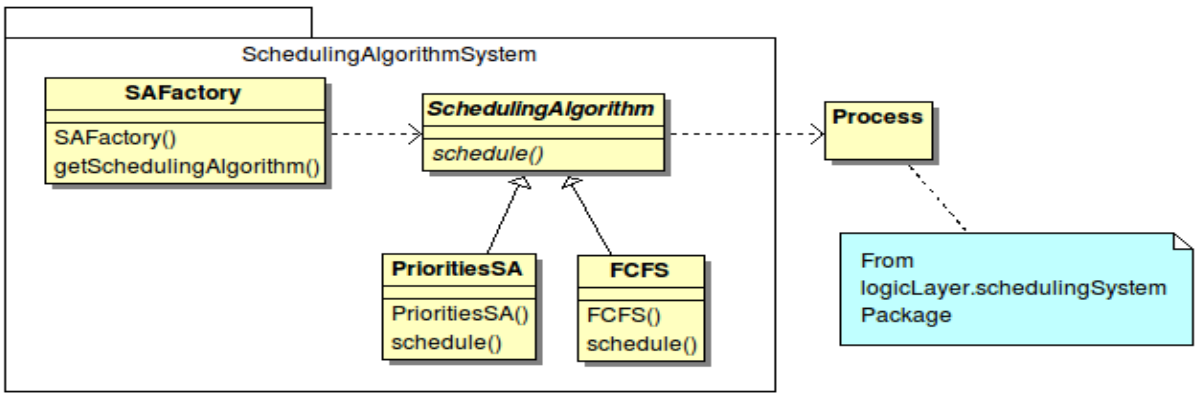


Ilustración 7: Diagrama de Clases, Algoritmos de Planificación

6.2.2 Paquete del 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.

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 Actor se encarga de realizar una ejecución en cada ciclo. La mayor parte del algoritmo de simulación se encuentra allí. Finalmente la clase Task (Tarea) contiene la información referente a una tarea.

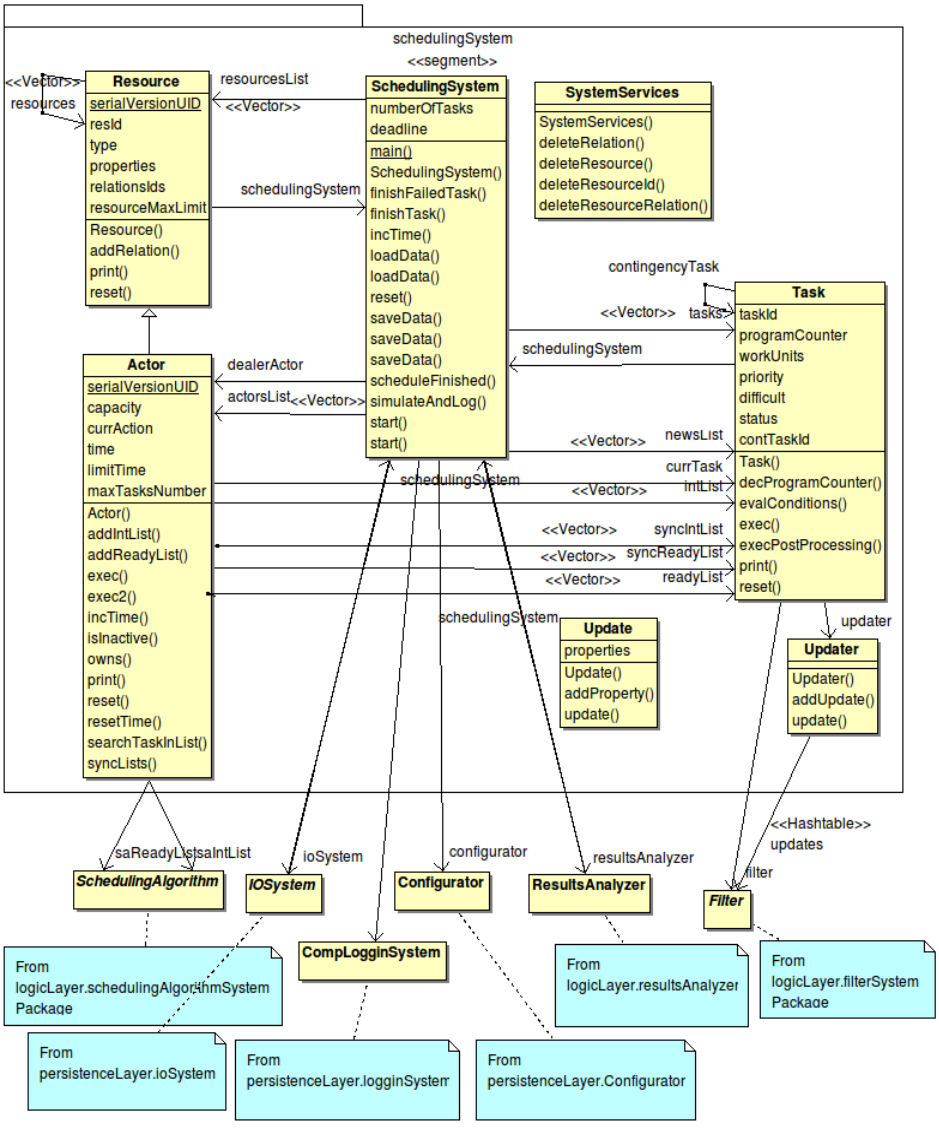

schedulingSystem
<>
Resource
serialVersionUID
resId
type
properties
relationsIds
resourceMaxLimit
Resource()
addRelation()
print()
reset()
resourcesList
<<Vector>>
resources
SchedulingSystem
numberOfTasks
deadline
main()
SchedulingSystem()
finishFailedTask()
finishTask()
incTime()
loadData()
loadData()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

Ilustración 8: Diagrama de Clases, Sistema de Simulación

6.2.2.1 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 hay una tarea activo*
      - *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*

Simular2:

- *Si la tarea en ejecución ha llegado a su fin:*
  - *Terminar tarea (pasa a estado finalizado).*
- *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 ejecutándose en 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.*

6.3 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.

6.3.1 Analizador de Resultados (resultsAnalizer)

Este paquete contiene la clase ResultsAnalizer, la cual, luego de terminada una simulación, es utilizada por el SchedulingSystem para analizar los resultados obtenidos. Si el usuario de la aplicación lo desea puede modificar los métodos de esta clase para agregar más análisis. Si dicho análisis resultara muy complejo se puede realizar el mismo en una o más clases aparte y agregar una instancia de la principal a las propiedades de ResultsAnalyzer.

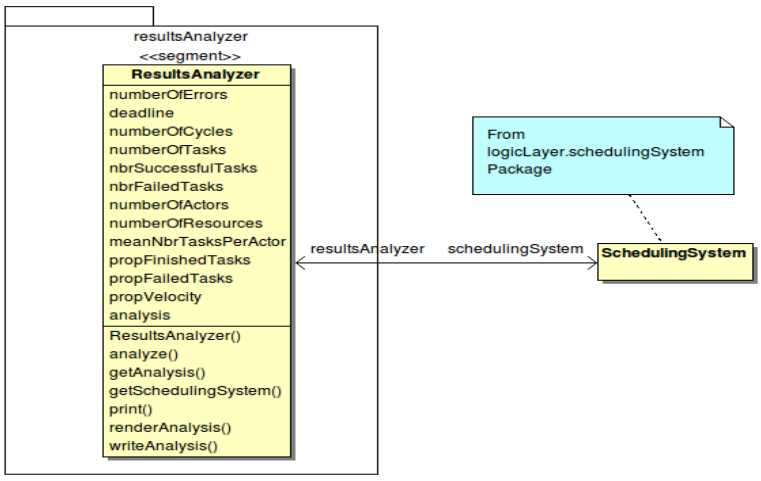

resultsAnalyzer
<>
ResultsAnalyzer
numberOfErrors
deadline
numberOfCycles
numberOfTasks
nbrSuccessfulTasks
nbrFailedTasks
numberOfActors
numberOfResources
meanNbrTasksPerActor
propFinishedTasks
propFailedTasks
propVelocity
analysis
ResultsAnalyzer()
analyze()
getAnalysis()
getSchedulingSystem()
print()
renderAnalysis()
writeAnalysis()
From
logicLayer.schedulingSystem
Package
resultsAnalyzer
schedulingSystem
SchedulingSystem

Ilustración 9: Diagrama de Clases, Analizador de Resultados

6.3.2 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.

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.

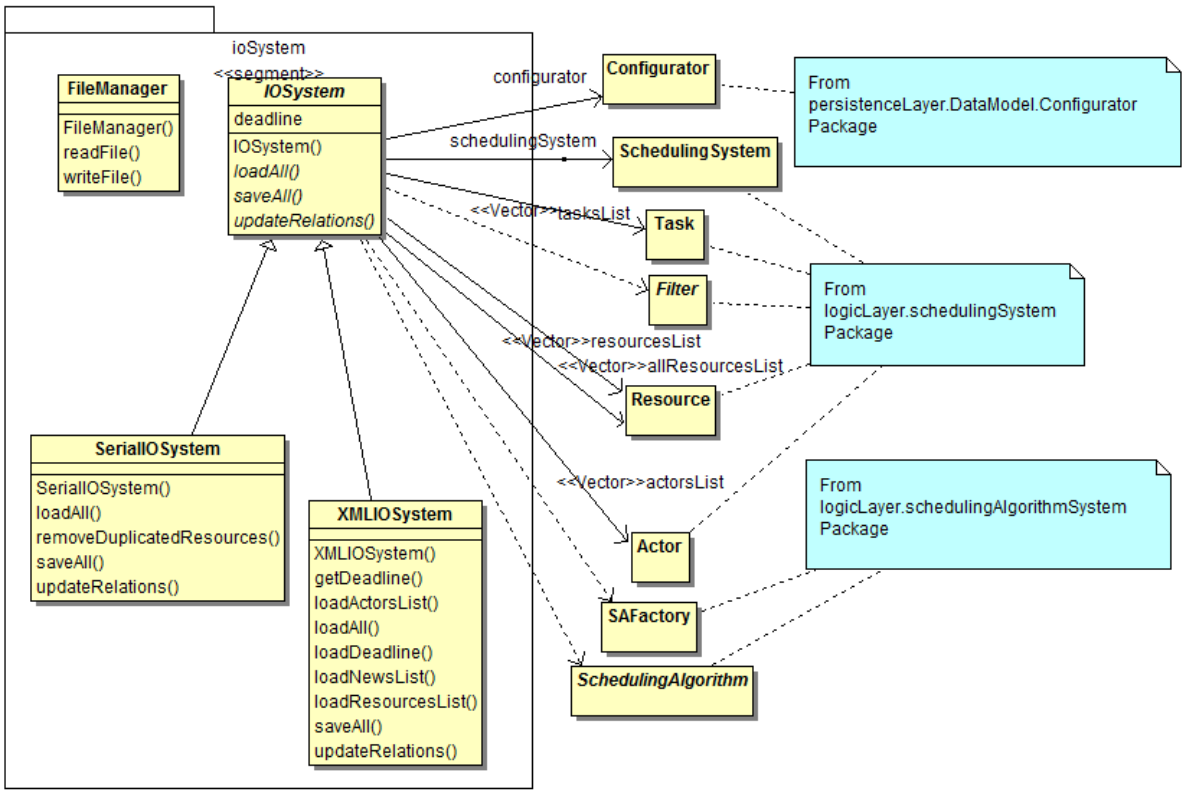

ioSystem
<>
FileManager
FileManager()
readFile()
writeFile()
IOSystem
deadline
IOSystem()
loadAll()
saveAll()
updateRelations()
configurator
Configurator
schedulingSystem
SchedulingSystem
<<Vector>>tasksList
Task
Filter
<<Vector>>resourcesList
<<Vector>>allResourcesList
Resource
<<Vector>>actorsList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

Ilustración 10: Diagrama de Clases, Sistema de Entrada/Salida

6.3.3 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.

La clase LogginSystem es abstracta, tiene implementado el metodo *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.

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.

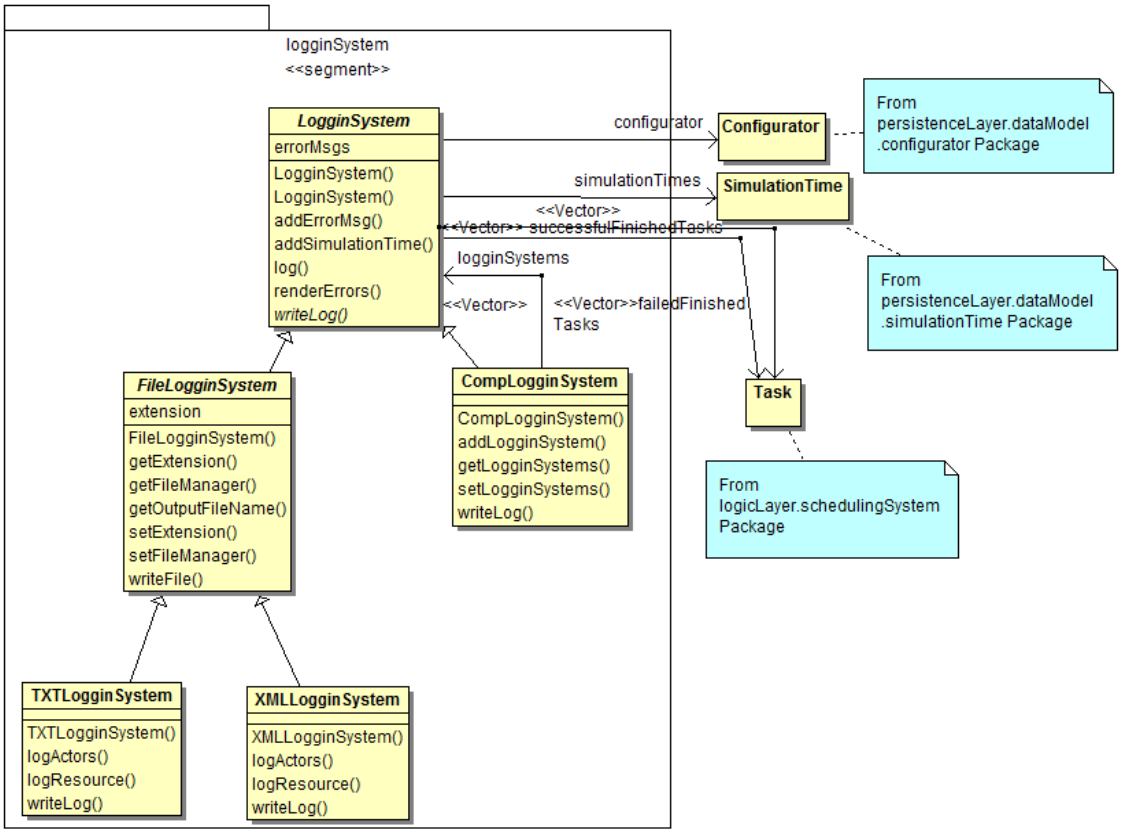

logginSystem
<>
LogginSystem
errorMsgs
LogginSystem()
LogginSystem()
addErrorMsg()
addSimulationTime()
log()
renderErrors()
writeLog()
configurator
Configurator
From
persistenceLayer.dataModel
.configurator Package
simulationTimes
SimulationTime
<<Vector>>
<<Vector>> successfulFinishedTasks
From
persistenceLayer.dataModel
.simulationTime Package
logginSystems
<<Vector>>
<<Vector>>failedFinished
Tasks
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 11: Diagrama de Clases, Logueo de Resultados

6.3.4 Paquete del 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.

6.3.4.1 Sistema de Filtros (filterSystem)

Los filtros forman parte de la capa lógica. El rubro 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. Aparte de los diferentes filtros nombrados, existen una serie más de los mismos encargados de formar parte de la estructura de herencia de tales.

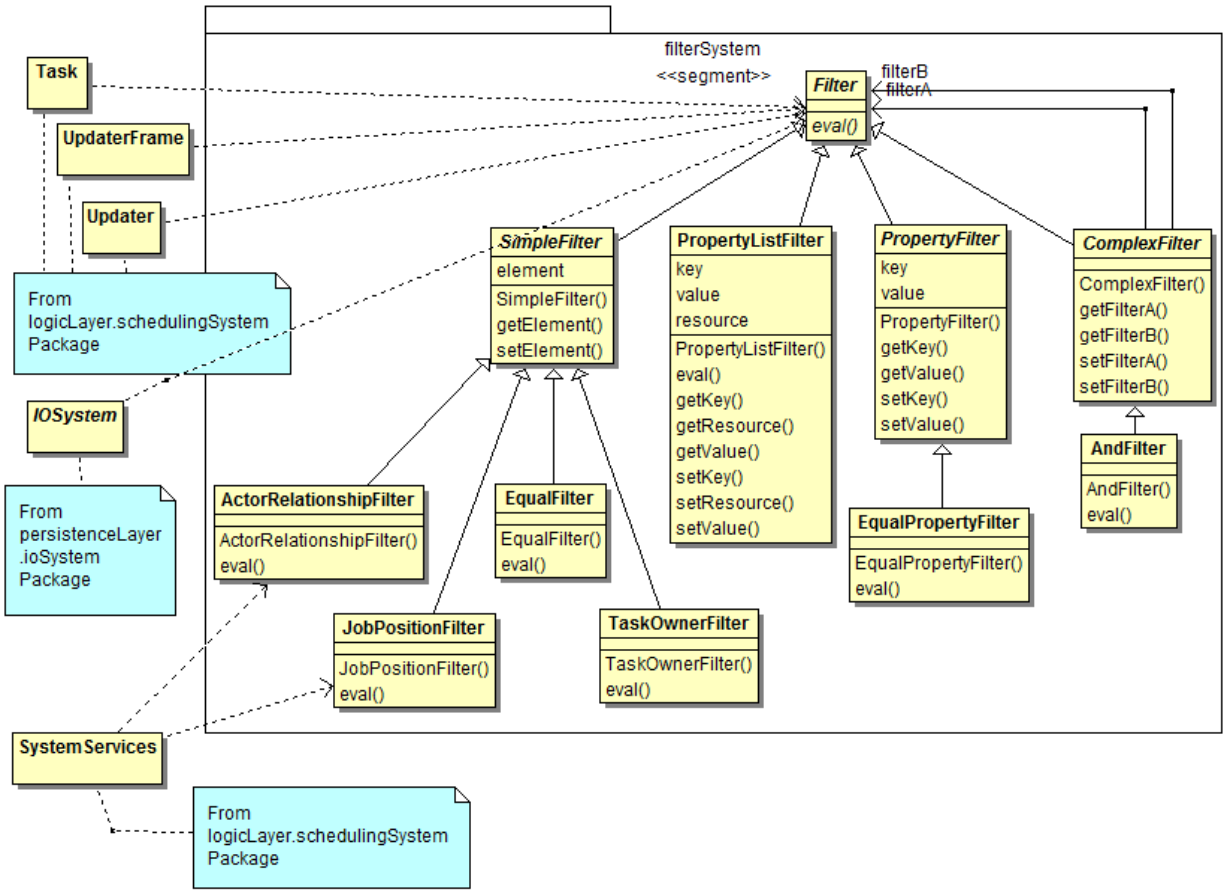

filterSystem
<>
Task
UpdaterFrame
Updater
From
logicLayer.schedulingSystem
Package
IOSystem
From
persistenceLayer
.ioSystem
Package
SystemServices
From
logicLayer.schedulingSystem
Package
Filter
eval()
filterB
filterA
SimpleFilter
element
SimpleFilter()
getElement()
setElement()
PropertyListFilter
key
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()
EqualPropertyFilter
EqualPropertyFilter()
eval()
ActorRelationshipFilter
ActorRelationshipFilter()
eval()
EqualFilter
EqualFilter()
eval()
JobPositionFilter
JobPositionFilter()
eval()
TaskOwnerFilter
TaskOwnerFilter()
eval()

Ilustración 12: Diagrama de Clases, Sistema de Filtros

7 Uso del Framework

7.1 Acerca de 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.

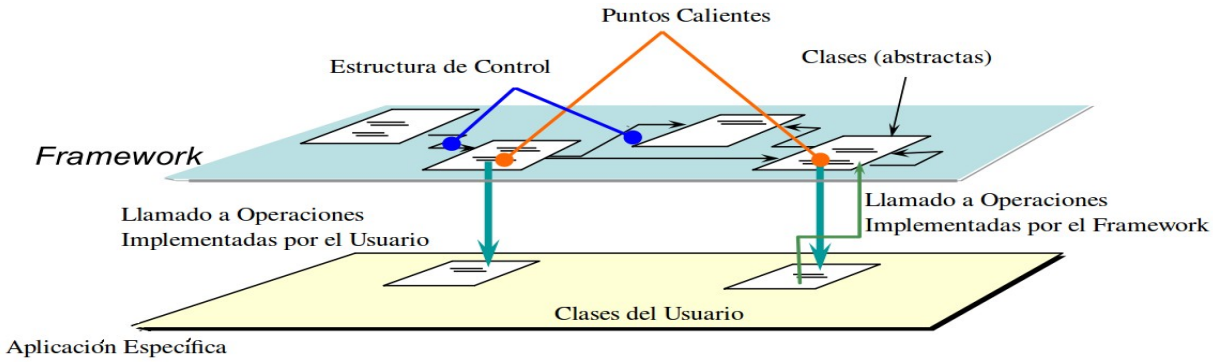


Ilustración 13: Conceptos Básicos de Frameworks

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.

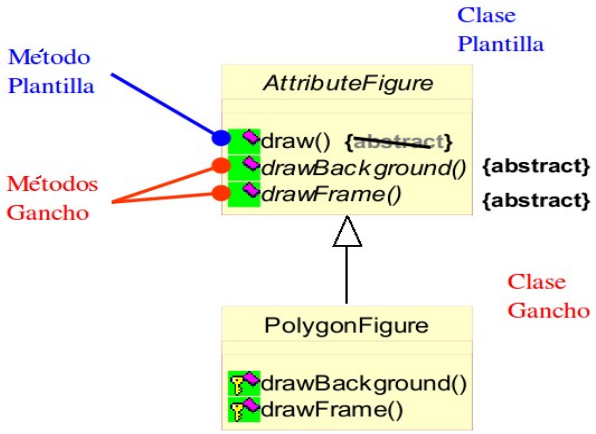


Ilustración 14: Métodos y Clases Plantilla y Gancho

[DSS07]

7.2 Crear un algoritmo de planificación

Básicamente se debe subclasear de SchedulingAlgorithm. 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;
        }
}
```

7.3 Leer Datos de Salida

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

...

8 Casos de test

En la ingeniería del software, los casos de prueba o test case 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 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.

8.2 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.

8.3 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.

8.4 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.

8.5 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.

8.6 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.

8.7 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.

9 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/

[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