Contenido

Índice de Tablas

Índice de Figuras

1. Introducción

1.1. Contexto general

AmiLab (Ambient Intelligence Laboratory), es el laboratorio de inteligencia ambiental de la Escuela Politécnica Superior de la Universidad Autónoma de Madrid (EPS-UAM), con sedes en Instituto Tecnológico Superior Zacatecas Norte (ITSZN) y la Universidad del Centro de la Provincia de Buenos Aires (UNCPBA).

En este laboratorio, un grupo de investigadores aplican paradigmas de computación ubicua en entornos activos. Se han creado salas de estar, salas de juntas y aulas inteligentes como entornos de desarrollo, experimentación y prueba para proyectos alineados a investigaciones sobre control, adaptabilidad, contexto y privacidad.

Sobre la estructura que proporciona AmiLab se ha desarrollado el proyecto ITECH CALLI ("Dentro de la casa" en lengua Nahuatl), cuyo objetivo principal consiste en desplegar un Entorno Activo de un aula inteligente y una sala de juntas en el Instituto Tecnológico Superior Zacatecas Norte e implementar aplicaciones tanto de software como de hardware en base a ese entorno para resolver problemáticas específicas.

Particularmente en el aula inteligente, se está trabajando en el desarrollo de aplicaciones que permitan cubrir las necesidades y problemáticas generadas en educación a distancia, por medio de la interacción local y remota con los elementos físicos y conceptuales del entorno.

1.2. Descripción del problema

En AmiLab se han generado múltiples aplicaciones de software para el control y comunicación de los elementos del entorno basándose en dispositivos de hardware como Phidget[1] y X10[2] y su representación conceptual en la Pizarra.

[1] Los Phidget son tarjetas que permiten controlar y censar dispositivos eléctricos desde una computadora a través de un API compatible con diversos lenguajes de programación, http://www.phidgets.com/

A pesar de que la dinámica de las aplicaciones es relativamente similar, no son desarrolladas de manera uniforme y consistente con un modelo, es decir, el desarrollo actual de aplicaciones depende mucho de la pericia y el conocimiento del investigador sobre el problema y el lenguaje de programación.

Cabe recalcar que la mayor parte de la creatividad y esfuerzo invertido en desarrollo se concentra en la codificación de la aplicación, más que el entendimiento lógico y la generación de un modelo de solución.

Las aplicaciones desarrolladas tienen un alto grado de dificultad en cuanto a mantenimiento. Puesto que en su mayoría tienen una gran cantidad de bifurcaciones y están pobremente documentados en su estructura, por consecuencia, las tareas de corrección y actualización se limitan a su autor y al uso de mensajes en consola para su depuración, además, al modificar la funcionalidad de un proyecto, se deben revisar nuevamente la mayor parte de los predicados para evitar efectos colaterales.

[2] El X10 es un "lenguaje" de comunicación que permite a los productos compatibles entablar comunicación a través del cableado eléctrico, http://www.smarthome.com/about_x10.html

1.3. Objetivos

1.3.1. Objetivo general

Implementar un paradigma de desarrollo de software que permita manejar la complejidad del software e incrementar la reusabilidad en aplicaciones para el aula inteligente.

1.3.2. Objetivos específicos

- Comprobar si el paradigma COSA puede ser implementado en el desarrollo de aplicaciones para el aula inteligente en AmiLab
- Documentar el proceso de implementación del paradigma COSA en un proyecto piloto dentro de AmiLab.
- Analizar los resultados obtenidos en el proyecto piloto mediante la comparación con las aplicaciones implementadas actualmente en AmiLab.

1.4. Alcances y limitaciones

Los alcances del presente estudio son:

- Se desarrolló un proyecto piloto usando el API para tarjetas phidget y drivers de la Pizarra para demostrar la posibilidad de implementar el paradigma COSA.
- Se documentó paso a paso el proceso de desarrollo del servidor de censado de butacas bajo el paradigma COSA.
- Se realizó una comparación con otros proyectos de software de naturaleza similar.

Las limitaciones son las siguientes:

- La interacción con la Pizarra se limita por medio de los drivers. No se manipuló ni publicó sus operaciones internas.

- La interacción de hardware se realizó únicamente con tarjetas Phidget, dejando de lado los dispositivos x10.
- Se realizó la ejecución del servidor de butacas con solo 8 butacas funcionales, por limitaciones de hardware en cuanto a disponibilidad de tarjetas Phidget.
- No se realizaron pruebas con usuarios del aula inteligentes en escenarios de estrés de solicitudes en las butacas.

1.5. Organización del reporte

En el presente reporte se abordan los siguientes tópicos:

En la sección 2 "Estado de arte" se plantean conceptos y enfoques recientes referentes al tratamiento de la complejidad del software y la coherencia modelo-aplicación.

En seguida, en la sección 3 "El paradigma COSA" se define algunos conceptos básicos, características y una propuesta del proceso de desarrollo usando el enfoque COSA.

Posteriormente, en la sección 4 "La implementación del paradigma COSA" se muestra paso a paso el desarrollo de un proyecto piloto realizado en el entorno del aula inteligente de AmiLab.

Finalmente, en la sección de 5 "Resultados" se plasman la comparación de la aplicación producida bajo el paradigma COSA frente a otras aplicaciones de naturaleza similar desarrolladas en AmiLab

2. Estado del arte

La complejidad del software es uno de los tópicos más comunes en el desarrollo del software actual, se han realizado múltiples investigaciones y se han generado diversos paradigmas que buscan manejarla, a continuación se realiza una reseña sobre algunos enfoques de interés a este respecto.

El uso de modelos de solución para desarrollar una aplicación ha sido un campo de estudio amplio, entre los enfoque más interesantes se encuentra el Model-Driven Engineering, por otro lado el manejo de la complejidad del software ha sido el centro de atención de otros enfoques como el Agents-Oriented, en ese sentido, la complejidad del problema se agrava con la dificultad de mantener el código "spaghetti" y ha promovido la búsqueda de nuevas técnicas de control-flow y data-flow como el enfoque table-driven.

Model-Driven Engineering (MDE)

En el artículo "Model-driven engineering" [(1)] se expone la dificultad de las herramientas CASE para transformar especificaciones de alto nivel a código ejecutable y para escalar y manejar la complejidad, como consecuencia, se han investigado diferentes enfoques para manejar la complejidad del software por medio de modelos.

El enfoque MDE (Model-Driven Engineering) es un enfoque de desarrollo de software que se centra en la creación de modelos o abstracciones de alto nivel para generar aplicaciones complejas de software, con el objetivo de incrementar la productividad, maximizar la compatibilidad entre sistemas, simplificar el proceso de diseño y promover la comunicación entre los miembros del equipo de desarrollo. Los enfoques MDE buscan solventar las deficiencias que hasta ahora aquejan a las herramientas CASE, reconociendo que los modelos por sí solo no son suficientes para manejar la complejidad del software y generando líneas de investigación a este respecto.

Para desarrollar tecnología bajo el enfoque MDE se combinan los siguientes tópicos: (a) Lenguajes de modelado de dominio específico, que permiten formalizar la estructura, comportamiento y requerimientos de los sistemas en un dominio en particular; (b) Motores de transformación y generadores, que analizan determinados aspectos de un modelo y los

sintetizan en varios tipos de artefactos tales como código fuente, descriptores XML, modelos alternativos, entre otros.

A pesar de esto, en un estudio publicado en el artículo "Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals" [(2)] donde se encontró como resultado que los desarrolladores continúan desarrollando bajo un enfoque code-centric y el modelado de software aún sigue siendo poco popular. Adicionalmente, se encontró que las herramientas de modelado son usadas, primordialmente, para crear documentación y con poca generación de código automática y que la percepción de los participantes que usan modelado concuerda en que los modelos quedan rápidamente obsoletos e inconsistentes con el código.

Agent–Oriented (AO) Software Methodologies

El paradigma de programación AO (Agent-Oriented) fue introducida en 1993 por Yoav Shoam [(3)], en el cual los agentes proveen un alto nivel de abstracción para desarrollar software y, potencialmente, simplificar el diseño de sistemas de información complejos.

En el libro Agent-oriented methodologies [(4)], se describe a un agente como una entidad de un sistema que posee proactividad, reactividad, autonomía, localización y sociabilidad, también son considerados como entidades autónomas de resolución de problemas constituidos por: creencias, deseos e intenciones, que además interactúan con otros agentes a través de un lenguaje de comunicación declarativo de alto nivel.

El enfoque A-O ha sido ampliamente usado en el desarrollo de sistemas complejos y distribuidos en entornos como inteligencia ambiental [(5)], manejo de sensores y dispositivos [(6)] e investigaciones académicas [(7)]. Existen múltiples referencias de éxito e incremento de popularidad de desarrollo bajo el enfoque A-O.

Approach table-driven

En 1994, Wayne Cunneyworth [(8)], definió que el enfoque Table-driven consiste en usar estructuras de datos como tablas y vectores para manipular el control-flow o el data-flow de un programa [(9)], teniendo dos consideraciones: (1) qué información se almacenará en las

tablas y (2) cómo accesar la información almacenada; este enfoque contribuye a decrementar considerablemente el número de sentencias de control y decisión (if, case) y usar el acceso a datos de manera directa mediante un índice o llave. El uso de algoritmos Table-driven se extiende desde tablas de relación, reglas, decisión, traducción, mensajes y llamados a métodos.

En el 2004, Steve McConnell [10], proporcionó un nuevo auge al enfoque table-driven, a pesar de ser una idea relativamente antigua, posicionándolo como un método para mejorar la flexibilidad de las aplicaciones, manejar la complejidad del software e indicando que el resultado es el desarrollo de aplicaciones más pequeñas, más abstractas y por lo tanto más fáciles de mantener.

3. El paradigma C.O.S.A.

3.1. ¿Qué es C.O.S.A.?

El paradigma COSA (Coherent Object System Architecture) es un paradigma de ingeniería de software temporal que permite enfrentar la **complejidad del software** y crear una **conexión sólida modelo-aplicación**, mediante la implementación de 3 partes esenciales: Engine, tabla de reglas y procedimientos de apoyo.

Basándose en los principios de máquinas de estados y notación BNF se desarrollan aplicaciones en las cuales el control-flow y el data-flow permanecen separados, facilitando el seguimiento (trace), depuración, validación y reuso de sus elementos.

El paradigma COSA fue desarrollado por Gordon Morrison [(11)] basándose en la experiencia obtenida en el desarrollo de sistemas de radar climático en tiempo-real, sistemas de comunicación de alto rendimiento, tecnología multi-core y hyper-threading.

3.2. Características

El paradigma COSA tiene las siguientes características:

- La base para desarrollar una solución es la comprensión lógica del problema y su representación mediante **especificaciones BNF** (Backus-Naur Form) aprovechando su potencial de presentar la especificación formal de aplicaciones tan diversas como aplicaciones WEB, lenguajes de programación, bases de datos, entre otras.

- El modelo de solución, representado en especificación BNF y diagrama de estado, es una pieza imprescindible del paradigma COSA, sin el desarrollo bajo este enfoque sería extremadamente difícil

- Tiene 3 partes esenciales en su implementación:

    - **Engine**. Es un algoritmo de control reutilizable que permite manipular reglas de control-flow contenidas en la tabla de reglas, el "COSA engine" considera el estado dinámico y el estado estático de las reglas,

proporcionando dos puntos de comportamiento (Verdadero y Falso) en los cuales se ejecuta una acción y se indica el siguiente estado válido. Cabe recalcar que una aplicación usa uno o más "Engines" como puntos centrales de control.

- **Tabla de reglas**. La tabla de reglas de COSA se basa en la especificación BNF, diagrama de estados de la solución y en la lógica binaria. Cada fila contiene una regla en la cual se especifica el comportamiento verdadero o el comportamiento falso y la siguiente regla a aplicar, es decir, una fila que indica qué se espera del problema y qué hacer cuando esos resultados esperados sean recibidos o bien no sean recibidos.

- **Procedimientos de soportes**. Los procedimientos, en el contexto de COSA, son acciones específicas en el comportamiento de la aplicación. Un procedimiento de soporte regularmente corresponde al comportamiento de un estado definido de la aplicación. se pueden considerar tres tipos de procedimientos: (1) Activos, los cuales ejecutan un conjunto de instrucciones definidas en un estado, (2) nulos, que NO ejecutan instrucciones y son usados para "ocupar" un lugar en la tabla de reglas e (3) inválidos, que permiten manejar los estados inválidos debido a un error específico o situación desconocida. Si es necesario, los procedimientos participan en la asignación del estado dinámico correcto.

- El paradigma COSA mejora la estabilidad de las aplicaciones debido a que la tabla de reglas representa un comportamiento binario los errores de lógica y operaciones inválidas son tratados implícitamente.

- En la arquitectura de COSA se ofrece la ventaja de que el "control-flow" y "data-flow" permanecen separados, esto se debe a que el Control-flow, definido en el *Engine*, interactúa con el Data-flow, definido por los procedimiento de soporte, mediante la tablas de reglas BNF (Logic-flow), lo que incrementa la flexibilidad de

las aplicaciones cuando se extienden las reglas y se adicionan los procedimientos de soporte necesarios para manejar nuevos estados.

- Otra característica del paradigma COSA es que usa *table-driven methods* (métodos dirigidos por tablas), mediante el polimorfismo y vinculación dinámica. El enfoque de *table-driven methods* usa una estructura de datos (tablas o vectores) en lugar de sentencias de control (if, case), en casos en los que el flujo de la lógica se torna más compleja, el enfoque *table-driven* se permite mantener una simplicidad, eficiencia y flexibilidad más alta que el enfoque ITE (*If-Then-Else*).

- Sobrepone la ingeniería temporal a la ingeniería espacial. La ingeniería temporal es una disciplina basada en una estructura de acceso que permita navegar entre los miembros, funciones o procedimientos de una aplicación, y cuyo proceso de desarrollo puede ser repetido manteniendo la coherencia entre el modelo y la solución.

3.3.Proceso de desarrollo

Un proceso de desarrollo adaptado al paradigma COSA sigue 6 pasos principales, los cuales pueden ser instrumentados o adaptados con *PSP*© o *TSP*© para obtener métricas provechosas sobre el desempeño, rapidez y calidad del desarrollo tanto del proceso como del producto.

En la Figura 3-1 se ofrece una reseña sobre los aspectos generales en el proceso, posteriormente se ofrecen 2 ejemplos completos para mostrar su realización:

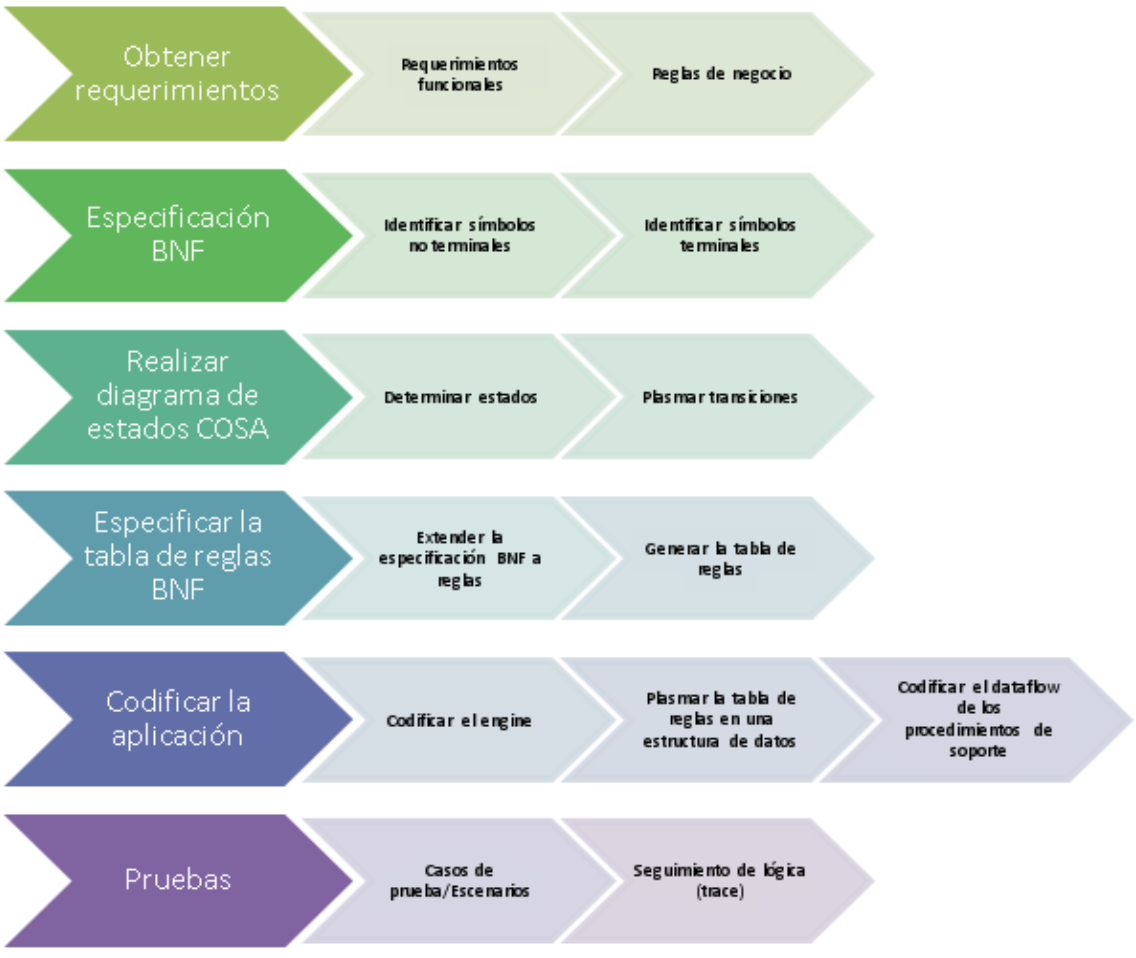


Figura 3-1 Proceso de desarrollo COSA

Las actividades a realizar en cada paso son las siguientes:

1. **Obtener requerimientos**: En este paso desarrollador recopila los requerimientos de la aplicación en base a entrevistas observaciones y experimentación, la formalidad depende del contexto de la aplicación, desde una descripción textual simple de los requerimientos hasta un SRS.

2. **Especificar reglas en BNF**. Este paso es uno de los más complejos en el proceso de desarrollo, pues requiere una comprensión amplia de la lógica del problema. Se usa la especificación BNF por la potencia y formalidad en la representación de la solución. La lógica del problema se representa por medio de reglas iniciando con las más generales, por ejemplo:

   Aplicación ::= <rRegla1>+<rRegla2>?

   Posteriormente cada regla se extiende hasta llegar a los símbolos terminales que corresponderá a los procedimientos de soporte. Por ejemplo:

   rRegla1 ::= <iEstado1>?<iEstado2> | <iEstado3> <iEstado4>*

   Donde *rRegla1* es un símbolo no terminal que se expande usando los símbolos terminales *iEstado1, iEstado2...iEstadoN*, donde cada estado especificará una serie de acciones

3. **Realizar el diagrama de estados COSA**. El diagrama de estados COSA es una representación alternativa a la especificación BNF, se recomienda realizar el diagrama para efectos de comprensión y visualización general de los estados y sus transiciones. Tanto la especificación BNF y el diagrama de estados deben ser *consistentes* entre sí, un cambio en uno debe reflejarse forzosamente en el otro.

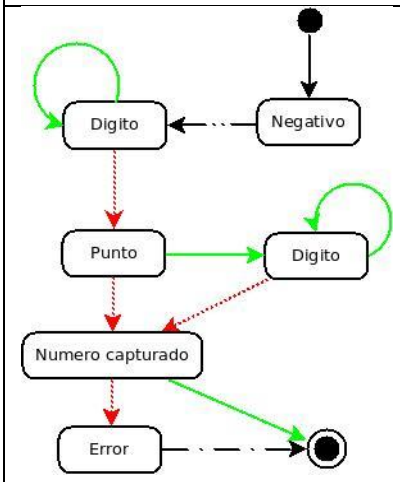
   El diagrama de estados se caracteriza por tener un conjunto de estados finito (correspondientes a los símbolos terminales de la especificación BNF) y 3 tipos de transiciones: (a) Verdadero, que indica el siguiente estado válido en caso de que el estado actual se haya ejecutado, (b) falso, que indica el siguiente estado válido en caso de que el estado actual no corresponda al estado dinámico y (c) ambos, que

indican el siguiente estado válido sin importar ninguna de las condiciones anteriores. Debido a lo anterior el diagrama no requiere una narrativa para describir las transiciones a otros estados, porque, gracias a la alta cohesión entre las acciones, los estados solo tienen transiciones *true, false o either*.

4. **Especificar la tabla de reglas BNF**. La tabla de reglas es una construcción realizada sobre una estructura de datos que contiene filas que representan reglas lógicas. Cada regla contiene los siguientes campos:

    - *Rule* = Índice o identificador de regla,
    - *State static* = Estado estático, se comparara con el estado dinámico, este último es modificado por estímulos del entorno o por los procedimientos de soporte.
    - *True action* = Representa el procedimientos de soporte (método o función) que se ejecutara si el estado estático y el estado dinámico **coinciden**.
    - *Next true* = Indica la siguiente regla a ejecutar si el estado estático y el estado dinámico **coinciden**.
    - *False action/Next false* = Similar a *True action/next true* con la diferencia que se ejecutaran cuando el estado estático y el estado dinámico **difieren**.

Las reglas se construyen a partir de la especificación BNF y del diagrama de estados completando sus elementos de la siguiente forma, suponiendo que se tiene el siguiente diagrama de estados y su especificación en BNF:

Diagrama de estados	Reglas BNF
	Ejemplo ::= <Negativo>? <Digito>* <Punto>? <Digito>+ <Numero capturado> | <Error>

Se identifican primero los estados estáticos (State static) de la especificación BNF o diagrama de estados, posteriormente se complementa la tabla en sus columnas *True action / False action* con referencias al procedimiento de soporte correspondiente. Note como se adiciona un procedimiento Ignore, en los casos en los cuales no se ha especificado acciones.

Finalmente se indica la siguiente regla a ejecutar con *Next true / Next false* correspondiente al estado válido siguiente.

Rule	State Static	True action	Next true	False action	Next false
1	iNegativo ?	<Negativo>	2	<Ignore>	2
2	iDigito *	<Digito>	2	<Ignore>	3
3	iPunto ?	<Punto>	4	<Ignore>	4
4	iDigito +	<Digito>	4	<Ignore>	5
5	iNumeroCapturado	<CapNum>	End	<Ignore>	6
6	iError	<Error>	End	<Unknown>	End

5. **Codificar la aplicación**. La penúltima etapa del proceso corresponde a la codificación, en esta etapa el modelo se traduce directamente a instrucciones del

lenguaje de programación deseado, manteniendo siempre la coherencia entre el modelo y la aplicación.

Codificar el Engine: A pesar de las diferencias entre los lenguajes de programación la traducción del *Engine* a la aplicación mantiene una coherencia notable, como se muestra en la Tabla 3-1.

Algoritmo Engine	**Engine Java**

Interface Name (Attributes if any)
Engine Control
Testing managing logic analysis TRUE or FALSE
True trace (optional)
True behaviors
Next true time
Or
True trace (optional)
True behaviors
Next true time
End of testing
End of engine control (scope)
Return value (if any)
End engine
	public void engine(int Arg){
	engineLocal=true;
	dynamicState=Arg;
	while (engineLocal && Global.engineGlobal){
	if(dynamicState==rRule[iTime].iState){
	rRule[iTime].pTrueRule.execute();
	iTime=rRule[iTime].iTrueRule;
	}
	else{
	rRule[iTime].pFalseRule.execute();
	iTime=rRule[iTime].iFalseRule;
	}
	}
	}

Engine en C++	**Engine en Delphi**

void engine(int iState){
dynamicState=iState;
while(engineLocal && engineGlobal){
if(dyanmicState==Tbl[iTime].state){
(this->*(Tbl[iTime].True_behavior))();
iTime=Tbl[iTime].Next_true;
}else {
(this->*(Tbl[iTime].False_behavior))();
iTime=Tbl[iTime].Next_false;
}
}
	procedure TCOSA.engine(intState:integer)
	begin
	engLocal:=true;
	dynamicState:=intState;
	while(engLocal) AND (engGlobal) do
	begin
	if dynamicState = rRule[iTime].staticState then
	begin
	rRule[iTime].pTrueRule;
	iTime:=rRule[iTime].iTrueRule;
	end else
	begin
	rRule[iTime].pFalseRule;
	iTime:=rRule[iTime].iFalseRule;
	end;
	end;

Tabla 3-1 Engine COSA

Plasmar la tabla de reglas: Se debe crear una estructura de datos que soporte la información correspondiente a las reglas BNF, adaptando características de polimorfismo, vinculación dinámica e interfaces. La estructura de las reglas se encuentra en la Tabla 3-2.

Rule	State Static	True action	Next true	False action	Next false	Trace
Identificador entero	Identificador entero	Apuntador a función/ interface/ superclase	Número entero	Apuntador a función/ interface/ superclase	Número entero	Número entero

Tabla 3-2 Estructura de regla BNF extendida

Note que se agregó una columna de Trace que sirve para seguir y depurar la ejecución de la aplicación

Traducido a un lenguaje de programación como Java sería:

Código de la estructura de la tabla de reglas

```
public class Rule
{
    public int iRule;
    public int StaticState;
    public State TrueAction;
    public int NextTrue;
    public State FalseAction;
    public int NextFalse;
    public int iTrace;

  public Rule(int Rule, int StaticState, State TrueBehavior, int NextTrueRule, State FalseBehavior, int NextFalseRule, int Trace){
    this.iRule=Rule;
    this.StaticState=StaticState;
    this.TrueAction=TrueBehavior;
    this.NextTrue=NextTrueRule;
    this.FalseAction=FalseBehavior;
    this.NextFalse=NextFalseRule;
    this.iTrace=Trace;
    }
  }
```

La tabla sería llenada de la siguiente forma:

Llenado de tabla de reglas

```
rRule=new Rule[N];
rRule[0]=new Rule(1, iNegativo, mNegativo, 2,   Ignore,   2,   100);
rRule[1]=new Rule(2, iDigito,    mDigito,    2,   Ignore,   3,   101);
rRule[2]=new Rule(3, iPunto,     mPunto,     4,   Ignore,   4,   102);
rRule[3]=new Rule(4, iDigito,    mDigito,    4,   Ignore,   5,   103);
rRule[4]=new Rule(5, iNumCap, mNumCap,   1,   Ignore,   6,   104);
rRule[5]=new Rule(6, iError,     mError,     1,   Ignore,   1,   105);
```

Donde las columnas correspondientes al *True action* y *False action* son procedimientos de soporte (en el caso de Java serían clases) que implementan una vinculación dinámica o polimorfismo (en Java seria una interface).

Codificar los procedimientos de soporte: Dentro de los procedimientos se plasma las acciones con alta cohesión correspondientes a un estado particular, en algunos casos los procedimientos de soporte pueden participar en la determinación del estado dinámico, por ejemplo:

Interface de estado

```
public interface State {
   public void execute();
}
```

Ejemplos de procedimientos de soporte

```
public class Negativo implements State {
   public void execute(){
      Numero= "-"  + Numero;
   }
}

public class Digito implements State {
   public void execute(){
```

```
        Numero=Numero+Digito;
    }
}
```

6. **Probar la aplicación**. La última etapa del proceso consiste en la prueba de la aplicación, es esta se selecciona una técnica de *Testing* que el desarrollador considere apropiada.

   El modelo generado con COSA tiene un mecanismo de seguimiento de lógica (Trace) que permite depurar y verificar la ejecución de la aplicación.

4. Implementación de C.O.S.A.

En esta sección se muestra un ejemplo de un proyecto piloto que consiste en un servidor de censado de butacas que demuestra la posibilidad de implementar el paradigma COSA en el desarrollo de aplicaciones de software en AmiLab.

Este proyecto se realizó basándose en las técnicas y conceptos de un proyecto de una **calculadora** que ha sido proporcionada en el libro "Breaking the time barrier – The temporal engineering of software" [(11)] e implementada en lenguaje Delphi y C++.

4.1. Desarrollo de servidor de censado de butacas

4.1.1. Obtener requerimientos

Un proyecto piloto para la implementación de COSA en AmiLab fue el servidor de censado de butacas. Después de entrevistas con los responsables del aula inteligentes y observaciones del área de trabajo se obtuvieron las siguientes notas respecto a la infraestructura:

- Existe un conjunto de 18 butacas en el aula inteligente (Ver Figura 4-1) distribuidas en 4 filas y 4 columnas. Cada butaca está identificada con un número consecutivo del 1 al 18
- Las butacas tiene 2 sensores de tipo push-button alimentados con una batería de 12 w:
    - Un sensor de ocupación (sensor A) ubicado en el asiento
    - Un sensor de petición (sensor B) ubicado en el descansabrazos.
- Los sensores están conectados a una tarjeta Phidget modelo 1012 PhidgetInterfaceKit 0/16/16 la cual a su vez se conecta a través del puerto USB a

una computadora donde se ubicara el servidor de censado de butacas. (Ver Figura 4-2)

- La computadora con el servidor se comunica a través de red LAN con una computadora central que contiene la “Pizarra” de AmiLab. (Ver Figura 4-2)

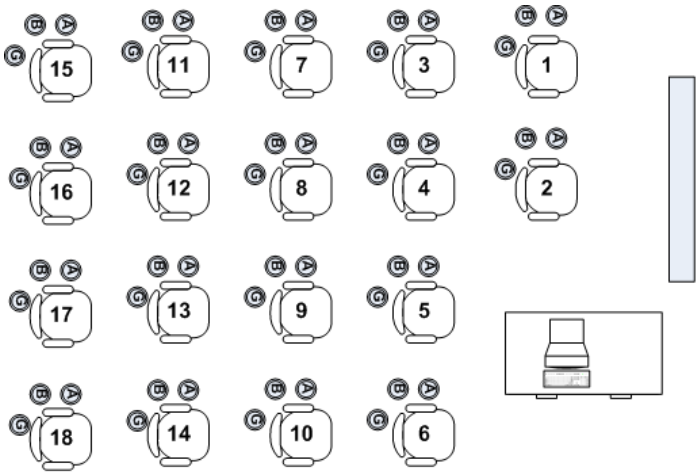


Figura 4-1 Organización de butacas en el laboratorio inteligente

El comportamiento esperado del servidor de censado de butacas es el siguiente:

- El servidor de censado de butacas leerá los estímulos de los sensores a través de la tarjeta PhidGet y notificará el estado de las butacas a la Pizarra usando el Driver correspondiente, el servidor debe minimizar el número de notificaciones al filtrar solo aquellas que sean pertinentes respecto al cambio de estado cambio de estado (Ver Figura 4-2).

- Las notificaciones pueden ser de 2 tipos: notificación de ocupación (Activo/Inactivo) y notificación de petición.

    - El censado de la ocupación/desocupación debe realizarse cada 500 milisegundos.

- El censado de peticiones debe ser inmediata siempre y cuando la butaca esté ocupada. En caso de que la butaca reciba una petición esta se mantendrá activa durante 3000 milisegundos, permitiendo que las demás aplicaciones reaccionen a la petición y mitigando múltiples peticiones en un corto periodo de tiempo.

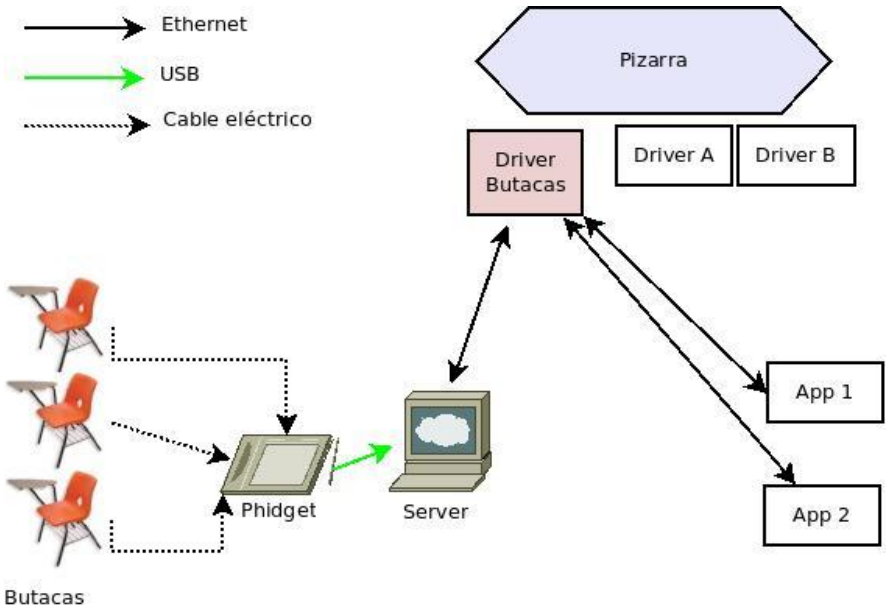


Figura 4-2 Arquitectura general del servidor de censado de butacas

4.1.2. Especificación BNF

Una vez obtenidos los requerimientos y analizando la lógica del problema, se formaliza el modelo de solución generando una especificación BNF en la cual se plantea el comportamiento de la aplicación.

Primeramente se identifican las reglas generales del problema:

ServerButacas ::= (<rCensar ocupación> (<rOcupado> <rPetición>* | <rDesocupado>))+;
El servidor censará la ocupación, posteriormente habrá dos comportamientos: (1) la butaca está ocupada y puede recibir cero o más peticiones o (2) la butaca está desocupada. El comportamiento del servidor se realiza uno o más veces.

Posteriormente se expande cada regla:

rCensar ocupación::= <Censar ocupación>
Al censar la ocupación se ejecuta el conjunto de acciones *<Censar ocupación>*
rOcupado::= <Filtro ocupar> <Notificar ocupación>?
Al ocupar una butaca se filtra la notificación de ocupación y, si es necesario, se notifica que la buta ha sido ocupada.
rPetición::= <Censar petición> (<Activar petición><espera larga><desactivar petición> | <espera corta>)
Al censar una petición, se pueden obtener dos comportamientos: (1) Se activa la petición, se ejecuta una espera larga y se desactiva la petición o (2) se realiza una espera corta.
rDesocupado::=<Filtro desocupado> <Notificar desocupación>? <Espera corta>
Al desocupar una butaca se filtra la notificación de desocupación y, si es necesario, se notifica que la butaca ha sido desocupada.

Una vez expandidas las reglas se procede a identificar los símbolos terminales que corresponderán a los procedimientos de soporte. Al final la especificación BNF completa se muestra en la Figura 4-3.

//Símbolos No Terminales

- **ServerButacas** ::= (<rCensar ocupación> (<rOcupado> <rPetición>* | <rDesocupado>))+;
- **rCensar ocupación**::= <Censar ocupación>
- **rOcupado**::= <Filtro ocupar> <Notificar ocupación>?
- **rPetición**::= <Censar petición> (<Activar petición><espera larga><desactivar petición> | <espera corta>)
- **rDesocupado**::=<Filtro desocupado> <Notificar desocupación>? <Espera corta>

//Símbolos Terminales

- **Censar ocupación**::= Leer del puerto phidget correspondiente a la ocupación
- **Censar petición** ::= Leer del puerto phidget correspondiente a la petición
- **Activar petición** ::= Enviar mensaje a la pizarra de petición activada
- **desactivar petición** ::= Enviar mensaje a la pizarra de petición desactivada
- **Filtro ocupar** ::= Filtrar solo los cambios de ocupación de las butacas
- **Notificar ocupación** ::= Enviar mensaje a la pizarra de ocupación activada
- **Filtro desocupado** ::= Filtrar solo los cambios de desocupación de las butacas
- **Notificar desocupación** ::= Enviar mensaje a la pizarra de ocupación desactivada
- **Espera corta** ::= Espera de 500 ms
- **Espera larga** ::= Espera de 3000 ms

Figura 4-3 Especificación BNF de servidor de censado de butacas

4.1.3. Diagrama de estados COSA

De manera alternativa se puede representar la especificación BNF usando uno o varios diagramas de estado en los que se plantean gráficamente las reglas y permite una mejor comprensión de las mismas.

De forma general, el diagrama de estado basado en la Figura 4-3, quedaría como se muestra en la Figura 4-4.

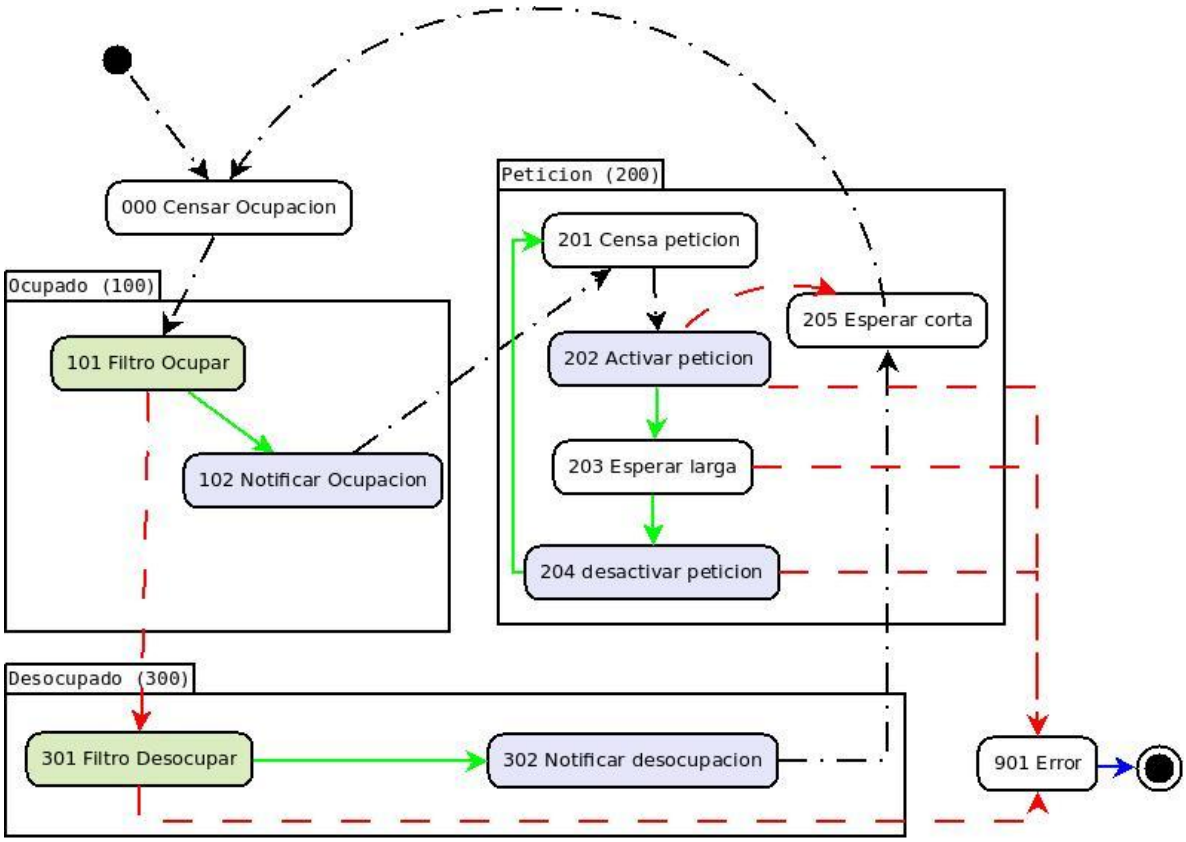
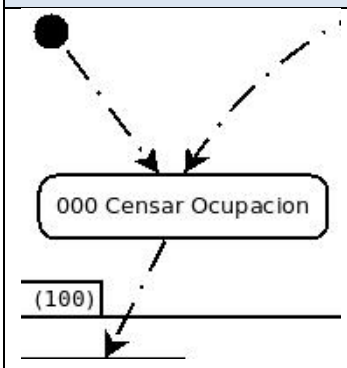
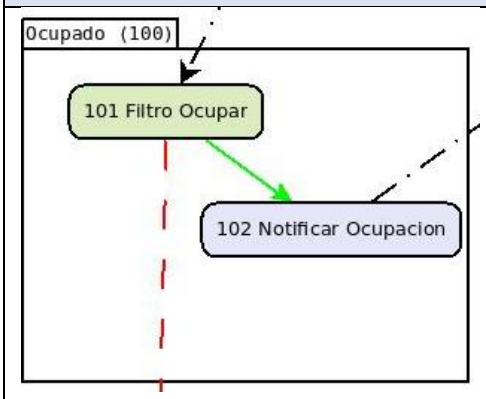
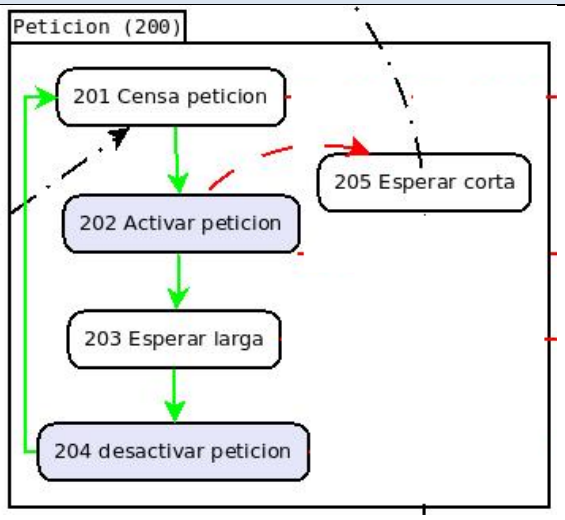


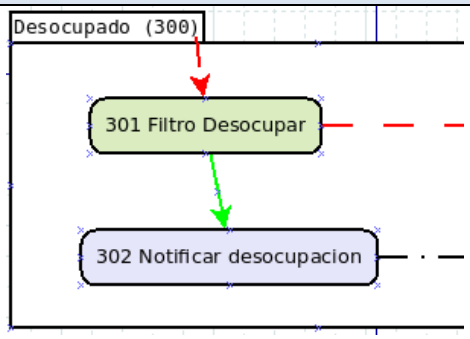
Figura 4-4 Diagrama de estados COSA del servidor de censado de butacas

El diagrama debe representar las reglas especificadas en BNF, a continuación se realiza una breve explicación de cada sección del diagrama.

rCensar ocupación::= <Censar ocupación>	
	En esta regla se ejecuta forzosamente el censado de la ocupación y se designa alguna de las dos reglas siguientes *rOcupar* o *rDesocupar*

rOcupado::= <Filtro ocupar> <Notificar ocupación>?	
	Una regla que contiene varios estados se numera con un identificador (generalmente múltiplo de 100) y sus estados internos tienen una numeración basada en dicho identificador. En esta regla el estado *Filtro ocupar* se ejecuta una vez y designa si *notificar ocupación* se ejecuta o no. Note como de *notificar ocupación* tiene una transición de entrada de *true* y una transición de salida *either*, esto indica que se ejecuta una o cero veces (símbolo ?)

rPetición::= <Censar petición> (<Activar petición><espera larga><desactivar petición> | <espera corta>)	
	En esta regla **rPetición** (identificada con 200) como primera acción se tiene la ejecución de Censar petición y se decide cuál de los comportamientos siguientes se ejecutará. Observe que la decisión se plasma en el estado *Activar petición*, esto se basa en la especificación BNF **<Activar petición>…** | **<espera corta>** que se interpreta como "si no se ejecuta *Activar petición* entonces se navegara a *Espera corta*"

rDesocupado::= <Filtro desocupar> <Notificar desocupación>?	
	En esta regla el estado *Filtro desocupar* se ejecuta una vez y designa si *notificar desocupación* se ejecuta o no. Note como de *notificar desocupación* tiene una transición de entrada de *true* y una transición de salida *either*, esto indica que se ejecuta una o cero veces (símbolo ?) En caso de que *Filtro desocupar* no se haya ejecutado se incluye un estado de manejo de errores.

4.1.4. Especificación de tabla de reglas BNF

Basándose en la especificación BNF y en el diagrama de estados COSA se identifican las reglas y sus estados correspondientes, es importante identificar las ocurrencias de los estados (símbolos ?, +, *), se añade un campo de *Trace* correspondiente al identificador de la regla.

Rule	**State**	**Trace**
rCensar ocupación	<Censar ocupación>	000
rOcupado	(<Filtro ocupar> <Notificar ocupación> ?) ?	100
rPetición	(<Censar peticion> <Activar peticion> ? <Espera larga> ? <Desactivar petición> ? <Espera corta> ?) *	200
rDesocupado	(<Filtro desocupar> <Notificar desocupación> ?) ?	300

En seguida se expande a la tabla de reglas BNF, llenando las columnas: *Rule, State static, true action y False action*; observe como los estados con una ocurrencia forzosa hace referencia a la misma acción tanto en *True action* como en *False action*, en los casos de ocurrencias cero, uno o varias en la columna *True action* hace referencia a la acción normal y en *False action* hace referencia a una acción nula (<Ignorar>).

Para llenar las columnas *Next true* y *Next false* es muy útil basarse en el diagrama de estados COSA, donde las transiciones indican el próximo estado de la siguiente manera:

- Las transiciones *True* (coloreadas en verde) indican el valor de *Next true*,
- las transiciones *False* (coloreadas en rojo) indican el valor de *Next false* y
- las transiciones *Either* (coloreadas en negro) indican que el valor *Next true* y *Next false* es el mismo

Por último se puede agregar un estado especial de manejo de errores (Ver la Tabla 4-1)

Rule	State Static	True action	Next true	False action	Next false	Trace
rCensar ocupación						
1	iCensar ocupación	<Censar ocupación>	2	<Censar ocupación>	2	000
rOcupado						
2	iFiltro ocupar	<Filtro ocupar>	3	<Ignorar>	9	101
3	iNotificar ocupación	<Notificar ocupación>	4	<Ignorar>	4	102
rPetición						
4	iCensar petición	<Censar petición>	5	<Censar petición>	5	201
5	iActivar petición	<Activar petición>	6	<Ignorar>	8	202
6	iEspera larga	<Espera larga>	7	<Ignorar>	11	203
7	iDesactivar petición	<Desactivar petición>	4	<Ignorar>	11	204
8	iEspera corta	<Espera corta>	1	<Ignorar>	1	205
rDesocupado						
9	iFiltro desocupar	<Filtro desocupar>	10	<Ignorar>	11	301
10	iNotificar desocupación	<Notificar desocupación>	8	<Ignorar>	8	302
Manejo de errores						
11	iError	<Error>	1	<Desconocido>	1	999

Tabla 4-1 Tabla de reglas BNF extendidas del servidor de censado de butacas

4.1.5. Codificación de la aplicación

Arquitectura de la aplicación

La arquitectura de la aplicación del servidor de censado de butacas es de tipo stand alone, cuya estructura interna consiste en dos paquetes generales: Control butacas contiene elementos referentes al control de butacas como un conjunto y COSAengine que contiene los elementos relacionados directamente con el modelo COSA se muestra en la Figura 4-5. La codificación completa de la aplicación se encuentra en al anexo 1, en la cual se incluye las reglas BNF, el *engine* COSA y los procedimientos de soporte

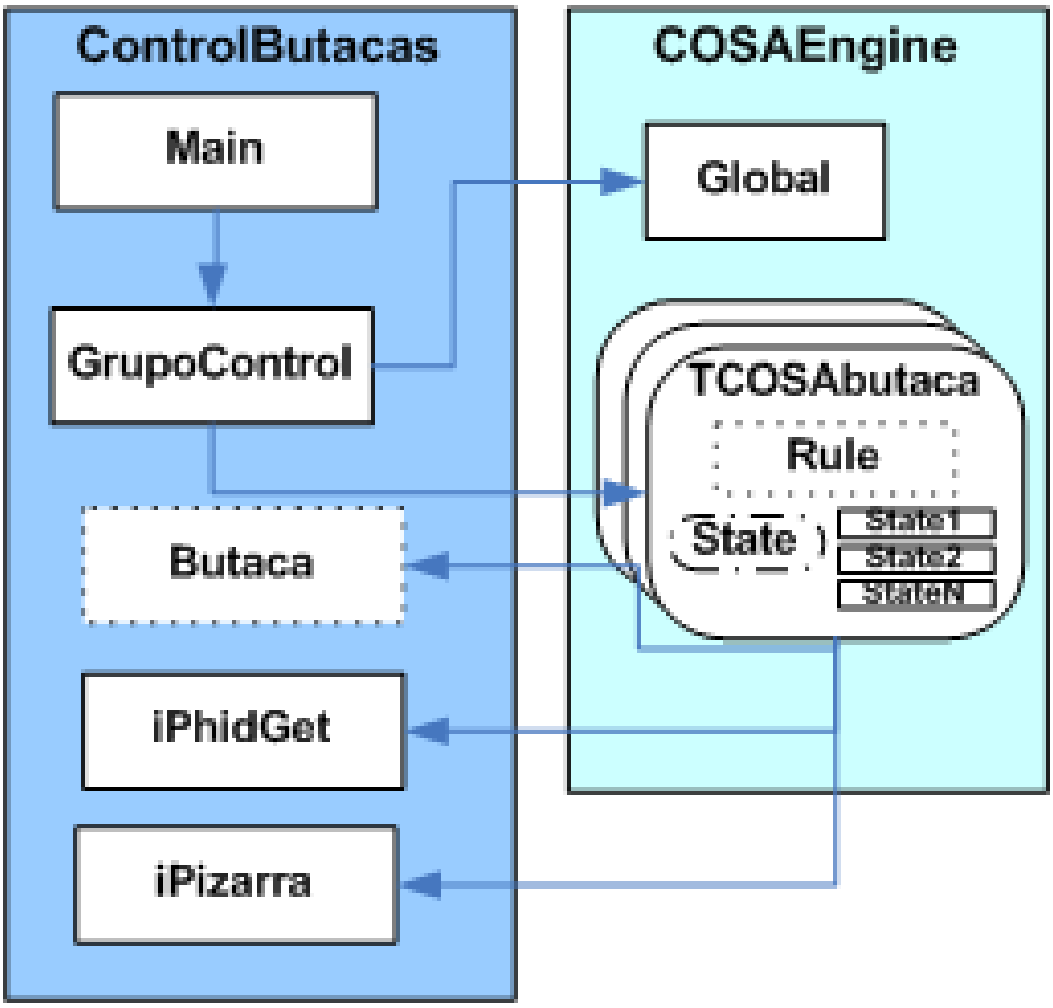


Figura 4-5 Arquitectura interna del servidor de censado de butacas

Los elementos involucrados en la aplicación son los listados en la Tabla 4-2.

Elemento	**Tipo**	**Descripción**
Main	Clase	Clase principal donde se crea una instancia del GrupoControl y se inicia su acción principal
GrupoControl	Clase	Es una clase de control que inicializa el iPhidget, asocia las butacas a sus puertos correspondientes del iPhidget y a un TCOSAbutaca. Inicia la ejecución del conjunto de TCOSAbutacas.
Butaca	Registro	Es una clase con atributos públicos que almacena datos referentes a los valores de las butacas.
iPhidget	Clase uso estático	Es una clase que contiene las operaciones relacionadas al control de puertos de la tarjeta Phidget.
iPizarra	Clase uso estático	Es una clase que contiene la configuración y las operaciones relacionadas con las notificaciones de estado a la Pizarra.
Global	Clase uso estático	Es una clase que se almacenan variables globales
TCOSAbutaca	Clase-hilo de ejecución	Es un hilo de ejecución que controlará el censado de una butaca. Se genera un TCOSAbutaca por cada butaca como un hilo de ejecución (thread)
Rule	Registro	Es una clase que almacena los valores correspondientes a una regla de tabla de reglas BNF
State	Interface	Es una interface que permite vincular dinámicamente una regla a un procedimiento de soporte.
State (1..N)	Clases	Clases que implementan la interface *State*, y corresponden a los procedimientos de soporte de la aplicación. Las acciones codificadas en dichos procedimientos deben mantener una alta cohesión.

Tabla 4-2 Descripción de los elementos del servidor de censado de butacas

COSA Engine

El elemento *engine* de COSA es reutilizable, cuyo algoritmo puede traducirse de manera natural al lenguaje de programación utilizado (Ver la Tabla 3-1). En Java, el *engine* sería el siguiente:

```
public void run()
{

    engineLocal=true;
    dynamicState=iCensoOcupacion;

    while (engineLocal && Global.engineGlobal)
    {
        System.out.println("Tiempo: "+System.currentTimeMillis()+" Estado:"+rRule[iTime].iTrace);
        if(dynamicState==rRule[iTime].iState)
        {
            rRule[iTime].pTrueRule.execute();
            iTime=rRule[iTime].iTrueRule;
        }
        else
        {
            rRule[iTime].pFalseRule.execute();
            iTime=rRule[iTime].iFalseRule;
        }

    }
}
```

Traducción de tabla de reglas BNF

La lógica plasmada en la tabla de reglas BNF (Ver la Tabla 4-1) se traduce directamente a la estructura de datos usada, en este caso se usó un vector de registro tipo Regla, cuidando los aspectos de implementación propios del lenguaje, como la numeración de los índices y aspectos de identificadores e inicialización de elementos. En la Figura 4-6 se muestra la traducción a lenguaje Java.

```
                //iTime      StaticSate     pTrueRule             iTrueRule   pFalseRule           iFalseRule       iTrace
rRule[0]=new Rule(0,    iCensoOcupacion,    mCensar_ocupacion,       1,       mCensar_ocupacion,    1,               000);
//100
rRule[1]=new Rule(1,    iFiltroOcupar,      mFiltrar_ocupacion,      2,       mIgnorar,             8,               101);
rRule[2]=new Rule(2,    iNotificaOcupacion, mNotificar_ocupacion,    3,       mIgnorar,             3,               102);
//200
rRule[3]=new Rule(3,    iCensoPeticion,     mCensar_peticion,        4,       mCensar_peticion,     4,               201);
rRule[4]=new Rule(4,    iActivaPeticion,    mActivar_peticion,       5,       mIgnorar,             7,               202);
rRule[5]=new Rule(5,    iEsperaLarga,       mEspera_larga,           6,       mIgnorar,            10,               203);
rRule[6]=new Rule(6,    iDesactivaPeticion, mDesactivar_peticion,    3,       mIgnorar,            10,               204);
rRule[7]=new Rule(7,    iEsperaCorta,       mEspera_corta,           0,       mEspera_corta,        0,               205);
//300
rRule[8]=new Rule(8,    iFiltroDesocupar,   mFiltrar_desocupacion,   9,       mIgnorar,             10,              301);
rRule[9]=new Rule(9,    iNotificaDesocupacion, mNotificar_desocupacion, 7,    mIgnorar,             7,               302);
//900
rRule[10]=new Rule(10,   iError,            mError,                  0,       mDesconocido,         0,               901);
```

Figura 4-6 Implementación de tabla de reglas BNF extendidas

Procedimientos de soporte

.Los procedimientos de soporte usan la vinculación dinámica para funcionar en coordinación con la tabla de reglas BNF extendida, para aplicar este principio en Java se uso una interface que será implementada por clases que corresponderá a los procedimientos de soporte. Dicha interface se implementa como se muestra en la Figura 4-7.

```
//Interfaz de estados
public interface State {
    public void execute();
}
```

Figura 4-7 Interface de estado para la vinculación dinámica

Los estados se codifican en base a los símbolos terminales de la especificación BNF (Ver la Figura 4-3) de la aplicación, su especificación puede ser relativamente informal, generalmente cada símbolo no terminal es un conjunto de acciones simples y muy relacionadas entre sí.

```
//Estados
class Censar_ocupacion implements State  {

    public void execute(){
        //Aqui lee los puertos del phidget
        //Lee el puerto correspondiente del phidget al boton B
        boolean ocupacion=false;
        ocupacion=InterfacePhidget.Censar_entrada(mButaca.num_entrada_B);
        if(ocupacion)
            dynamicState=iFiltroOcupar;//Indicar estado siguiente
        else
            dynamicState=iFiltroDesocupar;//Indicar estado siguiente
    }
}

class Filtrar_ocupacion implements State  {
    @Override
    public void execute(){
        if(mButaca.ocupacion!=1) //un no se actualiza la ocupacion
            dynamicState=iNotificaOcupacion;//Indicar estado siguiente
        else //La ocupacion ya esta actualizada
            dynamicState=iCensoPeticion;//Indicar estado siguiente
    }
}
```

También se implementan procedimientos de soporte para no realizar acciones y para el manejo de errores en la ejecución de la aplicación.

```
class Ignorar implements State  {
    @Override
    public void execute(){
          //No realizar acciones
    }
}

class Error implements State  {
    @Override
    public void execute(){
        System.out.print("Error en el censado de la butaca "+mButaca.id);
        engineLocal=false;
    }
}

class Desconocido implements State  {
    @Override
    public void execute(){
        System.out.print("Situacion anormal");
        engineLocal=false;
    }
}
```

5. Resultados

Durante el desarrollo de este trabajo se demostró la factibilidad de usar el paradigma COSA mediante la implementación y documentación de un proyecto piloto "Servidor de censado de butacas" el cual está actualmente funcionando en el aula inteligente de AmiLab del ITSZN.

El proyecto piloto desarrollado bajo el paradigma COSA fue comparado con otros proyectos similares desarrollados en AmiLab para el control de dispositivos en el entorno. Tomando en cuenta el tamaño de la aplicación el número de sentencias de control usadas se obtuvieron los siguientes datos comparativos:

Aplicación	**Lenguaje**	**LOC**	**ITE**		**% reuse**	**Complejidad ciclomática**[3]	**Complejidad /LOC**
			ciclos	**Bifurcaciones**			
Servidor de censado de butacas	Java	366	3	8	33.6 (123/366)	12 [4]	0.03
Control de puerta	Java	225	2	16	19.5 (44/225)	19	0.08
Control de dispositivo LCD	Java	192	2	5	13.4 (26/192)	8	0.04
Manejador sofá	Java	57	1	4	0.0 (0/57)	6	0.11
Control camera	Java	219	1	14	15.9 (35/219)	16	0.07
Control persianas	Java	577	24	49	5.5 (32/577)	74	0.13

[3] La complejidad ciclomática es una métrica de software que indica la complejidad del software en base al número de "caminos" independientes a través del código de un programa.

[4] Para calcular esta complejidad se uso la fórmula simplificada de McCabe: $v(G)= \pi + 1$, donde π = cantidad de predicados

OPENIFKIT (control de sensores de luminosidad, sofás, puerta y presencia)	Java	671	3	83	9.3 (63/671)	87	0.13

En base a la tabla anterior, se pueden puntualizar las siguientes observaciones:

- El desarrollo de las aplicaciones en AmiLab utilizan funciones reutilizables respecto al uso de dispositivos Phidget y la comunicación con la Pizarra, es decir, la complejidad de la operación radica en el comportamiento frente a los estímulos más que en la comunicación con las entidades externas (dispositivos de hardware y Pizarra).
- La estructura de control predominante son las bifurcaciones (if y cases) debido a las validaciones de censado de los puertos de los dispositivos que frecuentemente son de manera serializada y no anidada.
- En relación entre el tamaño de la aplicación y su complejidad ciclomática [(12)], el servidor de censado de butacas resulta ser la menor de todas debido al uso de la tabla de reglas para implementar el control-flow.
- De las aplicaciones presentadas, el servidor de censado de butacas es el único que tiene un modelo documentado de solución.
- El Servidor de butacas tiene una cantidad de LOC relativamente alta, pero cabe señalar que casi el 13.6% del código corresponde a secciones reutilizables del *Engine*, procedimientos de soporte estándar y tabla de reglas de la implementación del paradigma COSA.

En general, se puede concluir que la implementación del paradigma COSA es factible en el desarrollo de aplicaciones para el aula inteligentes, remarcado que los elementos de COSA son potencialmente reutilizables en todas las demás aplicaciones.

El paradigma COSA ofrece una forma distinta de plantear modelos de solución mediante una aplicación de esfuerzo concentrada en el entendimiento de la lógica y formalizada con reglas BNF, lo que produce aplicaciones mejor planeadas y más estables.

6. Conclusiones y trabajo futuro

En conclusión, es indudable que es necesario establecer un método de desarrollo que permita generar aplicaciones en AmiLab de manera estable, repetible y coherente desde el diseño hasta su implementación.

El paradigma COSA es una alternativa muy conveniente a las necesidades del laboratorio, particularmente en la interacción con los dispositivos del entorno y la Pizarra. Los beneficios de mantener separados el control-flow y el data-flow se reflejan en la disminución de la complejidad en la aplicaciones.

Además, las características de dicho paradigma permiten tener aplicaciones acordes a un modelo de solución documentado. Adicionalmente, debido a la modularización requerida en la generación de los estados estáticos del comportamiento de la aplicación se obtiene como resultado un alto grado de reusabilidad.

En el futuro a corto plazo, se deberán reexaminar las aplicaciones actuales debido a una actualización de la Pizarra, lo que ofrecerá una muy buena oportunidad de generar más aplicaciones piloto de la implementación de COSA. También se planea generar pruebas formales de desempeño para obtener más datos de referencia sobre los beneficios de la implementación de COSA en el desarrollo de aplicaciones en AmiLab.

A largo plazo, es importante incluir la enseñanza del paradigma COSA a los estudiantes como un tópico avanzado de desarrollo de software y demostrar su aplicación práctica en el campo de la inteligencia ambiental.

7. Anexos

Anexo 1 – Código fuente del servidor de censado de butacas

Butaca.java

```
package Control_Butacas;

public class Butaca{
   public int id;
   public int ocupacion;
   public int peticion_A;
   public int num_entrada_A;
   public int num_entrada_B;

   public Butaca(int i, int ent_A, int ent_B){
      id=i;
      ocupacion=-1; //La ocupacion inicialmente esta indefinida
      peticion_A=-1; //La peticion inicialmente esta indefinida
      num_entrada_A=ent_A;
      num_entrada_B=ent_B;
   }
}
```

GrupoControl.java

```
package Control_Butacas;

import COSAengine.*;
import javax.swing.JOptionPane;

public class GrupoControl {
   private TCOSAbutaca vControlButaca[];
   private final int maxButacas=7; //Respetar el maximo de butacas por tarjeta phidget


   public GrupoControl(){
      //Crear el grupo de controles (TCOSAbutaca), uno por cada butaca
      vControlButaca=new TCOSAbutaca[maxButacas];
      Butaca mButaca;
      InterfacePhidget.Inicializa_phidget();
      for(int i=0; i<maxButacas; i++)
      {
        mButaca=new Butaca(i+1, i*2, i*2+1 );
        vControlButaca[i]=new TCOSAbutaca();
        vControlButaca[i].Asignar_butaca(mButaca);
      }
   }



   public void ejecutar()
   {
      Global.engineGlobal=true;
```

```
        for(int i=0; i<maxButacas; i++)
        {
          vControlButaca[i].start();
        }

        JOptionPane.showMessageDialog(null,"Presione el boton para finalizar el servidor","Servidor de
Butacas", JOptionPane.WARNING_MESSAGE);
        Global.engineGlobal=false;
    }
}
```

InterfacePhidget.java

```
package Control_Butacas;
import java.util.logging.Level;
import java.util.logging.Logger;
import com.phidgets.InterfaceKitPhidget;
import com.phidgets.Phidget;
import com.phidgets.PhidgetException;
import com.phidgets.event.AttachEvent;
import com.phidgets.event.AttachListener;

public class InterfacePhidget {

    static private InterfaceKitPhidget Iphidget;

    static public void Inicializa_phidget()
    {
        System.out.println("Inicializando Phidget");
        try {
            System.out.println(Phidget.getLibraryVersion());
            Iphidget = new InterfaceKitPhidget();
            Iphidget.addAttachListener(new AttachListener() {
                public void attached(AttachEvent ae) {
                    System.out.println("attachment of " + ae);
                }
            });

            Iphidget.openAny();
            System.out.println("waiting for InterfaceKit attachment...");
            Iphidget.waitForAttachment();
            System.out.println("Phidget OK");
        } catch (PhidgetException ex) {
            Logger.getLogger(InterfacePhidget.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
    static public boolean Censar_entrada(int PuertoEntrada)
    {
        boolean Censo=false;
        try
        {
            Censo = Iphidget.getInputState(PuertoEntrada);

        }
        catch (PhidgetException ex)
```

```
        {
          Logger.getLogger(InterfacePhidget.class.getName()).log(Level.SEVERE, null, ex);
        }
        return Censo;
    }
}
```

Main.java

```
package Control_Butacas;

public class Main {

    public static void main(String[] args) {
        GrupoControl mControlButacas =new GrupoControl();
        mControlButacas.ejecutar();
    }

}
```

Global.java

```
package COSAengine;

public class Global {
    public static boolean engineGlobal;
}
```

Rule.java

```
package COSAengine;

//Registro de reglas---------------------------------------------------------
    public class Rule
    {
          public int iRule;
          public int StaticState;
          public TCOSAbutaca.State TrueAction;
          public int NextTrue;
          public TCOSAbutaca.State FalseAction;
          public int NextFalse;
          public int iTrace;

        public Rule(int Rule, int StaticState, TCOSAbutaca.State TrueBehavior, int NextTrueRule,
TCOSAbutaca.State FalseBehavior, int NextFalseRule, int Trace){
          this.iRule=Rule;
          this.StaticState=StaticState;
          this.TrueAction=TrueBehavior;
          this.NextTrue=NextTrueRule;
          this.FalseAction=FalseBehavior;
          this.NextFalse=NextFalseRule;
          this.iTrace=Trace;
          }
      }
```

TCOSAbutaca.java (engine y tabla de reglas)

```
package COSAengine;

import Control_Butacas.InterfacePhidget;
import Control_Butacas.Butaca;
import java.util.logging.Level;
import java.util.logging.Logger;


public class TCOSAbutaca extends Thread
{
   //Atributos de TCOSAbutaca
//    private State EstadoActual;
   private int dynamicState;
   private boolean engineLocal;
   private Butaca mButaca;
   private int iTime;
   private Rule rRule[];

   //Valores de estados estaticos
   private int iCensoOcupacion=000;
   private int iFiltroOcupar=101;
   private int iNotificaOcupacion=102;
   private int iCensoPeticion=201;
   private int iActivaPeticion=202;
   private int iEsperaLarga=203;
   private int iDesactivaPeticion=204;
   private int iEsperaCorta=205;
   private int iFiltroDesocupar=301;
   private int iNotificaDesocupacion=302;
   private int iError=901;



   public TCOSAbutaca()
   {
      CreateRules();
   }

   //Asignar la butaca a controlar
   public void Asignar_butaca(Butaca B)
   {
      mButaca=B;
   }


   //Crear las serie de reglas en base al diagrama de estados
   public void CreateRules()
   {

      //Estados
      State mError= new Error();
      State mIgnorar =new Ignorar();
      State mCensar_ocupacion = new Censar_ocupacion();
```

```
        State mFiltrar_ocupacion=new Filtrar_ocupacion();
        State mNotificar_ocupacion=new Notificar_ocupacion();
        State mCensar_peticion=new Censar_peticion_A();
        State mActivar_peticion=new Activar_peticion_A();
        State mEspera_larga=new Espera_larga();
        State mDesactivar_peticion=new Desactivar_peticion_A();
        State mEspera_corta=new Espera_corta();
        State mFiltrar_desocupacion=new Filtrar_desocupacion();
        State mNotificar_desocupacion=new Notificar_desocupacion();
        State mDesconocido=new Desconocido();

        rRule=new Rule[12];
        //Rules pagina 62
                    //iRule     StaticSate    TrueAction          NextTrue   FalseAction                    NextFalse
iTrace
        rRule[0]=new Rule(0,  iCensoOcupacion,   mCensar_ocupacion,     1,   mCensar_ocupacion,   1,
                                  000);
        //100
        rRule[1]=new Rule(1,  iFiltroOcupar,     mFiltrar_ocupacion,    2,        mIgnorar,          8,
                          101);
        rRule[2]=new Rule(2,  iNotificaOcupacion, mNotificar_ocupacion,  3,   mIgnorar,          3,
                          102);
        //200
        rRule[3]=new Rule(3,  iCensoPeticion,    mCensar_peticion,      4,        mCensar_peticion,   4,
                          201);
        rRule[4]=new Rule(4,  iActivaPeticion,   mActivar_peticion,     5,        mIgnorar,          7,
                          202);
        rRule[5]=new Rule(5,  iEsperaLarga,      mEspera_larga,         6,        mIgnorar,          10,
                          203);
        rRule[6]=new Rule(6,  iDesactivaPeticion, mDesactivar_peticion,  3,   mIgnorar,          10,
                          204);
        rRule[7]=new Rule(7,  iEsperaCorta,      mEspera_corta,         0,        mEspera_corta,      0,
                          205);
        //300
        rRule[8]=new Rule(8,  iFiltroDesocupar,  mFiltrar_desocupacion, 9,   mIgnorar,          10,
                          301);
        rRule[9]=new Rule(9,  iNotificaDesocupacion, mNotificar_desocupacion, 7,        mIgnorar,          7,
                                  302);
        //900
        rRule[10]=new Rule(10,  iError,          mError,                0,       mDesconocido,       0,
            901);

    }

    //Motor de TCOSA que manipula las reglas
    @Override
    public void run()
    {
        engineLocal=true;
        dynamicState=iCensoOcupacion;

        while (engineLocal && Global.engineGlobal)
        {
            System.out.println("Butaca: "+mButaca.id+" Tiempo: "+System.currentTimeMillis()+"
Estado:"+rRule[iTime].iTrace);
```

```
            if(dynamicState==rRule[iTime].StaticState)
            {
               rRule[iTime].TrueAction.execute();
               iTime=rRule[iTime].NextTrue;
            }
            else
            {
               rRule[iTime].FalseAction.execute();
               iTime=rRule[iTime].NextFalse;
            }

         }
      }
```

TCOSAbutaca.java (Estados y procedimientos de soporte)

```
//Estados dentro de TCOSA ----------------------------------------------------
//Interfaz de estados
public interface State {
   public void execute();
}

//Estados
class Censar_ocupacion implements State  {
   public void execute(){
      //Aqui lee los puertos del phidget
      //Lee el puerto correspondiente del phidget al boton B
      boolean ocupacion=false;
      ocupacion=InterfacePhidget.Censar_entrada(mButaca.num_entrada_B);
      if(ocupacion)
         dynamicState=iFiltroOcupar;//Indicar estado siguiente
      else
         dynamicState=iFiltroDesocupar;//Indicar estado siguiente
   }
}

class Filtrar_ocupacion implements State  {
   @Override
   public void execute(){
      if(mButaca.ocupacion!=1) //un no se actualiza la ocupacion
         dynamicState=iNotificaOcupacion;//Indicar estado siguiente
      else //La ocupacion ya esta actualizada
         dynamicState=iCensoPeticion;//Indicar estado siguiente
   }
}

class Notificar_ocupacion implements State  {
   @Override
   public void execute(){
      mButaca.ocupacion=1;
      //Aqui se conecta con la pizarra y le notifica que la silla esta ocupada
      System.out.println("Pizarra: Ocupación activada en butaca "+mButaca.id);
   }
}
```

```
class Filtrar_desocupacion implements State  {
   @Override
   public void execute(){
      if(mButaca.ocupacion!=0) //aun no se actualiza la desocupacion
         dynamicState=iNotificaDesocupacion;//Indicar estado siguiente
      else //La desocupacion ya esta actualizada
         dynamicState=iEsperaCorta;//Indicar estado siguiente
   }
}

class Notificar_desocupacion implements State  {
   @Override
   public void execute(){
      mButaca.ocupacion=0;
      //Aqui se conecta con la pizarra y le notifica que la silla esta desocupada
      System.out.println("Pizarra: Ocupación desactivada en butaca "+mButaca.id);
   }
}

class Censar_peticion_A implements State  {
   @Override
   public void execute(){
      //Lee el puerto correspondiente del phidget al boton A
          boolean peticion_A=false;
          peticion_A=InterfacePhidget.Censar_entrada(mButaca.num_entrada_A);
      if(peticion_A)
         dynamicState=iActivaPeticion;//Indicar estado siguiente
      else
         dynamicState=iEsperaCorta;
   }
}

class Activar_peticion_A implements State  {
   @Override
   public void execute(){
      mButaca.peticion_A=1;
      //Aqui se conecta con la pizarra y le notifica que la silla realiza una peticion
      //El boton es de tipo push-release
      System.out.println("Pizarra: Peticion activada en butaca "+mButaca.id);
      dynamicState=iEsperaLarga;//Indicar estado siguiente
   }
}

class Desactivar_peticion_A implements State  {
   @Override
   public void execute(){

      mButaca.peticion_A=0;
      //Aqui se conecta con la pizarra y le notifica que la silla desactivo la peticion
      System.out.println("Pizarra: Peticion desactivada en butaca "+mButaca.id);
      dynamicState=iCensoPeticion;//Indicar estado siguiente
   }
}

//Estados especiales ------------------------------------------------------
```

```
class Ignorar implements State  {
   @Override
   public void execute(){   //No realizar acciones
   }
}

//Estados de manejo de errores ----------------------------------------------
class Error implements State  {
   @Override
   public void execute(){
      System.out.print("Error en el censado de la butaca "+mButaca.id);
      engineLocal=false;
   }
}

class Desconocido implements State  {
   @Override
   public void execute(){
      System.out.print("Situacion anormal");
      engineLocal=false;
   }
}

//Estados de control de hilo ------------------------------------------------
class Espera_corta implements State  {
   @Override
   public void execute(){
         try {
            Esperar(500);
         } catch (InterruptedException ex) {
            Logger.getLogger(TCOSAbutaca.class.getName()).log(Level.SEVERE, null, ex);
         }
   }
}

class Espera_larga implements State  {
   @Override
   public void execute(){
         try {
            Esperar(3000);
         } catch (InterruptedException ex) {
            Logger.getLogger(TCOSAbutaca.class.getName()).log(Level.SEVERE, null, ex);
         }
         dynamicState=iDesactivaPeticion;//Indicar estado siguiente
   }
}

public void Esperar(int duracion) throws InterruptedException
   {
      sleep(duracion);
   }
}
```

8. Referencias bibliográficas

1. **Schmidt, Douglas C.** *Model driven engineering.* s.l. : IEEE Computer 39, Febrero 2006.

2. **Lethbridge, Andrew Forward y Timothy C.** *Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals.* s.l. : Proceedings of the 2008 international workshop on Models in software engineering, 2008.

3. **Shoham, Yoav.** *Agent-Oriented Programming, Artificial Intelligence.* s.l. : Elsevier Science Publishers Ltd, 1993.

4. **Henderson-Sellers, Brian y Giorgini, Paolo.** *Agent-oriented methodologies.* s.l. : Idea Group Inc (IGI), 2005.

5. *Argumentation-Based Agent Interaction in an Ambient-Intelligence Context.* **Moraitis, Pavlos y Spanoudakis, Nikolaos.** 6, s.l. : IEEE Intelligent Systems, 2007, Vol. 22.

6. *Realizing an Agent-oriented Middleware for Heterogeneous Sensor Networks.* **Muldoon, Conor, y otros.** s.l. : In Proceedings of the ACM/IFIP/USENIX international Middleware Conference on Middleware, Diciembre 2008.

7. *Agent-oriented Knowledge Management in Learning Environments: A Peer-to-Peer Helpdesk Case Study.* **Guizzard, Renata S. S., Aroyo, Lora y Wagner, Gerd.**

8. **Cunneyworth, Wayne.** *Table Driven Design - A development strategy for minimal maintenance information systems.* 1994.

9. **Bentley, Jon Louis.** *Programming Pearls.* s.l. : Addison-Wesley, 2000.

10. **McConnell, Steve.** *Code Complete 2.* s.l. : Microsoft Press, 2004.

11. **Gordon, Morrison E.** *Breaking the time barrirer – The temporal engineering of software.* s.l. : Outskirt Press Inc., 2009.

12. *A Complexity Measure.* **McCabe, Tomas J.** 308–320, s.l. : IEEE Transactions on Software Engineering, December 1976.