Modelo para aplicaciones sensibles al contexto (MASCO): Formalización de las capas relacionadas con datos tomados desde el contexto.

María del Pilar Galvez Díaz[1,2], Nélida Raquel Caceres[1]

1 Universidad Nacional de Jujuy, Facultad de Ingeniería, San Salvador de Jujuy, Jujuy, Argentina
2 Universidad Católica de Santiago del Estero, Departamento Académico San Salvador, San Salvador de Jujuy, Argentina
{mdpgalvezdiaz,nrcaceres}@fi.unju.edu.ar

Resumen. El modelo de Aplicaciones Sensibles al Contexto (MASCO) presenta una arquitectura en capas que provee servicios para aplicaciones sensibles al contexto, considera el tratamiento de más de una variable de contexto, la relación de una entidad con más de una variable de contexto, como así también la interacción entre variables de contexto y entidades. El modelo fue validado usando diferentes experimentaciones las cuales permitieron su refinamiento y determinar sus atributos de calidad. El modelado se realizó usando UML, al ser un lenguaje visual, sus diagramas no siempre son lo suficientemente expresivos como para especificar todos los aspectos que son relevantes. En este trabajo se presenta una introducción a la formalización de MASCO utilizando OCL de forma que el modelo no presente ambigüedades en su interpretación y uso como framework en el desarrollo de sistemas Context Aware.

Palabras Clave: Context Aware, modelado, UML, OCL, MASCO.

1 Introducción

El grupo de investigación de Aplicaciones Sensibles al Contexto (GRISECO) ha desarrollado un Modelo para Aplicaciones Sensibles al Contexto (MASCO), orientado a objetos basado en capas.

MASCO considera el tratamiento de más de una variable de contexto, la relación de una entidad con más de una variable de contexto, como así también la interacción entre variables de contexto y entidades. El modelo fue validado usando diferentes experimentaciones las cuales permitieron su refinamiento y determinar sus atributos de calidad como modelo de arquitectura.

MASCO fue modelado usando UML, al ser un lenguaje visual, los diagramas de UML no siempre son lo suficientemente expresivos como para especificar todos los aspectos que son relevantes en una especificación, por ejemplo la especificación de restricciones adicionales con respecto a los objetos que pertenecen a un modelo. Ante la falta de un lenguaje de especificación formal de restricciones, estas se describen usando lenguaje natural lo cual introduce ambigüedades. El propósito de este trabajo

es presentar una introducción a la formalización de MASCO utilizando OCL de forma que el modelo no presente ambigüedades en su interpretación y aplicación como framework en el desarrollo de sistemas Context Aware.

En el apartado 2 se describe MASCO, en el apartado 3 se describe OCL, en el apartado 4 se comienza con la formalización del modelo, en el apartado 5 se presentan las conclusiones, y finalmente en el apartado 6, se detallan las referencias.

2 MASCO

MASCO es un modelo de arquitectura en capas, modelado con UML, orientado a objetos, que fue desarrollado con el objetivo de ser usado como framework para desarrollar sistemas sensibles al contexto, Fig. 1, y fue evolucionando y refinándose a través de los trabajos presentados en [1] [2], [3], [4] y [5].

En MASCO se identifican cinco capas: Application Layer: se encuentran los objetos del dominio de la aplicación; Context Layer: contiene los objetos necesarios para procesar la información de contexto; Service Layer: contiene los objetos necesarios para proveer servicios tanto internos como externos al sistema; Sensing Concerns Layer: se encarga de interpretar o traducir los datos que provienen de la capa Hardware Abstractions; Hardware Abstractions Layer: esta capa agrupa los objetos que representan los sensores y actuadores.

Los trabajos realizados permitieron determinar además los atributos de calidad de MASCO y atributos o requisitos técnicos de calidad no funcionales: Integridad Conceptual, Corrección, Compleción, Capacidad de realización, Modificabilidad, Testeabilidad, Simplificación de la funcionalidad, Reusabilidad, y Portabilidad [5].

Descripción del funcionamiento de las capas relacionadas con datos tomados desde el contexto. Cada vez que una entidad cambia su posición o una variable de contexto ingresa un nuevo valor, situaciones que se reflejan en un objeto IRport, un objeto SensingConcern es notificado. SensingConcern abstrae las reglas de sensado, representadas por una subclase SensingPolicy. El objeto SensingPolicy analiza el valor y, si es necesario, produce alguna transformación en los datos y, posteriormente, notifica a ContextAggregator. Si se trata de un valor de ubicación busca a qué objeto pertenece esa ubicación en Context Layer (ej: Office, Corridor). En esta capa se estableció la clase Context (Fig. 1) con la subclases Location (para considerar la variable de contexto Ubicación tanto de un usuario como otras entidades relevantes para la aplicación) [1], y AnotherVariable1, AnotherVariable2… AnotherVariableN (para considerar otras variables de contexto tales como presión, temperatura, nivel de agua, que son inherentes, por ejemplo, a los procesos industriales o aplicaciones médicas). En tiempo de implementación Sensing Concern actúa como un patrón Strategy y un patrón Observer y ContextAggregator como un patrón Observer [6].

En Application Layer la clase Discoverer es la responsable de mantener la información sobre los procesos, entidades y variables inherentes a una aplicación y la clase EntityAggregator es la responsable de coordinar la interacción entre objetos Entity [3]. Esta clase, en tiempo de implementación, constituye un patrón Mediator. Cada objeto Process puede estar relacionado con uno ó más objetos Entity. A su vez,

cada uno de éstos puede estar relacionado con un objeto ContextAggregator. Cuando una variable de contexto cambia su valor éste determina de qué manera deben reaccionarse las variables restantes. Una vez definido el comportamiento, solicita a ServiceCoordinator la ejecución de un determinado servicio.

En muchas oportunidades la actuación de ServiceCoordinator involucra la solicitud de un actuador (ej: abrir una compuerta, elevar temperatura); por lo tanto se puede hacer una distinción entre objetos Actuator y Sensor que aparecen en el modelo como subclases de IRPort.

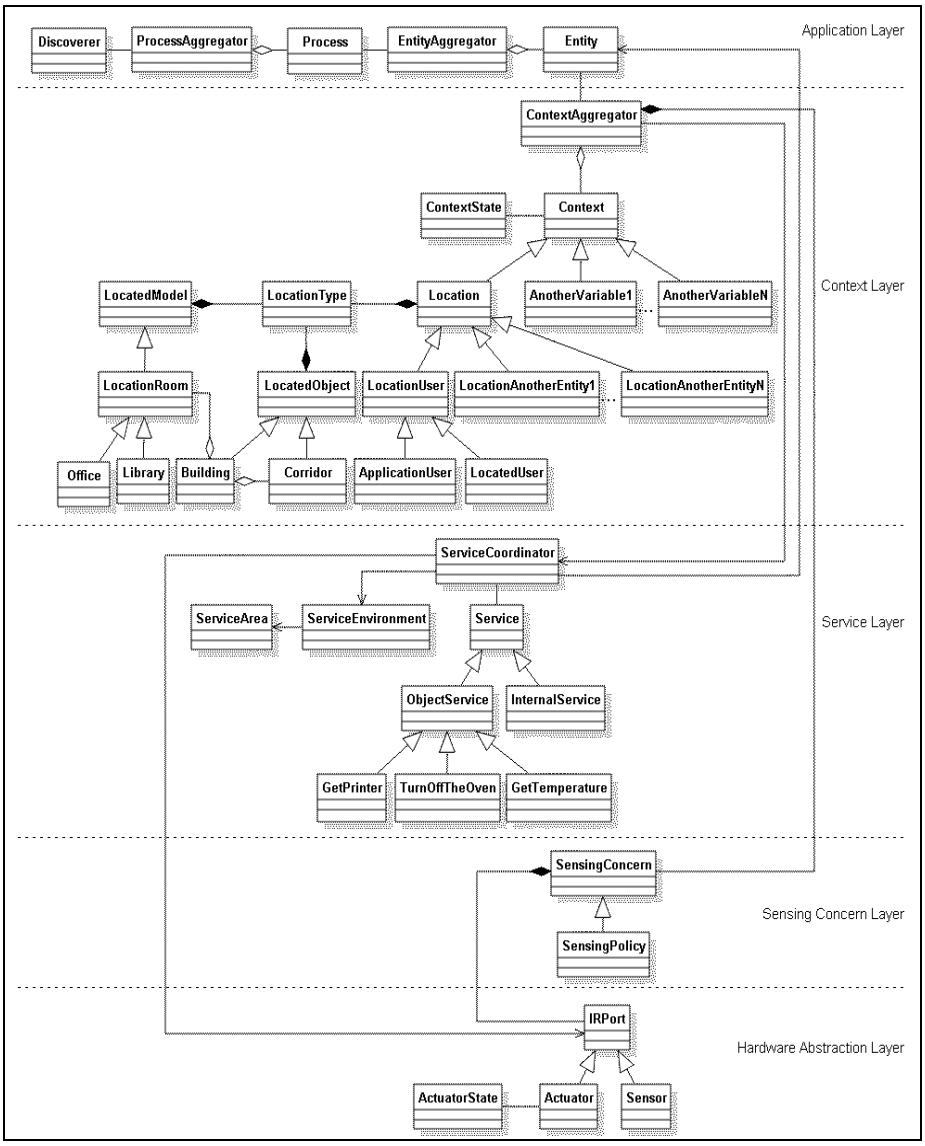

Fig. 1. Modelo MASCO

Cada clase que representa un objeto Actuator contiene operaciones que hacen referencia a los servicios que brinda el objeto real.

La clase Actuator representa la generalización de los actuadores y éstos en general accionan en base a sus estados, por esta razón se incorporó la clase ActuatorState, que en tiempo de implementación actúa como un patrón State [6] [7].

3 OCL (Object Constraint Language)

Un lenguaje gráfico de definición de modelos, como UML, proporciona una idea general e intuitiva, generalmente fácil de comprender por parte del usuario, de los conceptos del dominio de referencia que se está modelando y de las relaciones entre ellos. Sin embargo, estos lenguajes no son lo suficientemente expresivos como para definir toda la información relevante del dominio de referencia, estos aspectos hacen referencia a requisitos establecidos en éste y también tienen que ser definidos en el modelo ya que de lo contrario la definición del dominio sería incompleta.

Por este motivo, el uso de lenguajes gráficos de modelado se complementa con el uso de otros lenguajes, normalmente textuales aunque basados en un formalismo lógico, que permitan definir esta información de manera precisa y no ambigua. El lenguaje OCL (Object Constraint Language), lenguaje de restricción de objetos es el más conocido de ellos. OCL ha sido propuesto por el Object Management Group [8], y recientemente ha sido adoptado como estándar por el ISO/IEC [9].

Las características principales de OCL que se definen en [10] son: a) Es un lenguaje de definición de expresiones, a partir de las cuales pueden definirse condiciones que un modelo debe satisfacer; b) Es un lenguaje formal, en el sentido que define expresiones con una semántica precisa y no ambigua, y que por lo tanto pueden ser evaluadas mediante herramientas automáticas; c) Es un lenguaje tipado, ya que cada expresión corresponde a un tipo de datos determinado y solo se pueden combinar en OCL expresiones de tipos compatibles entre sí; d) Es un lenguaje de especificación y no de programación.

OCL puede ser utilizado tanto para definir aspectos estructurales (o estáticos) de un modelo como para definir aspectos dinámicos (o de comportamiento). En la parte estructural de un modelo, OCL se usa para especificar las restricciones de integridad que el diagrama de clases debería satisfacer y también para definir la información derivada de este diagrama. En la parte del comportamiento, OCL se usa para definir el efecto de la ejecución de una operación sobre la base de información (mediante expresiones que especifican las precondiciones y las postcondiciones de dicha operación) y para definir consultas sobre la base de información.

Este lenguaje es un lenguaje puro de especificación, por lo que las expresiones escritas en él no tienen efectos secundarios, es decir, no modifican el estado del modelo consultado [11]. Por lo tanto OCL es una extensión estándar a UML que permite realizar lo siguiente: [12]

- Al ser un lenguaje de consulta: especificar consultas para acceder a elementos de modelo y sus valores, y definir operaciones de consulta.
- Indicar restricciones en elementos de un modelo al definir reglas de negocio como restricciones en elementos de un modelo.

4 Planteamiento de Restricciones en MASCO

Las restricciones son planteadas en lenguaje natural, y a modo de introducirlas posteriormente en lenguaje OCL, se puede determinar las siguientes:

- **Restricciones del dominio**
  El modelo sólo puede ser utilizado por aplicaciones context-aware.
- **Restricciones del modelo**
  El modelo solo trabaja con objetos y con patrones de diseño
  El modelo permite trabajar con N variables de contexto
  El modelo permite trabajar con N entidades.
  El modelo permite trabajar con N procesos
  La clase Actuator existirá dependiendo del tipo de aplicación.

4.1 Restricciones en Hardware Abstraction Layer

El modelo trabaja con N variables de contexto, las cuales deben ser sensadas y de acuerdo a estos valores sensados se determina la intervención de un actuador, por lo cual es necesario cumplir con lo siguiente:

- Deben existir 1 a N sensores
- Deben existir 1 a N actuadores
- La actuación de los actuadores va a depender de los valores que sean sensados.

Los atributos y métodos son determinados de forma genérica para realizar la formalización de éste modelo y se muestran en la Fig. 2. Las restricciones de esta capa se presentan a continuación:

- **Clase IRPort**
  - *Atributos*
    - Los atributos en común del Sensor y actuador deben colocarse en esta clase.
  - *Métodos*
    - Los métodos en común del sensor y el actuador deben colocarse en esta clase.
- **Clase Sensor**
  - *Atributos*
    - Identificar el sensor del cual proviene el dato.
    - Identificar el tipo de dato sensado.
    - El valor sensado debe ser del tipo numérico real.
    - Deben existir 1 a N sensores que proveen los datos.
    - Toda identificación debe ser única.
    - La ubicación del sensor debe ser única.
  - *Métodos*
    - El valor leído desde el sensor no debe pertenecer al conjunto vacío.
- **Clase Actuator**
  - *Atributos*
    - Determinar el actuador al cual se envía la orden.
    - Pueden existir 1 a N actuadores.
    - Toda identificación debe ser única.
    - La ubicación del actuador debe ser única.

- *Métodos*
  - El cambio de estado se va a producir dependiendo del estado actual del actuador definido en StateActuator. Se utiliza el patrón diseño State que permite que un objeto modifique su comportamiento cada vez que cambie su estado interno [6]:
    - Si el estado del actuador no se corresponde con el estado actual del actuador la orden debe ser ejecutada.
    - Si el estado actual del actuador se corresponde con la orden de actuación, la aplicación debe anular la orden. Por ejemplo, si un motor está encendido no se puede solicitar que se lo encienda, en este caso la orden queda sin efecto.

- **Clase StateActuator**
  - *Atributos*
    - Representa el estado actual del actuador.
    - El estado debe ser booleano (por ejemplo, pueden encontrarse los estados: abierto/cerrado, encendido/apagado, activado/desactivado, entre otros).
  - *Métodos*
    - Realiza la consulta del estado actual del actuador.
- **Relación de Asociación**
  - Es necesario definir la relación de asociación entre los objetos Actuator y StateActuator: "Un estado corresponde a 1 o N actuadores" (por ejemplo, el estado "encendido" puede corresponder a distintos actuadores: encendida la ventilación, encendida la luz, entre otros).

Especificación con OCL. Luego de plantear las restricciones en lenguaje natural, se puede declarar el contexto especificando el elemento del modelo en el cual será definida la restricción. Para las invariantes la declaración del contexto será una clase. Para las precondiciones y postcondiciones el contexto será una clase seguida de la signatura de la operación [13], utilizando las clases representadas en la Fig. 2 se puede iniciar una especificación usando OCL de la siguiente forma:

- **Context IRPort**

  inv: self.NumIdentification >0
  inv: self.NumIdentification notEmpty()
  inv: self.NumIdentificationÚnique
  inv: self.Address notEmpty()
  inv: self.LocationUnique
  --a cada sensor o actuador le corresponde una única identificación
  inv: UniqueNumIdentification:IRPort.AllInstances( ) → is Unique (NumIdentification)
  --a cada sensor o actuador le corresponde una única ubicación
  inv: UniqueAddress:IRPort.AllInstances( ) → is Unique(Address)
- **Context Sensor**

  inv: self.SensingValue notEmpty()
  inv: self.SensingValue = real
  inv: self.VariableType = "Digital" or self.VariableType = "Analogic"
  inv: self.ReadValue() = real implies notEmpty()
- **Context Actuator**

  inv: self.StateActuator
  --Es el que implementa el patrón de diseño State

inv: self.ChangeState() = true implies self.State = false or self.ChangeState() = false implies self.State = true

- **Context StateActuator**

  inv: self State = true or self State = false

  --*Es el que revisa en qué estado se encuentra el actuador*

  Context.StateActuator :: CheckStatus()

  Pre: state notEmpty()

  Post: state = true or state = false

En la Fig. 2 se puede observar que las clases Sensor y Actuator heredan desde la clase abstracta IRPort, lo cual puede ser representado en la tabla 1 según lo propuesto en [14]:

Tabla 1. Herencia expresada en OCL de Hardware Abstraction Layer.

Clase Abstracta	Subclases
IRPort:	**Sensor:**
Context IRPort	*Context Sensor*
Inv: self.oclIsKindOf (IRPort) = true	Inv: self.oclIsKindOf (IRPort) = true
Inv: self.oclIsTypeOf (IRPort) = true	Inv: self.oclIsTypeOf (IRPort) = false
Inv: self.oclIsTypeOf (Sensor) = false	Inv: self.oclIsTypeOf (Sensor) = true
Inv: self.oclIsKindOf (Sensor) = false	Inv: self.oclIsKindOf (Sensor) = true
Inv: self.oclIsTypeOf (Actuator) = false	Inv: self.oclIsTypeOf (Actuator) = false
Inv: self.oclIsKindOf (Actuator) = false	Inv: self.oclIsKindOf (Actuator) = false
	Actuator:
	Context Actuator
	Inv: self.oclIsKindOf (IRPort) = true
	Inv: self.oclIsTypeOf (IRPort) = false
	Inv: self.oclIsTypeOf (Actuator) = true
	Inv: self.oclIsKindOf (Actuator) = true
	Inv: self.oclIsTypeOf (Sensor) = false
	Inv: self.oclIsKindOf (Sensor) = false

La asociación "*corresponds to*" ("corresponde a") entre las clases StateActuator y Actuator se expresa de la siguiente forma:

- **Context StateActuador**

  Inv: self.State= true or self.State= false implies self.Set → notEmpty()

  Self.Set → collect (State)

4.2 Restricciones en Sensing Concern Layer

Esta capa recibe los valores sensados en la Hardware Abstraction Layer y los transforma, si es necesario, en datos con los cuales pueda trabajar la aplicación, por lo que las clases que se muestran en la Fig. 2 deben estar presentes:

- **Clase SensingConcern**
  - *Atributos*
    - Es necesario conocer el Identificador del sensor que está siendo leído.
- **Clase SensingPolicy**

o *Métodos*

- Los valores sensados llegan a esta capa a medida que son leídos por los sensores, en tiempo de implementación se utiliza el patrón de diseño Observer. [6].
- El valor sensado debe ser transformado si es necesario.
- La clase ContextAggregator en Context Layer debe ser notificada al finalizar la transformación (si fuera necesario) de los valores sensados.
- En caso de que el valor sensado necesite ser transformado debe aplicarse la política de conversión más adecuada. Para la situación en que sean leídos más de una variable de contexto o de ubicación, en tiempo de implementación es necesario utilizar el patrón de diseño Strategy [6] que determina cual es la política de transformación adecuada para cada uno de los datos sensados que necesite ser transformado.

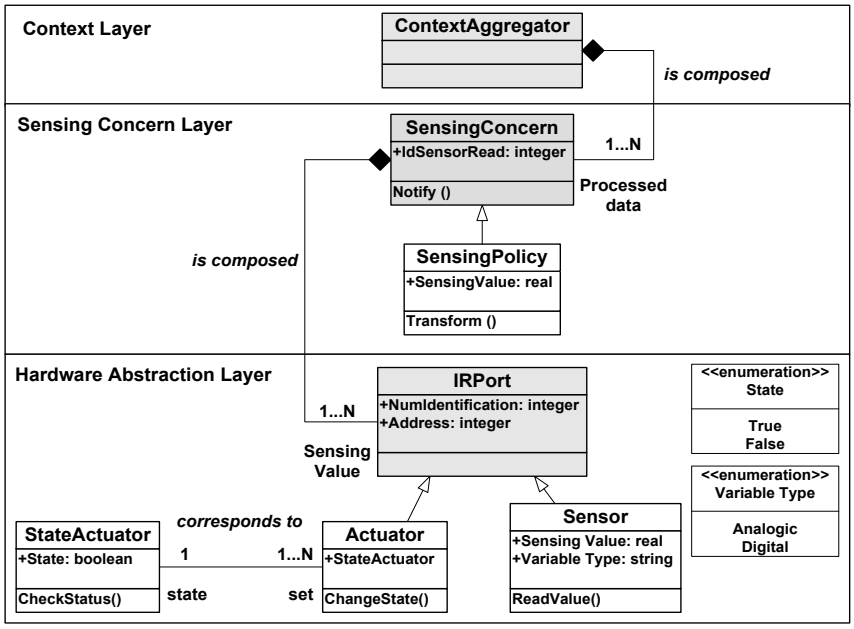

Fig. 2. MASCO con atributos, métodos genéricos y relaciones entre capas..

Especificación con OCL. Utilizando las clases representadas en la Fig. 2 se puede iniciar una especificación usando OCL de la siguiente forma:

- **Context SensingConcern**

  Inv: self.IdSensorRead notEmpty()
  Inv: self.IdSensorRead = integer
  Inv: self.Notify()
  --*Con este método se notifica a Context Layer los datos procesados*

- **Context SensingPolicy**

  Inv: self.SensingValue notEmpty()
  Inv: self.SensingValue = real

Inv: self.Transform()
--Debe existir un valor a transformar
Pre: = SensingValue notEmpty()
--El valor sensado es transformado, el cual deriva en un dato procesado
Post: Context.SensingConcern :: SensingValue: real
Derive
Transformer(SensingValue) → processed Data

En la Fig. 2 se puede observar que la clase Sensing Policy hereda desde la clase Sensing Concern, lo cual puede ser representado en la tabla 2 según lo propuesto en [14]:

Tabla 2. Herencia expresada en OCL de Sensing Concern Layer.

Clase Abstracta	Subclases
SensingConcern:	**SensingPolicy:**
Context SensingConcern	*Context SensingPolicy*
Inv: self.oclIsKindOf (SensingConcern) =true	Inv: self.oclIsKindOf (SensingConcern) =true
Inv: self.oclIsTypeOf (SensingConcern)=true	Inv: self.oclIsTypeOf(SensingConcern)=false
Inv: self.oclIsTypeOf (SensingPolicy) = false	Inv: self.oclIsTypeOf (SensingPolicy) = true
Inv: self.oclIsKindOf (SensingPolicy) = false	Inv: self.oclIsKindOf (SensingPolicy) = true

4.3 Especificación con OCL de las relaciones entre las capas del modelo

Para representar las relaciones entre las capas del modelo se plantearon asociaciones cualificadas [12] y [14]. En la Fig. 2 se observan las siguientes composiciones:

- La clase SensingConcern en Sensing Concern Layer está compuesta por uno o varios valores que son enviados desde la clase IRPort en Hardware Abstraction Layer.
- La clase ContextAggregator en Context Layer está compuesta por uno o varios datos procesados que son notificados desde la clase SensingConcern en Sensing Concern Layer, para éste envío se emplea el método Notity().

Las restricciones expresadas en OCL son las siguientes:

- **Context SensingConcern**
  --Deben existir uno o varios valores sensados para que pueda ser procesado
  Inv: self.SensingValue [N].NumIdentification notEmpty()
- **Context ContextAggregator**
  --Los datos procesados deben ser notificados en Context Layer
  Inv: self.Processed data [N].SensingValue → Notify()

5 Conclusiones

La formalización de MASCO utilizando OCL, permite obtener un modelo libre de malas interpretaciones, con una arquitectura definida, con atributos de calidad determinados, el cual ya fue probado y validado en distintos ámbitos de aplicación, para ser usado en el desarrollo de aplicaciones Context Aware.

Se comenzó éste trabajo definiendo para el modelo solo los aspectos estructurales (o estáticos) de las capas Hardware Abstraction y Sensing Concern y las asociaciones entre ellas, continuando en futuros trabajos con la formalización completa de MASCO, considerando que además de los aspectos estructurales serán incorporados los aspectos de comportamiento (consultas).

6 Referencias

1. Quincoces, V.E., Gálvez, M.P., Cáceres, N.R., Vega, A.A., Ramos, H.O.: Extensión de un modelo en capas que provee servicios para aplicaciones sensibles al contexto. En: Investigaciones en Facultades de Ingeniería del NOA, Vol I, pp. 35-40, Cap. IV. EUNSa, Argentina (2009)
2. Gálvez, M.P., Quincoces, V.E., Cáceres, N.R., Vega, A.A.: Refinamiento de un Modelo en Capas que Provee Servicios de Ubicación para Aplicaciones Sensibles al Contexto. En: III Congreso Internacional de Telecomunicaciones, Tecnologías de la Información y las Comunicaciones, Quito (2010)
3. Gálvez, M.P., Brouchy C., González O., Cáceres, N.R., Quincoces, V.E.: Modelo que provee servicios para aplicaciones sensibles al contexto (MASCO): Interacción entre entidades. En: Investigaciones en Facultades de Ingeniería del NOA, pp. 1103-1109. Científica Universitaria, UNCa, Argentina (2011)
4. Gálvez, M.P., Cáceres, N.R., Brouchy C., Velázquez C.E., González O.M, Guzmán A.N., Romero, N.D., Quincoces, V.E.: Modelo que provee aplicaciones sensibles al contexto (MASCO): validación e inicio de evaluación. En: Investigaciones en Facultades de Ingeniería del NOA, pp. 146. Grupo Loza Impresiones S.R.L., UNT, Tucumán (2012)
5. Gálvez, M.P., Cáceres, N.R, Velázquez, C. E., Guzmán, A. N.: Atributos de calidad del modelo para aplicaciones sensibles al contexto MASCO. En: 5to Simposio Internacional de Investigación: "Interdisciplinariedad, Multidisciplinariedad y/o transdisciplinariedad: en la búsqueda de respuestas desde las experiencias de investicación". Universidad Católica de Santiago del Estero, DASS, Jujuy (2013)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable OO Software. Addison Wesley, USA (1995)
7. Velázquez, C. E., Guzmán, A. N., Gálvez, M.P., Cáceres, N.R.: Modelo para aplicaciones sensibles al contexto (MASCO): Un caso de estudio para validación. En: XIX Congreso Argentino de Ciencia de la Computación, Mar del Plata (2013)
8. Object Constraint Language (OCL), http:// www.omg.org/spec/OCL/
9. ISO/ IEC 19505-2:2012 – OMG UML superstructure 2.4.1, http/www.iso.org/
10. Garcia Molina, J., García Rubio, F., Pelechano, V., Vallecito, A., Vara, J. M., Vicente-Chicote, C.: Desarrollo de software dirigido por Modelos: conceptos, métodos y herramientas. Alfaomega, Rama, Mexico (2014)
11. UML 2.0 Superstructure Specification, 2003, http://www.omg.org/docs/ptc/03-08-02.pdf
12. Arlow, J., Neustadt, I.: UML 2. Anaya, Madrid (2005)
13. Vidal, D. E., Vidal, S. C.: Como reforzar Diagramas de Clases UML aplicando OCL y Object-Z: un caso práctico, http://eventos.spc.org.pe/jpc2007/MyReview/FILES/p23.pdf
14. Warmer, J. B., Kepple, A. G.: The Object Constraint Languaje: Getting Your Models Ready for MDA. Pearson Education, USA (2003)