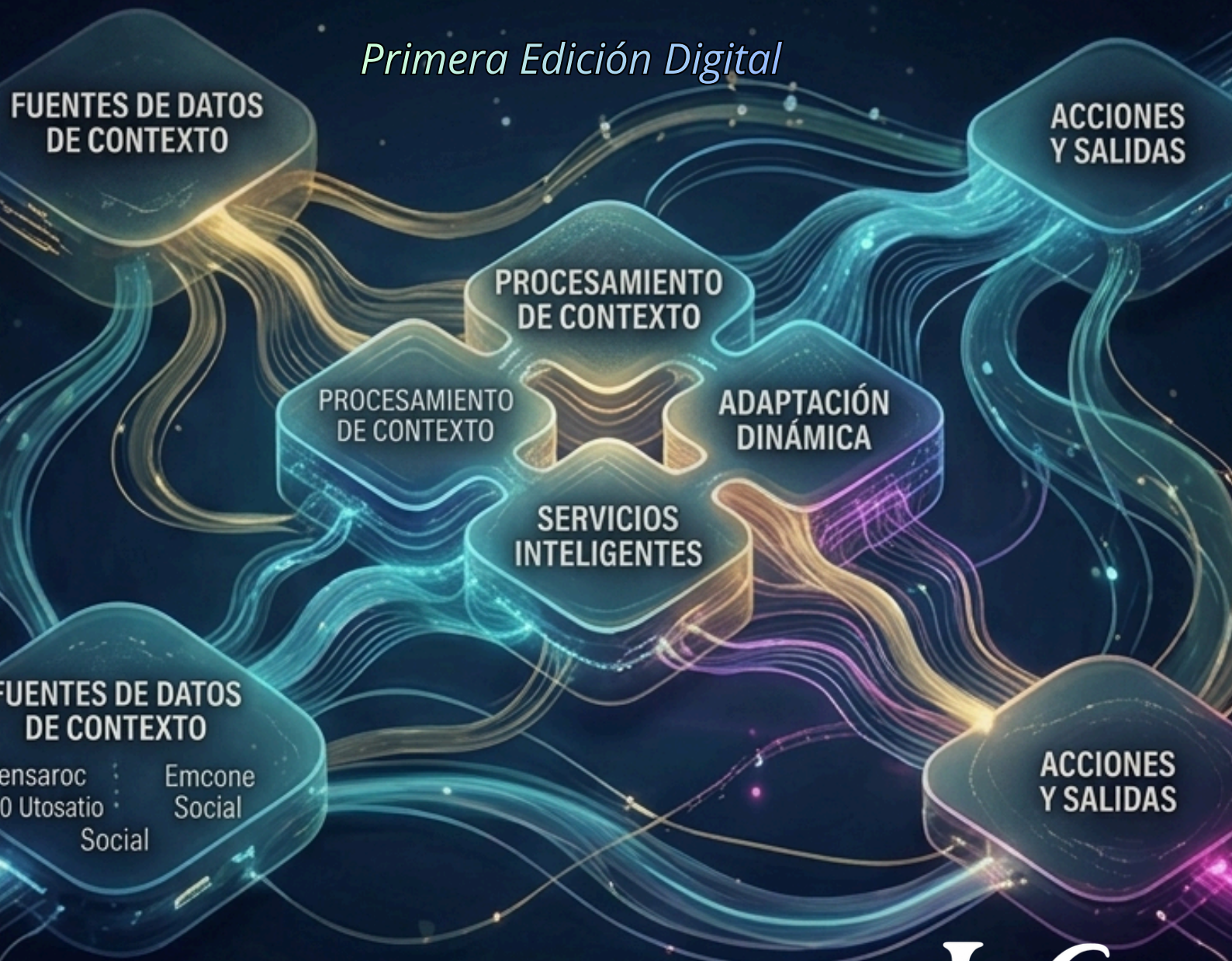
Arquitectura de Software en Sistemas Sensibles al Contexto

Enfoques, Modelado y Propuesta

Primera Edición Digital



Orlando Clemente Iparraguirre Villanueva
Andrés David Epifanía Huerta
William George Paucar Palomino
Paolo Edver Solis Jara

Ho Nexus
EDITORIAL

Arquitectura de Software en Sistemas Sensibles al Contexto

Enfoques, Modelado y Propuesta

Primera Edición Digital

Orlando Clemente Iparraguirre Villanueva
Andrés David Epifanía Huerta
William George Paucar Palomino
Paolo Edver Solis Jara

Ho Nexus
EDITORIAL

Arquitectura de Software en Sistemas Sensibles al Contexto: Enfoques, Modelado y Propuesta

Editor de contenido: Natalia Beltran
Diseño de cubierta: Ho Nexus

1ª edición digital, abril 2026

Editado por:

© HO NEXUS E.I.R.L.
Dirección legal: Urb. Paseo del Mar Mz L4, Lt 33
Nuevo Chimbote, Santa, Ancash - Perú
Correo electrónico; ed.honexus@gmail.com
teléfono: 978 653 152
https://books.honexus.org
DOI: https://doi.org//10.70504/978-612-99401-1-3

Depósito Legal: 2026-04704
ISBN: 978-612-99401-1-3

Revisión por pares:
Este libro (o monografía) fue sometido a evaluación de pares mediante el sistema de doble ciego (doubleblinded review), garantizando la calidad, pertinencia, ética y rigor académico de la obra, conforme a los estándares internacionales de revisión científica y las políticas editoriales de Ho Nexus.

ÍNDICE

PRÓLOGO

¿Alguna vez has deseado que tu teléfono silenciara automáticamente las notificaciones cuando entras al cine, o que la calefacción de tu casa se encendiera justo cuando sales del trabajo? Vivimos rodeados de dispositivos "inteligentes", pero a menudo su inteligencia es más bien una torpeza programada. No saben quiénes somos, dónde estamos o qué hacemos realmente.

Esta obra nace de una investigación (Iparraguirre Villanueva, 2017) Se propuso un reto ambicioso: dejar de pedir a las máquinas que hagan lo que les decimos, para empezar a construir sistemas que entiendan lo que necesitamos. El resultado es un análisis profundo de los enfoques existentes, las técnicas de modelado de contexto y una propuesta arquitectónica original: ECA-DX.

Este manuscrito no es una simple recopilación de teorías. Es una orientación práctica y reflexiva que transforma una compleja investigación académica en un recurso comprensible para estudiantes, desarrolladores, y cualquier entusiasta de la tecnología que quiera entender cómo crear software que realmente se adapte a las personas, y no al revés. Te invito a dejar atrás los modelos rígidos y a adentrarte en el mundo de las reglas ECA, las fuentes de contexto y los gestores de jerarquía. Construyamos juntos un futuro donde la tecnología sea una aliada silenciosa e intuitiva.

RESUMEN

Este texto presenta una exploración completa de la arquitectura de software en sistemas sensibles al contexto, organizada en tres ejes: enfoques (patrones ECA, jerarquías, acciones), modelado (UML, OCL, situaciones) y una propuesta original (la arquitectura ECA-DX).

Nace de la necesidad de superar las arquitecturas tradicionales (cliente-servidor, MVC, SOA) que, aunque potentes, son "ciegas" a las condiciones del mundo real. A través de sus páginas, descubrirás los fundamentos de la computación ubicua, la inteligencia ambiental y, sobre todo, el poderoso modelo Evento-Control-Acción (ECA).

La obra propone, valida y explica la arquitectura ECA-DX (Dominio Extendido) , una solución robusta que integra la extensión "DominioX" (DX) para gestionar fuentes de información híbridas (sensores, GPS, datos de usuario, etc.). Todo ello se demuestra mediante un caso práctico real: el desarrollo de un prototipo de asistencia médica para pacientes con epilepsia, que alerta automáticamente a familiares o médicos cercanos sin intervención humana.

Aprenderás a modelar el contexto, a definir "situaciones" y a utilizar motores de reglas como Jess para dar vida a aplicaciones autónomas, escalables y respetuosas con la privacidad. Es una hoja de ruta para cualquiera que quiera construir el futuro de la tecnología: un futuro que nos entiende.

Palabras clave: Contexto, Arquitectura de Software, Patrón ECA, Computación Ubicua, Internet de las Cosas (IoT), Desarrollo de Aplicaciones, Inteligencia Ambiental.

INTRODUCCIÓN

En la década de los 90, Mark Weiser, un visionario de Xerox PARC, acuñó un término que sonaba a ciencia ficción: la computación ubicua. Su idea era radical: la tecnología más poderosa es la que se vuelve invisible, integrada en el tejido de nuestra vida diaria hasta que la necesitamos, como la electricidad o el agua potable (Weiser, 1991).

Hoy, tres décadas después, vivimos en los albores de esa visión. Llevamos en el bolsillo teléfonos con más potencia que las supercomputadoras de los 80, tenemos altavoces que nos "escuchan" y relojes que monitorizan nuestra salud. Sin embargo, una pieza crucial del rompecabezas sigue sin encajar del todo: la capacidad de estas aplicaciones para entender realmente su contexto.

Imagina que tu móvil no solo supiera tu ubicación (gracias al GPS), sino que también interpretara que estás en una reunión importante (por el ruido ambiente y las palabras clave en tu calendario) y, en consecuencia, desviara las llamadas no urgentes a correo de voz, mostrando solo un mensaje discreto. Eso es ser "sensible al contexto".

El problema es que desarrollar este tipo de software es notoriamente difícil. Las arquitecturas tradicionales con las que la mayoría de los programadores están familiarizados, como Cliente-Servidor o Modelo-Vista-Controlador (MVC) , son excelentes para gestionar bases de datos o peticiones de usuario, pero son "sordas" y "ciegas" al mundo físico. No están diseñadas para procesar el flujo constante de datos de un sensor de

temperatura, una señal de Bluetooth o la actividad frenética del usuario sobre la pantalla (García, 2023).

El gran reto para el desarrollador actual no es solo hacer que la app "funcione", sino hacer que "reaccione" de forma autónoma e inteligente a un entorno cambiante.

Investigaciones recientes en el campo de la Inteligencia Ambiental (AmI) subrayan que la verdadera adopción de sistemas autónomos dependerá de su capacidad para operar con información incompleta y dinámica, aprendiendo del comportamiento del usuario (Bimpas et al., 2024), (Dai et al., 2025). Paralelamente, el auge del Internet de las Cosas (IoT) ha creado un ecosistema de sensores que genera una cantidad ingente de datos contextuales, pero cuyo potencial para la "proactividad" del software aún está por explotar completamente (Vijayvargia et al., 2025), (Mouhim & Lachhab, 2025). Incluso plataformas modernas de Edge Computing proponen acercar el procesamiento a las fuentes de datos para reducir la latencia, un requisito fundamental para las aplicaciones reactivas que exploramos en este libro (Ficili et al., 2025), (Cajas Ordóñez et al., 2025).

Este manuscrito nace con un propósito claro: democratizar el conocimiento necesario para construir estas aplicaciones. No es una investigación cerrada, sino una orientación abierta que parte de una exhaustiva investigación doctoral (Iparraguirre Villanueva, 2017) Y la transforma en un lenguaje claro y práctico.

¿Qué encontrarás en las próximas páginas?

A lo largo de estos capítulos, desglosaremos los enfoques, el modelado y la propuesta que dan título a esta obra.

Capítulo 1: El diagnóstico. Veremos por qué las arquitecturas actuales se quedan cortas y definiremos el problema a resolver.

Capítulo 2: Las herramientas conceptuales. Exploraremos patrones de diseño como el Evento-Control-Acción (ECA) , el Gestor de Jerarquías y el de Acciones, que son los ladrillos de nuestra propuesta.

Capítulo 3: La arquitectura propuesta (ECA-DX). Aquí presentaremos el corazón del libro: nuestra solución para integrar fuentes de información heterogéneas de manera eficiente.

Capítulo 4: Manos a la obra. A través del desarrollo de una aplicación real para monitorizar a pacientes con epilepsia, demostraremos cómo funciona todo en la práctica.

Capítulo 5: Reflexiones y futuro. Analizaremos los resultados, discutiremos los desafíos de rendimiento y escalabilidad, y lanzaremos ideas sobre hacia dónde debe dirigirse la próxima generación de software contextual.

Prepárate para un viaje fascinante. Al final, no solo entenderás la teoría, sino que tendrás una hoja de ruta para construir aplicaciones que, literalmente, nos entienden.

CAPÍTULO 1:

El Desafío de un Software que "Siente" el Mundo

1.1 ¿Qué es una Aplicación Sensible al Contexto?

Imagina que llegas a tu casa después de un largo día de trabajo. Mientras te acercas a la puerta, la luz del recibidor se enciende automáticamente, el termostato ajusta la temperatura a tu grado de confort favorito y tu asistente de voz te informa suavemente: "Tienes un mensaje de tu hija. ¿Quieres escucharlo?". No has tocado un solo botón, no has dicho una orden explícita. El sistema simplemente sabe que has llegado, que es de noche, que prefieres un ambiente cálido y que lo primero que te interesa es saber de tu familia.

Esa, querido lector, es la promesa de las Aplicaciones Sensibles al Contexto. Son programas de software con una habilidad casi mágica: la capacidad de percibir su entorno y adaptar su comportamiento sin que un humano se lo ordene directamente.

En términos más técnicos, pero igual de fascinantes, una aplicación es "sensible al contexto" cuando puede extraer, interpretar y utilizar información del entorno (el contexto) para modificar sus funcionalidades de forma dinámica.

Pero ¿qué es ese tal "contexto"? La definición más aceptada en el mundo académico proviene de (Dey et al., 2001), quienes lo describen como:

> *"Cualquier información que pueda utilizarse para caracterizar la situación de una entidad (una persona, un lugar o un objeto) que se considere relevante para la interacción entre un usuario y una aplicación."*

En cristiano: el contexto es todo aquello que rodea al usuario y que puede ser útil para que la aplicación tome mejores decisiones. Esto incluye, entre muchos otros factores:

Tipo de Contexto	**Ejemplos**
Ubicación física	GPS, sala de la casa, nombre del edificio, piso
Identidad y perfil	Quién eres, tus preferencias musicales, tu edad
Tiempo	Hora del día, fecha, estación del año
Entorno físico	Temperatura, nivel de ruido, intensidad de luz
Estado del dispositivo	Batería restante, conectividad Wifi, memoria disponible
Actividad del usuario	Caminando, conduciendo, durmiendo, en reunión
Relaciones sociales	Cerca de amigos, en familia, con colegas del trabajo

Un ejemplo clásico y ya cotidiano es el GPS de navegación. Cuando abres Waze o Google Maps, la aplicación no solo te muestra un mapa estático. Utiliza tu ubicación actual (contexto espacial), la hora del día (contexto temporal) y los datos de tráfico en tiempo real (contexto del entorno) para calcular la ruta más rápida. Incluso puede sugerirte salir antes si detecta una congestión inminente. Eso es sensibilidad al contexto en acción.

Pero el verdadero potencial va mucho más allá. Un sistema realmente inteligente podría, por ejemplo:

- Un **smartwatch** que detecta un ritmo cardíaco anormal y, sin que tú hagas nada, contacta a tus familiares o a los servicios de emergencia más cercanos, enviando además tu ubicación exacta.
- Una **aplicación de oficina** que, al detectar que has entrado en la sala de reuniones (mediante balizas Bluetooth), silencia automáticamente tu teléfono y activa el "modo no molestar" hasta que salgas.
- Un **asistente de conducción** que, sabiendo que estás a punto de quedarte sin batería en tu coche eléctrico (contexto del vehículo), te sugiere la estación de carga más cercana y además reserva el puesto automáticamente.

Como ves, el hilo conductor es la autonomía: la capacidad del software para actuar por sí mismo, liberando al usuario de la carga de tener que programar, configurar o dar órdenes constantes.

1.2 El Problema: ¿Por qué el Software Tradicional es "Ciego"?

Si la idea es tan maravillosa, cabe preguntarse: ¿por qué no todas las aplicaciones son sensibles al contexto? ¿Por qué la mayoría de las apps que usas a diario son, en el fondo, "torpes" e "insensibles"?

La respuesta no está en la falta de voluntad de los creadores, sino en una limitación de base: las arquitecturas de software con las que hemos

estado trabajando durante décadas no están diseñadas para este propósito.

Pensemos en las arquitecturas más comunes con las que los desarrolladores construyen aplicaciones hoy en día:

1.2.1 Arquitectura Cliente-Servidor

Es el modelo más extendido (Figura 1). Un cliente (tu móvil, tu ordenador) pide algo, y un servidor (un ordenador potente en la nube) responde (Lituma Sarmiento, 2023). Es como pedir un café en una cafetería: tú pides (cliente), el camarero va a la máquina (servidor) y te trae el café.

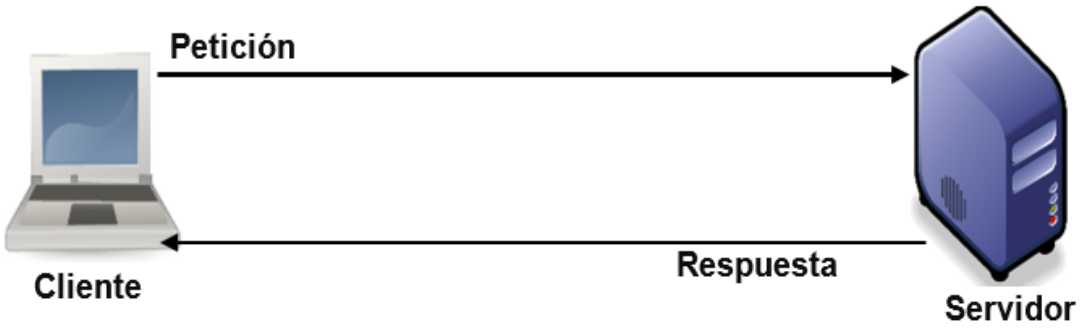


Figura 1: Arquitectura Cliente Servidor *(Iparraguirre Villanueva, 2017)*

El problema: El cliente solo habla cuando tiene algo que pedir. El servidor nunca inicia una conversación. En este modelo, el servidor no puede "sentir" que el cliente se esté quedando sin batería, que haya cambiado de ubicación o que el usuario esté estresado. El sistema es intrínsecamente reactivo a las órdenes del usuario, no proactivo ante los cambios del entorno.

1.2.2 Arquitectura Modelo-Vista-Controlador (MVC)

Es el estándar de facto para aplicaciones con interfaces de usuario (García, 2023). Separa la aplicación en tres partes: el Modelo (los datos), la Vista (lo que ves en pantalla) y el Controlador (el que gestiona las interacciones, como hacer clic en un botón), como se aprecia en la Figura 2).

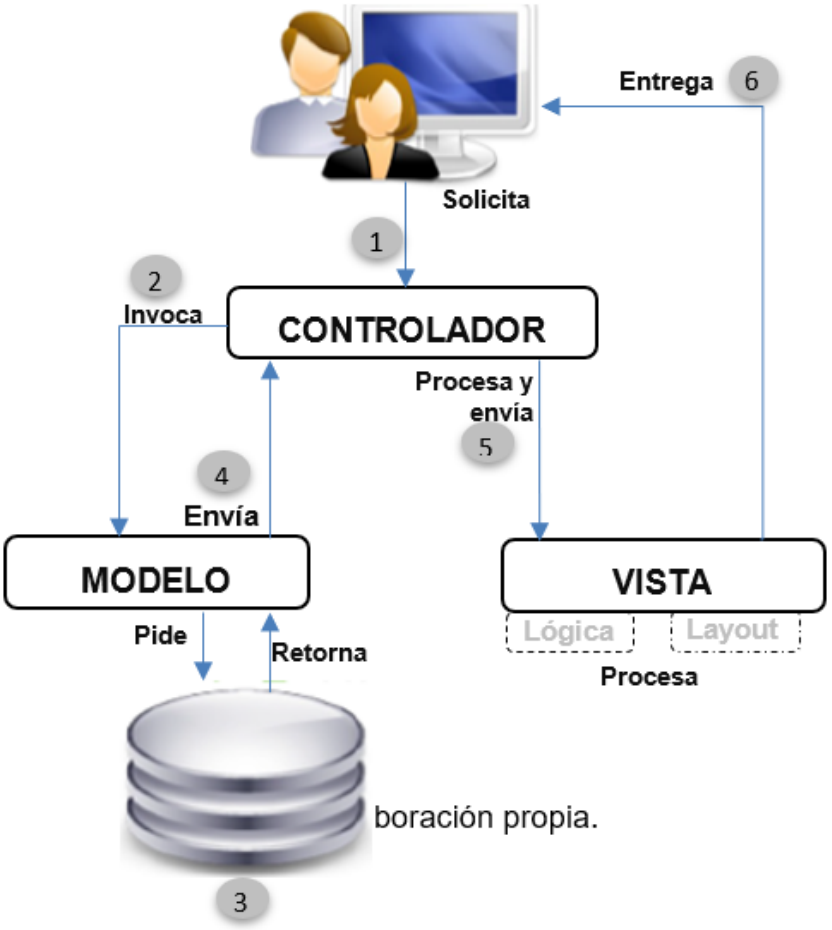


Figura 2: Funcionamiento del patrón modelo-vista-controlador *(Iparraguirre Villanueva, 2017)*

El problema: Todo gira en torno a la interacción explícita del usuario con la interfaz. Si el usuario no pulsa, desliza o escribe, el controlador no hace

nada. No puede "reaccionar" porque, por ejemplo, la temperatura subió de 30 a 40 grados. No hay un "controlador de temperatura" esperando a que algo ocurra. El modelo MVC asume que toda acción comienza con un gesto humano.

1.2.3 Arquitectura Orientada a Servicios (SOA)

Muy popular en entornos empresariales. Como se puede apreciar en la Figura 3, consiste en dividir las funcionalidades en "servicios" independientes que se comunican entre sí (Gereda Hernandez, 2024), (Zohra Trabelsi et al., 2024). Un servicio de "facturación", uno de "clientes", uno de "logística", etc.

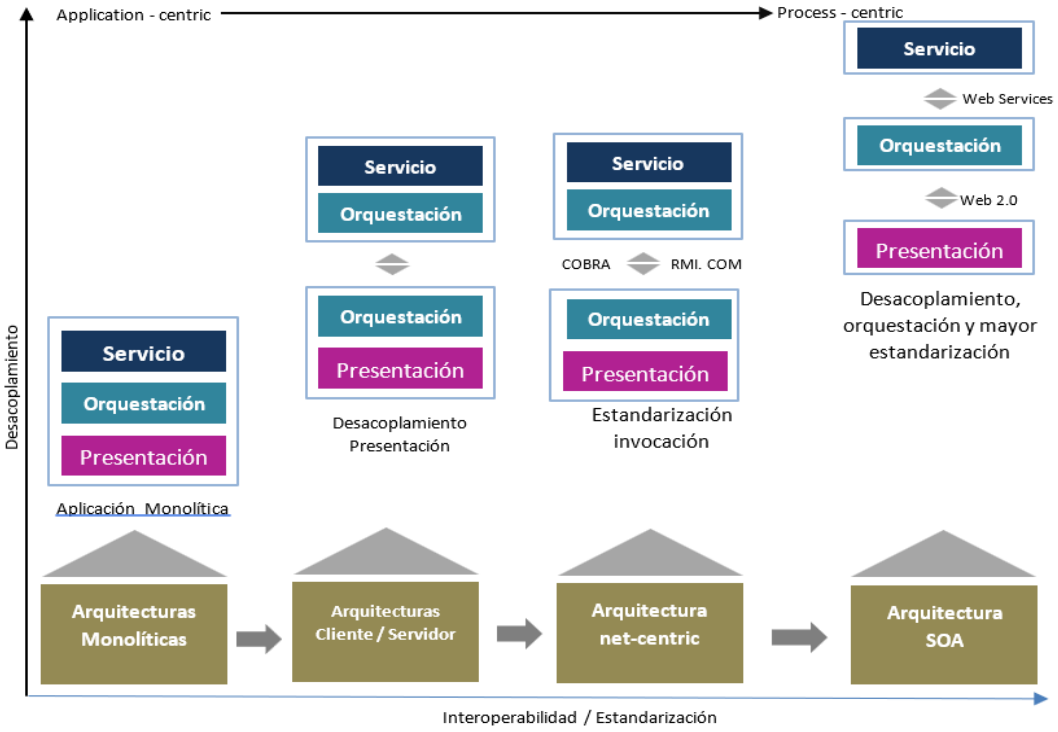


Figura 3: Impacto de SOA en la evaluación de las TI. Fuente: (González Quiroga, 2011)

El problema: Aunque SOA permite una gran flexibilidad para integrar sistemas, sigue siendo intrínsecamente pasiva. Los servicios esperan a ser llamados. No hay, por diseño, un servicio de "monitoreo de contexto" que pueda despertar a otros servicios cuando algo cambia en el mundo real. SOA organiza muy bien los recursos digitales, pero es ajena al entorno físico del usuario.

La conclusión es contundente:

Las arquitecturas tradicionales funcionan como un taxi: esperan a que les indiques el destino y luego actúan. Las aplicaciones sensibles al contexto necesitan funcionar como un mayordomo atento: observan constantemente, anticipan necesidades y actúan antes de que tengas que pedirlo.

"Uno de los puntos débiles de las aplicaciones actuales es la insensibilidad ante los cambios del contexto... Las arquitecturas de software para el desarrollo de aplicaciones basadas en el contexto no existen; en el mejor de los casos, están en proyectos y aún no han madurado."

Y aquí radica el corazón del problema: no es que los desarrolladores sean perezosos o poco visionarios. Es que las herramientas conceptuales que tienen a su disposición (los patrones, los frameworks, los lenguajes) fueron inventadas para un mundo donde la computadora era un "artefacto de escritorio", un "cajón mágico" al que había que acercarse a pedirle cosas. Ese mundo ya no existe.

Hoy, la computadora está en todas partes: en tu muñeca, en tu bolsillo, en tu coche, en tus lentes, en tus electrodomésticos. Necesitamos un nuevo tipo de arquitectura de software, una que esté diseñada desde los cimientos para sentir el mundo.

1.3 El Coste de la Distracción: La Necesidad de Aplicaciones Autónomas

Quizás aún no eres plenamente consciente del problema. Después de todo, tus aplicaciones actuales funcionan, ¿no es así? Enciendes el móvil, abres la app de música, seleccionas una lista de reproducción y le das al "play". Funciona.

Pero pensemos en el coste invisible de esta interacción constante.

El Coste de la Atención

Cada vez que sacas tu teléfono para hacer algo, estás realizando una transacción de atención. Le estás robando unos segundos (y a veces minutos) de concentración a lo que realmente estás haciendo: conducir, conversar con alguien, leer un documento importante, o simplemente descansar. (Weiser, 1993), el padre de la computación ubicua, ya lo advirtió:

"El modelo de interacción entre los dispositivos móviles actuales requiere de una acción humana para ejecutar un evento. Actualmente, este modelo genera una distracción en los usuarios... la interacción con la tecnología debe ser inconsciente."

Situaciones cotidianas que reflejan este coste:

- **Mientras conduces:** necesitas cambiar de canción o ver el mapa. Miras la pantalla. En esos 2 segundos, un peatón puede cruzar inesperadamente. Los accidentes por distracción al volante son una de las principales causas de siniestralidad.
- **En una reunión importante:** tu teléfono vibra. Sacas el móvil para ver si es urgente. Tu gesto rompe el hilo de la conversación. Tu equipo percibe que no estás 100% presente.
- **En casa, de madrugada:** tu hijo tiene fiebre. Necesitas buscar "qué hacer ante una fiebre alta", leer, decidir. Tu atención está dividida entre la pantalla y el pequeño. Y estás perdiendo minutos preciosos.

El coste de la distracción es real, medible y, en muchos casos, peligroso.

La Curva de Aprendizaje

Cada nueva aplicación que instalas tiene su propia lógica, sus menús, sus gestos. Esa "curva de aprendizaje" es otro coste. Los seres humanos no deberíamos tener que "aprender a usar" objetos inteligentes. Un objeto realmente inteligente debería adaptarse a nosotros, no nosotros a él. Weiser lo expresó de forma brillante:

"La tecnología más profunda es la que desaparece. Se teje en el tejido de la vida cotidiana hasta que es indistinguible de ella."
(Iparraguirre Villanueva, 2017)

Una puerta no necesita un folleto de instrucciones. La aprendes a usar en segundos. Una aplicación que te pide que estudies su interfaz está fallando en ese ideal.

El Sueño de la Proactividad Silenciosa

Las aplicaciones sensibles al contexto aspiran a un mundo donde la tecnología actúe en segundo plano, como un asistente silencioso:

- Tu móvil sabe que estás manejando (por el Bluetooth del coche y el acelerómetro) y activa automáticamente el "modo coche": responde los mensajes con un "Estoy conduciendo, te contacto luego".
- La pulsera de actividad detecta que llevas 30 minutos con un ritmo cardíaco elevado estando en reposo, y te sugiere hacer ejercicios de respiración.
- Tu agenda inteligente nota que has cancelado dos reuniones seguidas por enfermedad, y reprograma automáticamente tus tareas menos prioritarias para la próxima semana.

No se trata de quitarle control al usuario. Se trata de devolverle su atención. La tecnología debería ocuparse de lo rutinario y lo obvio, para que el ser humano pueda dedicarse a lo creativo, a lo emocional, a lo que realmente importa.

Este manuscrito se propone, precisamente, ofrecer una hoja de ruta para construir ese tipo de software atento y autónomo.

1.4 Estado del Arte: ¿Qué se ha intentado hasta ahora? (Un breve recorrido por Middlewares y Frameworks)

No partimos de cero. La comunidad científica y tecnológica lleva años intentando resolver este rompecabezas. Se han propuesto múltiples frameworks (esqueletos de aplicación reutilizables) y middlewares (capas de software que facilitan la comunicación entre partes) para ayudar a los desarrolladores a crear aplicaciones sensibles al contexto.

A continuación, presentamos un recorrido por los más representativos, destacando sus logros y sus limitaciones.

1.4.1 ContextPhone

Desarrollado por investigadores de la Universidad de Helsinki (Raento et al., 2005), fue uno de los primeros en apostar por los teléfonos móviles como plataforma contextual. Ofrecía módulos para acceder a sensores (GPS, batería, registros de llamadas) y para comunicarse (SMS, Bluetooth).

Aciertos: Demostró que un teléfono podía ser una fuente rica de información contextual y que se podían construir aplicaciones que reaccionaran a cambios en el entorno (por ejemplo, cambiar el perfil de sonido según la ubicación).

Limitaciones: Estaba atado a la plataforma Symbian, un sistema operativo para móviles que hoy es historia. Cuando Android e iOS arrasaron el mercado, ContextPhone quedó obsoleto. Es un claro ejemplo

de un problema recurrente: la alta dependencia de hardware y sistemas operativos concretos.

1.4.2 JCAF (Java Context Awareness Framework)

Propuesto por (Bardram, 2005), JCAF es un framework escrito en Java, uno de los lenguajes más universales. Está diseñado como una infraestructura de servicios distribuida y basada en eventos.

Aciertos: Su gran aporte fue la separación entre la adquisición de contexto (los sensores) y la lógica de la aplicación. Es extensible y seguro, y al estar basado en Java, corre en múltiples plataformas.

Limitaciones: Aunque es potente, la curva de aprendizaje es alta. Requiere que el desarrollador entienda conceptos como "servicios de contexto", "monitores" y "actuadores". Además, no ofrece un lenguaje de alto nivel para definir comportamientos reactivos complejos; el programador debe escribir mucho código Java para cada regla.

1.4.3 SOCAM (Service-Oriented Context-Aware Middleware)

Presentado por (Chen et al., 2003) para entornos de "coche inteligente". SOCAM utiliza ontologías (representaciones formales del conocimiento) para describir el contexto de manera semántica.

Aciertos: Su uso de ontologías es su principal fortaleza. Permite que diferentes aplicaciones compartan y entiendan el contexto sin ambigüedades. Incorpora un razonador de contexto basado en lógica, lo

que le permite inferir nueva información (por ejemplo, si estás en la oficina y son las 9:00, infiere que "estás trabajando").

Limitaciones: Es una arquitectura centralizada (cliente-servidor). Si el servidor central falla, todo el sistema colapsa. Eso contradice la naturaleza distribuida de la computación ubicua. Además, el razonador puede volverse un cuello de botella si hay muchos usuarios y sensores.

1.4.4 CASS (Context Awareness Sub-Structure)

Propuesta por (Fahy & Clarke, 2004). CASS es un middleware basado en servidor, diseñado para dispositivos con recursos limitados (como las primeras PDAs).

Aciertos: Su mayor contribución es llevar el procesamiento pesado al servidor, liberando al dispositivo móvil de tareas costosas. El dispositivo solo envía datos crudos de sensores y recibe ya la información contextual procesada.

Limitaciones: Al igual que SOCAM, es centralizado y depende de la conectividad permanente con el servidor. Si la conexión a internet falla, la aplicación móvil se queda "ciega". En la época de su creación (el año 2000), las conexiones móviles no eran ni de lejos tan estables como hoy.

1.4.5 Hydrogen

(Hofer et al., 2003) propusieron Hydrogen, una arquitectura de tres capas para dispositivos móviles (Adaptación, Gestión y Aplicación) que utiliza un modelo de comunicación peer-to-peer (entre iguales).

Aciertos: Al ser distribuida (peer-to-peer), es más robusta que las soluciones centralizadas. Si un dispositivo falla, los demás pueden seguir operando. Está pensada para los limitados recursos de los móviles de principios de los 2000.

Limitaciones: La capa de "Adaptación", que se encarga de conectar con los sensores, está muy acoplada al hardware. Cambiar un sensor implica tener que reescribir parte de la capa, lo que dificulta la reutilización y el mantenimiento.

Tabla Resumen: Frameworks para aplicaciones sensibles al contexto.

Framework	Enfoque	Fortaleza Principal	Debilidad Principal
ContextPhone	Móvil (Symbian)	Fue pionero en usar el móvil como sensor	Obsoleto, atado a una plataforma concreta
JCAF	Java / Distribuido	Extensible, seguro, separa adquisición de lógica	Curva de aprendizaje alta, mucho código escrito a mano

SOCAM	Ontologías / Centralizado	Poderoso razonador semántico del contexto	Punto único de fallo (servidor central)
CASS	Cliente-Servidor	Libera recursos del dispositivo móvil	Dependencia total de la conectividad al servidor
Hydrogen	Peer-to-peer / 3 capas	Robusto, diseñado para recursos limitados	Capa de adaptación muy ligada al hardware

¿Qué nos enseñan estos intentos?

La lección principal es que no existe una solución perfecta y universal. Cada propuesta sobresale en algún aspecto (semántica, distribución, eficiencia) pero cojea en otros (dependencia de hardware, complejidad de uso, falta de estándares).

Este diagnóstico nos llevó a plantearnos una pregunta clave: ¿Podemos diseñar una arquitectura que combine lo mejor de cada una? Que sea distribuida como Hydrogen, con un razonador potente como SOCAM, pero independiente del hardware como JCAF, y que ofrezca un lenguaje de alto nivel para definir el comportamiento reactivo.

Esa es exactamente la propuesta de este libro, y la desarrollaremos en detalle en el Capítulo 3.

1.5 El Camino a Seguir: Objetivos de esta Obra

Llegados a este punto, el diagnóstico está claro: las arquitecturas tradicionales son insuficientes para el desarrollo ágil y eficaz de aplicaciones sensibles al contexto. Los frameworks y middlewares existentes, aunque valiosos, presentan limitaciones importantes en términos de dependencia tecnológica, centralización o complejidad de uso.

Este texto, nacido de una rigurosa investigación (Iparraguirre Villanueva, 2017), se propone ofrecer una solución integrada y práctica. No es solo teoría: presentaremos una arquitectura concreta, la llamaremos ECA-DX (Evento-Control-Acción con Dominio Extendido), y la pondremos a prueba con un caso real.

Nuestro Objetivo General

Diseñar y presentar una arquitectura de software, clara y aplicable, que permita el desarrollo estándar de aplicaciones sensibles al contexto, facilitando el trabajo de los desarrolladores y abriendo nuevas posibilidades a los usuarios finales.

Objetivos Específicos que orientan este libro

A lo largo de los siguientes capítulos, perseguiremos metas concretas:

- **Analizar el estado del arte de forma práctica.** No solo citaremos teorías, sino que extraeremos las lecciones aprendidas de arquitecturas, middlewares y patrones existentes (como ECA, el patrón de jerarquías y el patrón de acciones) para entender qué funciona y qué no.
- **Presentar y justificar la extensión "DominioX" (DX).** Mostraremos cómo añadimos una capa a la arquitectura ECA clásica que permite integrar de forma limpia y eficiente fuentes de información híbridas: sensores físicos, datos de APIs web, información del propio usuario, etc.
- **Demostrar con un ejemplo real (y funcional).** No te dejaremos con las ganas. Construiremos paso a paso un prototipo de aplicación de asistencia médica para pacientes con epilepsia. Verás cómo modelamos el contexto, definimos las "situaciones" de riesgo, implementamos las reglas y, finalmente, evaluamos su rendimiento y escalabilidad.

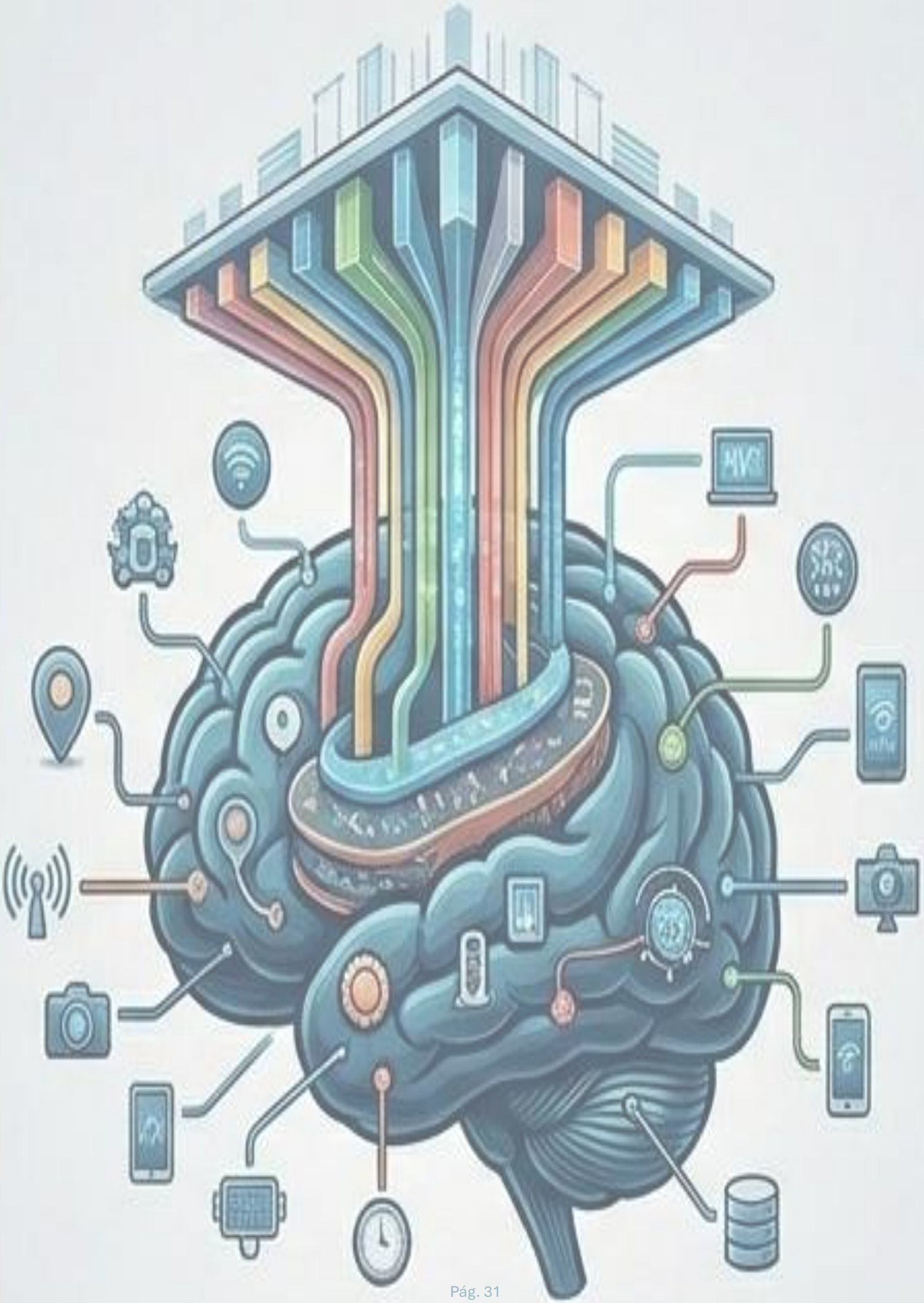
Para conseguir estos objetivos, el libro sigue una estructura clara:

- **Capítulo 2:** Introduciremos los conceptos y patrones fundamentales que forman la base de nuestra propuesta (ECA, jerarquías de fuentes, gestión de acciones).
- **Capítulo 3:** Desglosaremos la arquitectura ECA-DX, explicando cada uno de sus componentes y cómo se articulan.
- **Capítulo 4:** Manos a la obra con el caso práctico de salud, donde verás la teoría convertida en código y funcionando.

- **Capítulo 5:** Reflexionaremos sobre los resultados, las lecciones aprendidas y los desafíos que aún quedan por delante.

Estás a punto de emprender un viaje fascinante al corazón del software inteligente. Pasa la página. El futuro de la tecnología, una que nos entiende sin que tengamos que pedírselo, comienza aquí.

En el siguiente capítulo, exploraremos los fundamentos teóricos que necesitamos para construir nuestra arquitectura. Descubriremos los patrones de diseño que nos permitirán manejar eventos, gestionar jerarquías de información contextual y ejecutar acciones de forma eficiente. Te espero en el Capítulo 2.



CAPÍTULO 2: Fundamentos para Construir Conciencia Contextual

2.1 De la Computación Ubicua a la Inteligencia Ambiental (cambio de paradigma)

Para entender hacia dónde se dirige el software sensible al contexto, primero debemos comprender de dónde viene la idea. Y esa historia comienza con un hombre, una visión y una frase que se convertiría en profética.

El Visionario Mark Weiser

A principios de la década de 1990, mientras la mayoría de la humanidad todavía veía los ordenadores como máquinas grandes, ruidosas y complicadas que habitaban en cuartos fríos y exclusivos, un científico de Xerox PARC llamado Mark Weiser planteó una idea revolucionaria.

Su propuesta, publicada en el artículo "The Computer for the 21st Century" (Weiser, 1991), llevaba por título una declaración de intenciones. La computadora del siglo XXI, según Weiser, no sería una caja imponente sobre un escritorio. Sería algo mucho más sutil y profundo:

"La tecnología más profunda es la que desaparece. Se teje en el tejido de la vida cotidiana hasta que es indistinguible de ella."

Así nació el concepto de Computación Ubicua (Ubiquitous Computing), también conocida como "UbiComp". La idea central era sencilla pero poderosa: en lugar de que las personas se acerquen a una única computadora, las computadoras se dispersarán por todo el entorno,

integradas en objetos cotidianos, y trabajarán en segundo plano para servir a las personas (Weiser, 1993).

Los Tres Estados de la Computación, según Weiser

Weiser describió la evolución de la computación en tres fases o "estados":

Estado	Característica	Ejemplo
Mainframe	Muchas personas comparten una sola computadora	Los enormes ordenadores centrales de los años 60-70
PC (Personal Computer)	Una persona, una computadora	El ordenador de escritorio que popularizaron Apple e IBM en los 80-90
Computación Ubicua	Muchas computadoras por persona, distribuidas y transparentes	El mundo que empezamos a vivir hoy: móviles, wearables, sensores en el hogar

No se trata de tener más potencia de cómputo en un solo dispositivo, sino de repartir esa potencia en muchos dispositivos pequeños, especializados y conectados.

El Nacimiento de la Inteligencia Ambiental (AmI)

Si la Computación Ubicua puso el foco en la distribución y la transparencia, la Inteligencia Ambiental (Ambient Intelligence, AmI) añadió un ingrediente crucial: la inteligencia y la capacidad de respuesta.

El término AmI, popularizado a finales de los 90 por el consejo asesor de tecnologías de la información de la Unión Europea, añade una dimensión más humana al concepto de Weiser. Un entorno con Inteligencia Ambiental no solo está lleno de dispositivos conectados; es un entorno que:

1) Reconoce la presencia de las personas.
2) Comprende sus necesidades y preferencias (a veces implícitas).
3) Actúa en consecuencia de forma autónoma y proactiva.

Imagina una habitación con Inteligencia Ambiental. Al entrar, no solo se enciende la luz (ubicuidad). La luz se ajusta a la intensidad que prefieres habitualmente por las tardes (inteligencia). Si detecta que estás leyendo un libro (quizás porque el sensor de movimiento te ve quieto y enfocando hacia una superficie), aumenta la luz sobre el asiento (capacidad de respuesta). Si suena el teléfono, baja la música automáticamente sin que nadie se lo pida (proactividad).

¿Por qué es importante este cambio de paradigma?

Porque redefine quién tiene la iniciativa en la interacción humano-computadora.

- En el paradigma tradicional (PC), la iniciativa la tiene siempre el usuario: "Yo ordeno, la máquina obedece."
- En el paradigma de la Computación Ubicua/Inteligencia Ambiental, la iniciativa puede ser compartida. El entorno puede anticiparse: "El

usuario está haciendo X. Según su patrón de comportamiento, probablemente necesitará Y. Actúo en consecuencia."

Este cambio de paradigma es el pilar sobre el que se asientan las aplicaciones sensibles al contexto. Ya no somos solo "operadores" de máquinas. Somos "habitantes" de entornos inteligentes que nos sirven de forma silenciosa y atenta.

2.2 El Contexto: Más Allá de la Simple Localización

En el imaginario popular, cuando se habla de aplicaciones que "saben cosas de ti", lo primero que viene a la mente es la ubicación. Y es cierto: la geolocalización es una de las fuentes de contexto más poderosas y utilizadas. Pero sería un error reducirlo solo a eso.

Definiendo el Contexto (de una vez por todas)

La definición más aceptada en el mundo académico proviene, como ya adelantamos, de (Dey et al., 2001):

"El contexto es cualquier información que pueda utilizarse para caracterizar la situación de una entidad (una persona, un lugar o un objeto) que se considere relevante para la interacción entre un usuario y una aplicación."

Desglosemos esta definición pieza por pieza:

- **"Cualquier información":** No hay límite. Puede ser un dato cuantitativo (22 grados centígrados) o cualitativo ("nublado"), algo medido por un sensor (latitud, longitud) o algo introducido por el usuario ("estoy en una reunión").
- **"Situación de una entidad":** El contexto no existe en el vacío. Siempre es el contexto de algo (una persona, un edificio, un sensor, un coche).
- **"Relevante para la interacción":** Si el dato no va a influir en el comportamiento de la aplicación, no es contexto relevante. Es solo un dato más.

Tipos de Información Contextual

A lo largo de los años, los investigadores han propuesto múltiples clasificaciones. Una de las más útiles distingue tres grandes categorías:

1. Contexto del Usuario

Todo lo relacionado con la persona que utiliza la aplicación.

Subcategoría	Ejemplos
Perfil e identidad	Nombre, edad, idioma preferido, suscripciones activas
Localización	Dónde está ahora, por dónde ha pasado, hacia dónde se dirige
Estado emocional o físico	Frecuencia cardíaca, nivel de estrés (estimado), estado de ánimo

Actividad actual	Caminando, durmiendo, conduciendo, trabajando, comiendo
Relaciones sociales	Con quién está (familia, amigos, compañeros), si está solo o acompañado

2. Contexto Físico

Las condiciones del entorno material que rodea al usuario y al dispositivo.

Subcategoría	Ejemplos
Condiciones climáticas	Temperatura, humedad, presión atmosférica, luminosidad
Condiciones acústicas	Nivel de ruido ambiente, tipo de sonido predominante (música, voz, tráfico)
Infraestructura	Redes Wifi disponibles, balizas Bluetooth, torres de telefonía cercanas

3. Contexto Computacional

El estado del propio sistema y los recursos tecnológicos a su alcance.

Subcategoría	Ejemplos
Estado del dispositivo	Batería restante, memoria disponible, capacidad de almacenamiento
Conectividad	Tipo de red (Wifi, 5G, sin conexión), calidad de la señal
Recursos cercanos	Impresoras, pantallas, altavoces, otros dispositivos disponibles

El Contexto es Dinámico y Jerárquico

Una característica fundamental del contexto es que cambia constantemente. Tu ubicación, tu nivel de batería, el ruido a tu alrededor... todo varía en cuestión de segundos. Por eso las aplicaciones sensibles al contexto deben estar siempre "escuchando" y "observando".

Además, el contexto puede entenderse en niveles de abstracción. No es lo mismo un dato bruto de un sensor (p.ej., "latitud: 40.4168, longitud: -3.7038") que un hecho contextual de alto nivel ("María está en la Puerta del Sol"). A menudo, la aplicación no necesita saber las coordenadas exactas; necesita saber la interpretación semántica de esos datos.

¿Por qué el contexto es tan importante para el software?

Porque es la materia prima de la adaptación. Sin contexto, la aplicación es genérica y rígida. Con contexto, la aplicación puede:

- **Personalizar** su comportamiento: mostrar la información relevante para el momento y lugar.
- **Automatizar** tareas: realizar acciones que el usuario haría manualmente, pero sin que él tenga que intervenir.
- **Anticiparse** a necesidades: sugerir una ruta alternativa antes de que el usuario llegue al atasco, o recordar una tarea antes de que se olvide.
- **Mejorar la seguridad**: detectar situaciones anómalas (un ritmo cardíaco irregular, un movimiento brusco del coche) y reaccionar de inmediato.

Como veremos más adelante, capturar, interpretar y explotar el contexto es la esencia de los patrones arquitectónicos que vamos a aprender.

2.3 Arquitecturas Existentes y sus Limitaciones (Cliente-Servidor, MVC, SOA)

Ya hemos hablado someramente de este tema en el Capítulo 1. Pero merece la pena profundizar un poco más desde una perspectiva histórica y conceptual, para entender por qué tenemos que buscar nuevas soluciones.

2.3.1 Arquitectura Cliente-Servidor

Es la más antigua y la más extendida en aplicaciones empresariales y de red (Lituma Sarmiento, 2023).

¿Cómo funciona?

Hay dos tipos de entidades lógicas:

- **Cliente**: el que inicia la comunicación, solicita un servicio, espera la respuesta y la muestra/usa. Es "activo".
- **Servidor**: el que recibe la solicitud, la procesa (consulta una base de datos, ejecuta un cálculo) y devuelve la respuesta. Es "pasivo".

Ejemplo cotidiano:

Cuando entras a Instagram, tu móvil (cliente) le pide al servidor de Instagram: "dame las últimas 20 fotos de los usuarios que sigo". El servidor las busca y te las envía.

Limitación principal para el contexto:

El servidor solo actúa cuando el cliente le pide algo. Es inherentemente reactivo a las peticiones. Un servidor no puede "decidir por sí mismo" enviarte una alerta porque detectó (a través de algún sensor) que te estás acercando a un lugar peligroso. Para eso, el cliente tendría que estar preguntando constantemente ("sondeando", en el argot técnico), lo que consume batería y ancho de banda. El modelo Cliente-Servidor no está diseñado para la proactividad.

2.3.2 Arquitectura Modelo-Vista-Controlador (MVC)

Es el estándar para aplicaciones con interfaz gráfica de usuario (GUI), desde webs hasta apps de escritorio (García, 2023).

¿Cómo funciona?

Divide la aplicación en tres capas interconectadas:

- **Modelo**: la lógica de negocio y los datos. Sabe cómo consultar la base de datos, cómo hacer cálculos, etc.

- **Vista**: lo que el usuario ve (pantallas, botones, listas). Es la "cara visible" de la aplicación.
- **Controlador**: el intermediario. Escucha las acciones del usuario (clics, tecleos, gestos) y le dice al Modelo qué hacer y a la Vista qué mostrar.

Ejemplo cotidiano:

En una tienda online, el Modelo sabe cómo buscar productos, la Vista muestra el catálogo, y el Controlador interpreta que has pulsado en "Añadir al carrito" y le dice al Modelo que actualice el carrito y a la Vista que muestre el nuevo total.

Limitación principal para el contexto:

Todo el flujo de trabajo se inicia con una acción explícita del usuario sobre la Vista. Sin esa acción, el Controlador no hace nada. El MVC no tiene un mecanismo natural para manejar "eventos del entorno". Si la temperatura de la habitación sube 5 grados, no hay un "controlador de temperatura" que reciba ese evento y actúe en consecuencia. Para introducir sensibilidad al contexto en MVC, los programadores tienen que añadir código "artificial" (temporizadores, hilos en segundo plano) que son difíciles de mantener y escalar.

2.3.3 Arquitectura Orientada a Servicios (SOA)

Muy popular en entornos empresariales y sistemas distribuidos de gran escala (Zohra Trabelsi et al., 2024).

¿Cómo funciona?

Una aplicación se compone de múltiples "servicios" independientes que se comunican entre sí a través de una red, normalmente utilizando protocolos estándar (como HTTP, SOAP, REST). Cada servicio ofrece una funcionalidad concreta (por ejemplo: "Servicio de Autenticación", "Servicio de Facturación", "Servicio de Envío de Correos").

Ejemplo cotidiano:

Una aerolínea puede tener un servicio de "búsqueda de vuelos", otro de "reserva de asientos", otro de "pago con tarjeta". Cuando reservas un vuelo, tu aplicación llama secuencialmente a esos servicios.

Limitación principal para el contexto:

Aunque SOA permite una excelente modularidad y reutilización, los servicios siguen siendo pasivos. Esperan a que otro servicio (o un orquestador) los invoque. No hay una forma natural de que un servicio "se entere" de un cambio en el contexto y actúe por su cuenta, a menos que otro componente esté constantemente vigilando e invocándolo. Al igual

que en Cliente-Servidor, la proactividad no es un ciudadano de primera clase en SOA.

Tabla Comparativa de Limitaciones

Arquitectura	Rol del Usuario	¿Maneja Eventos del Entorno?	Capacidad Proactiva
Cliente-Servidor	Inicia peticiones	No	Nula (requiere sondeo)
Modelo-Vista-Controlador (MVC)	Realiza acciones explícitas sobre la interfaz	No	Nula (requiere código adicional)
Orientada a Servicios (SOA)	Indirecto (a través de servicios)	No (necesita orquestador externo)	Muy limitada

¿Notas un patrón? Todas estas arquitecturas asumen que el mundo es estático y predecible, y que son los humanos quienes deben empujar la interacción. Pero el mundo real es dinámico, impredecible y está lleno de eventos que no se originan en un teclado o una pantalla táctil.

Necesitamos una arquitectura que ponga los eventos del mundo en el centro, no las acciones del usuario. Ese es el punto de partida de los patrones que veremos a continuación.

2.4 Patrón 1: Evento-Control-Acción (ECA) - La Base de la Reactividad

Llegamos al primer gran patrón arquitectónico para construir aplicaciones sensibles al contexto, y quizás el más importante de todos.

Se llama Evento-Control-Acción, o por sus siglas en inglés, ECA (Event-Condition-Action) (Maatjes, 2008).

¿De dónde viene el patrón ECA?

Originalmente, el patrón ECA nació en el mundo de las bases de datos activas. Una base de datos tradicional solo hace algo cuando le preguntas (SELECT, INSERT, UPDATE). Una base de datos activa podía ejecutar reglas del tipo "Si ocurre un INSERT en la tabla de ventas Y el producto es 'paraguas' Y estamos en el mes de julio, entonces envía una alerta al gerente". ¿Te suena familiar? Es exactamente la misma lógica.

En los años 90 y 2000, investigadores en computación ubicua y sistemas sensibles al contexto tomaron prestado este patrón y lo adaptaron a su dominio. Y funcionó extraordinariamente bien.

La Estructura del Patrón ECA

El patrón ECA se expresa mediante reglas de la forma:

SI ocurre [Evento] Y se cumple [Condición] ENTONCES ejecuta [Acción]

O, en su formulación canónica:

ON [Event] IF [Condition] DO [Action]

Esta sencilla estructura de tres partes encapsula perfectamente el comportamiento que queremos de una aplicación sensible al contexto: observar el mundo, evaluar si es relevante, y actuar en consecuencia.

Desglosemos cada componente:

1. El Evento (Event)

Es la señal que indica que algo ha cambiado en el entorno. Los eventos representan "transiciones" o "sucesos interesantes". Pueden ser:

- **Eventos primitivos:** Provienen directamente de un sensor o de una interacción básica. Ejemplos: "la temperatura superó los 30 grados", "el usuario hizo clic en el botón SOS", "se detectó un movimiento brusco en el acelerómetro".
- **Eventos compuestos:** Son combinaciones lógicas de eventos más simples. Ejemplos: "temperatura > 30 Y humedad > 80%", "llegó un mensaje de texto Y el teléfono estaba en modo silencio".

El patrón distingue claramente entre el estado (que puede durar en el tiempo) y el evento (que es un cambio de estado). Por ejemplo, "temperatura = 32 grados" puede ser un estado; el evento sería el cambio de "estaba a 29 y ahora está a 32".

2. La Condición (Condition)

Es un predicado (algo que se puede evaluar como verdadero o falso) que debe cumplirse además del evento para que se dispare la acción. A

veces la condición puede estar implícita en el evento; otras veces es necesaria para afinar la lógica.

- **Ejemplo con condición:** "Si ocurre un evento de 'pérdida de conexión Wifi' Y el nivel de batería es inferior al 15%, entonces activa el modo ahorro de energía".
- **Función de la condición:** Permite que la regla no se dispare siempre que ocurre el evento, sino solo cuando el contexto adicional así lo requiere. Hace que la regla sea más fina y menos intrusiva.

3. La Acción (Action)

Es la respuesta del sistema. Lo que hace la aplicación cuando se detecta el evento y se cumple la condición. Las acciones pueden ser de muy diversa índole:

- **Mostrar información:** "mostrar una alerta en pantalla", "enviar una notificación al centro de control".
- **Modificar el comportamiento:** "cambiar al perfil silencioso", "aumentar el brillo de la pantalla".
- **Invocar servicios externos:** "enviar un SMS", "llamar a una API de emergencias", "activar una alarma física".

¿Cómo se organiza una aplicación basada en ECA?

El patrón ECA no solo define reglas; también propone una estructura de componentes muy clara:

1) Procesador de Contexto (Context Processor): Es el componente encargado de generar y observar los eventos. Se conecta a los sensores, procesa los datos brutos y determina cuándo se ha producido un cambio de estado relevante (el evento).
2) Controlador (Controller): Es el cerebro. Mantiene un conjunto de reglas ECA (la lógica de la aplicación). Recibe los eventos del Procesador de Contexto, evalúa las condiciones (a menudo también consultando al Procesador de Contexto por el estado actual) y, si procede, decide qué acción ejecutar.
3) Ejecutor de Acciones (Action Performer): Es el brazo ejecutor. Recibe la orden del Controlador (por ejemplo, "ejecuta la acción de enviar un SMS al número X") y se encarga de llevarla a cabo, ya sea internamente o invocando servicios externos.

Un ejemplo concreto para entenderlo mejor

Imaginemos una regla ECA para una aplicación de seguridad en el hogar:

- **Evento**: "Se ha detectado movimiento en la sala de estar a las 3:00 AM".
- **Condición (implícita en este caso):** "El modo 'Armado' está activado".
- **Acción**: "Enviar notificación push a la app del propietario y activar la sirena local".

Flujo:

1) El Procesador de Contexto (conectado al sensor de movimiento PIR y al reloj) detecta el movimiento y genera el evento MOVEMENT_DETECTED a las 3:00 AM.
2) El Controlador recibe el evento. Consulta el estado de la variable "ModoSistema". Está en "Armado". Evalúa la condición como verdadera. La regla se dispara.
3) El Controlador invoca al Ejecutor de Acciones con la orden "EnviaNotificacionAlarma".
4) El Ejecutor de Acciones llama a los servicios de push notification y se conecta con el módulo de la sirena.

¿Por qué el patrón ECA es tan poderoso para el contexto?

Porque coloca los eventos del mundo real en el centro de la arquitectura. Ya no partimos de una petición del usuario o de una interacción con la interfaz. Partimos de algo que ocurre en el entorno. El usuario puede estar durmiendo, y la aplicación se activará sola si algo anómalo sucede. Esa es la base de la proactividad.

2.5 Patrón 2: Fuentes de Contexto y Gestores de Jerarquía - Organizando la Información

El patrón ECA resuelve el problema de "cómo reaccionar", pero deja abierta una cuestión fundamental: ¿cómo obtenemos el contexto y, sobre

todo, cómo lo organizamos cuando proviene de muchas fuentes diferentes y con distintos niveles de abstracción?

Imagina una aplicación domótica avanzada. Tienes decenas de sensores: temperatura, humedad, luz, presencia, consumo eléctrico, estado de puertas y ventanas, etc. El evento "SE_ABRE_LA_PUERTA" puede ser interesante por sí mismo, pero combinarlo con otros eventos y estados es mucho más poderoso: "SE_ABRE_LA_PUERTA Y ESTÁS_FUERA DE_CASA Y ES_DE_NOCHE" podría activar una alerta de seguridad.

¿Cómo construimos esa cadena de razonamiento? Ahí entra el Patrón de Fuentes de Contexto y Gestores de Jerarquía.

La Necesidad de Abstracción

Los datos que vienen directamente de los sensores suelen ser de bajo nivel y muy ruidosos:

- Un GPS te da latitud y longitud (datos numéricos).
- Un acelerómetro te da valores de aceleración en tres ejes.
- Un micrófono te da una onda de sonido (muestras de presión acústica).

Ninguna aplicación quiere lidiar con esos datos en bruto. Lo que quiere saber es:

- "El usuario está en su oficina" (en lugar de lat: 40.438, lon: -3.673).

- "El usuario está corriendo" (en lugar de un patrón complejo de aceleraciones).
- "Hay una conversación en la habitación" (en lugar de una waveform).

Para pasar de los datos brutos a los hechos semánticos, necesitamos una cadena de procesamiento. Un componente procesa los datos del sensor, los interpreta, y genera nueva información de más alto nivel, que a su vez puede ser usada por otro componente.

La Solución: Jerarquía de Componentes

El patrón define dos tipos básicos de componentes de procesamiento de información contextual:

1. Fuente de Contexto (Context Source)

Es el componente que encapsula un sensor concreto o una fuente de datos elemental. Su responsabilidad es obtener los datos en bruto y presentarlos de forma estandarizada al resto del sistema.

- Ejemplo 1: Una GPSContextSource se conecta al receptor GPS del móvil y publica eventos con coordenadas periódicamente.
- Ejemplo 2: Una BloodPressureContextSource se comunica con un tensiómetro bluetooth y publica los valores de presión sistólica y diastólica.

Las Fuentes de Contexto son los vértices iniciales de nuestra cadena. No dependen de otras fuentes; obtienen el dato directamente del hardware o de una API externa.

2. Gestor de Contexto (Context Manager)

Es el componente que procesa, agrega, infiere o predice nueva información a partir de una o más fuentes (u otros gestores). Los Gestores de Contexto son los que crean la "magia" de la abstracción.

- Ejemplo de agregación: Un LocationContextManager que recibe coordenadas GPS de una fuente, coordenadas de torres de telefonía de otra, y señales de Wifi de una tercera, y las combina para dar una estimación de ubicación más precisa y fiable.
- Ejemplo de inferencia: Un ActivityContextManager que recibe datos de acelerómetro y giroscopio y ejecuta un algoritmo (p.ej., basado en redes neuronales) para determinar si el usuario está quieto, caminando, corriendo, o subiendo escaleras.
- Ejemplo de predicción: Un TrafficContextManager que recibe datos históricos de tráfico, el día de la semana y la hora actual, y predice cuánto tiempo se tardará en llegar al trabajo.

Los Gestores de Contexto pueden consumir de Fuentes de Contexto o de otros Gestores, creando así una jerarquía o, más precisamente, un grafo acíclico dirigido de procesamiento.

Un Ejemplo Visual (Imagina la Jerarquía)

Nivel más bajo (Fuentes):

- GPSContextSource → Coordenadas brutas.
- CellIDContextSource → Identificador de antena de telefonía.
- WiFiContextSource → Lista de MACs de redes wifi cercanas.

Nivel intermedio (Gestores que agregan o mejoran):

- HybridLocationManager → Consume las tres fuentes anteriores y produce una ubicación "fusionada" con mejor precisión.
- RoadSegmentManager → Consume el HybridLocationManager y, usando un mapa digital, determina en qué calle o carretera está el usuario.

Nivel alto (Gestores que infieren semántica):

- PlaceTypeManager → Consume de RoadSegmentManager y de una base de datos de puntos de interés (POIs) para determinar si el usuario está en "zona comercial", "residencial", "universidad", etc.
- DangerousSituationManager → Consume de PlaceTypeManager y de un sensor de velocidad para activar una alarma si está en zona escolar a más de 30 km/h.

Ventajas de este patrón

1) **Modularidad**: Cada componente tiene una responsabilidad única y bien definida. Se pueden añadir, quitar o reemplazar gestores sin afectar al resto.
2) **Reutilización**: Un mismo Gestor de Contexto (p.ej., ActivityManager) puede ser usado por múltiples aplicaciones (una app de salud, una app de seguridad, una app de productividad).
3) **Eficiencia**: El filtrado se puede hacer en los niveles más bajos, evitando que información irrelevante suba por la jerarquía y consuma recursos.
4) **Escalabilidad**: Podemos distribuir los diferentes gestores en distintas máquinas según su carga computacional.

Combinado con el patrón ECA, este patrón de jerarquías nos permite construir desde los datos más crudos hasta los eventos semánticos de más alto nivel.

2.6 Patrón 3: Arquitectura de Acciones - Desacoplando la Respuesta

Hasta ahora, hemos visto cómo detectar eventos (patrón ECA) y cómo procesar y abstraer el contexto (patrón de jerarquías). Pero nos falta un tercer elemento fundamental: ¿cómo ejecutamos las acciones de forma flexible, desacoplada y componible?

El patrón ECA, en su formulación más simple, asume que la acción es algo concreto y bien definido, como "enviar un SMS". Pero en la realidad, las acciones pueden ser mucho más complejas:

- Pueden ser compuestas: "notificar a los médicos y llamar a los familiares y activar la alarma local".
- Pueden tener dependencias: "intenta notificar por SMS; si no es posible, notifica por WhatsApp; si tampoco, por llamada de voz".
- Pueden ser parametrizables: "muestra un mensaje que depende del nombre del paciente y de su ubicación actual".
- Pueden necesitar seleccionar entre diferentes implementaciones (un proveedor de SMS u otro, según coste o disponibilidad).

Para manejar esta complejidad, necesitamos el Patrón de Arquitectura de Acciones.

Separar el "Qué" del "Cómo"

La idea central de este patrón es desacoplar el propósito abstracto de una acción de su implementación concreta.

- Propósito de la acción: Describe qué se quiere hacer, de forma declarativa. Ejemplo: "enviar un mensaje de alerta al cuidador del paciente".
- Implementación de la acción: Describe cómo se hace eso realmente. Ejemplo: "usando el servicio SMS del operador

Movistar, con el texto 'ALERTA: Juan necesita ayuda', al número 654123987".

Este desacoplamiento permite una flexibilidad enorme. Podemos cambiar la implementación de una acción (porque el operador de SMS subió el precio, o porque el usuario prefiere WhatsApp) sin tener que modificar las reglas ECA que la invocan.

Los Componentes de la Arquitectura de Acciones

El patrón propone tres tipos de componentes principales:

1. Solicitante de Acción (Action Requester)

Es el componente que necesita que se ejecute una acción. Normalmente, será el Controlador (del patrón ECA) que, al dispararse una regla, solicita la ejecución de una o varias acciones. El Solicitante trabaja con propósitos de acción, no con implementaciones concretas.

2. Resolvedor de Acción (Action Resolver)

Es el componente más inteligente de este patrón. Su responsabilidad es recibir un propósito de acción y determinar la mejor forma de realizarlo (o la secuencia de formas). Para ello, puede:

- Descomponer una acción compuesta en acciones más simples.
- Seleccionar entre múltiples proveedores de acción el más adecuado según criterios (coste, calidad, latencia, etc.).

- Gestionar dependencias: Si la acción "A" falla, ejecutar la acción "B" como alternativa.
- Invocar finalmente a las implementaciones concretas.

3. Proveedor de Acción (Action Provider) / Implementador (Implementor)

Son los componentes que realmente ejecutan las acciones. Ofrecen una implementación concreta para uno o varios propósitos de acción. Cada Proveedor se registra en el sistema indicando qué propósito(s) sabe cumplir y en qué condiciones (p.ej., coste por uso, tiempo medio de respuesta).

- Ejemplo de implementación: SMSTelefonicaProvider implementa el propósito sendAlertMessage utilizando la API de mensajería de Telefónica.
- Otro ejemplo: WhatsAppProvider implementa el mismo propósito sendAlertMessage, pero usando la API de WhatsApp Business.

Un Ejemplo de Coordinación de Acciones

Imaginemos una regla ECA que dice: AlarmaEpileptica → EnviarAyuda(paciente)

El Controlador solicita el propósito EnviarAyuda al Resolvedor de Acción.

El Resolvedor, conocedor de las políticas (quizás configuradas por el hospital), actúa así:

1) Intenta primero notificar a los médicos mediante la acción compuesta NotificarMedicos.
2) NotificarMedicos se descompone en: EnviarSMS + HacerLlamada (ejecutadas en paralelo).
3) Si EnviarSMS falla (p.ej., sin cobertura), entonces ejecuta EnviarWhatsApp como respaldo.
4) Independientemente de lo anterior, ejecuta RegistrarEventoEnLog para auditoría.

El Resolvedor selecciona los Proveedores concretos:

- Para EnviarSMS: elige el proveedor con mejor relación coste-velocidad del momento.
- Para EnviarWhatsApp: solo hay un proveedor, ese.
- Para HacerLlamada: elige un proveedor de VoIP porque es más barato que la red tradicional.

Y así, el Resolvedor orquesta toda la secuencia, mientras que las reglas ECA se mantienen limpias y legibles, ignorando toda esta complejidad.

Beneficios de este Patrón

- Flexibilidad máxima: Las aplicaciones pueden cambiar dinámicamente la forma de ejecutar las acciones sin necesidad de parar el sistema ni modificar el código de las reglas.
- Extensibilidad: Añadir un nuevo proveedor de acción (un nuevo servicio de mensajería, una nueva forma de notificación) es trivial. Solo hay que registrarlo, las reglas ya existentes pueden empezar a usarlo.
- Mantenibilidad: La lógica orquestadora está concentrada en el Resolvedor, no dispersa en cientos de reglas.
- Reutilización: Un mismo Resolvedor puede servir a múltiples aplicaciones diferentes.

2.7 El Rol de los "Stakeholders": Quién crea, provee y usa la plataforma

Hasta ahora hemos hablado de componentes técnicos: controladores, fuentes de contexto, resolvedores de acciones. Pero detrás de cada componente hay personas, roles y organizaciones. Para que una plataforma de software sensible al contexto sea viable en el mundo real, debemos entender quién hace qué y quién se beneficia de qué.

En la jerga de negocio y de ingeniería de software, a estos actores se les llama stakeholders (partes interesadas). La investigación identifica varios roles clave.

2.7.1 Diseñador de la Plataforma

Es el arquitecto de software que diseña y construye la plataforma base. Define las interfaces, los modelos de contexto estándar, el núcleo del controlador, y proporciona las directrices para extender la plataforma.

Ejemplo: El equipo de ingeniería de una gran empresa tecnológica que decide construir un ecosistema de dispositivos inteligentes para el hogar. Ellos definen cómo se conectan los sensores, cómo se registran los servicios, etc.

2.7.2 Proveedor de Negocio de la Plataforma

Es la entidad (empresa, organización) que comercializa y gestiona la plataforma. Ofrece los servicios de la plataforma a los desarrolladores de aplicaciones y a los usuarios finales, a menudo mediante suscripciones, contratos o acuerdos de nivel de servicio (SLA).

Ejemplo: La misma empresa tecnológica tiene un departamento de negocio que decide cuánto cobrar por acceso a la API, qué garantías de disponibilidad ofrecer, y cómo resolver disputas.

2.7.3 Proveedor de Servicio de la Plataforma

Es el responsable técnico de mantener la plataforma operativa. Configura los servidores, monitoriza el rendimiento, aplica parches de seguridad, gestiona las copias de seguridad, etc. Es el "ingeniero de operaciones" o "SRE" (Site Reliability Engineer).

Ejemplo: El equipo de DevOps que asegura que los servidores de la plataforma de domótica nunca se caigan y respondan en menos de 100 milisegundos.

2.7.4 Diseñador de Aplicación

Es el desarrollador o equipo de desarrollo que construye la aplicación sensible al contexto concreta usando los servicios de la plataforma. No le importan los detalles de bajo nivel de los sensores; se centra en la lógica de su aplicación (salud, turismo, oficina, etc.).

Ejemplo: Una pequeña startup que desarrolla una app para monitorizar a pacientes con Alzheimer. Usan la plataforma de domótica para obtener ubicación y datos de presencia, pero añaden su propia lógica de alertas y su propia interfaz de usuario.

2.7.5 Proveedor de Negocio de la Aplicación

Es la entidad que comercializa y gestiona la aplicación concreta. Puede ser la misma startup o una empresa que distribuye la app. Establece la relación comercial con los usuarios finales.

Ejemplo: La startup vende la app de Alzheimer a residencias de ancianos y a familias particulares, mediante suscripción mensual.

2.7.6 Proveedor de Servicio de la Aplicación

Es quien mantiene operativa la aplicación (actualizaciones, servidores de la app, base de datos de usuarios, etc.). A veces coincide con el Proveedor de Negocio; a veces se externaliza.

2.7.7 Diseñador de Procesador de Contexto (Fuente o Gestor)

Son desarrolladores especializados que construyen componentes de contexto concretos (una fuente de GPS, un gestor de actividad, un gestor de lugar de trabajo) para que luego los usen los diseñadores de aplicaciones.

Ejemplo: Una empresa de telemetría desarrolla un gestor TrafficManager que fusiona datos de miles de coches y provee información de tráfico en tiempo real. Ofrece ese gestor como un servicio a través de la plataforma.

2.7.8 Proveedores de Negocio y Servicio de Procesador de Contexto

Análogos a los anteriores, pero para los componentes de contexto. Un "Proveedor de Contexto" puede ofrecer datos de localización de alta precisión, o de calidad del aire, o de afluencia de personas. Otros desarrolladores pagan por usar esos datos en sus aplicaciones.

2.7.9 Usuario Final (End-User)

Es la persona que utiliza la aplicación sensible al contexto para mejorar su vida o su trabajo. No interactúa directamente con la

plataforma (quizás ni sabe que existe). Su relación es con la aplicación y, a través de ella, confía algunos datos (con su permiso) a la plataforma.

Ejemplo: Un familiar de un paciente con Alzheimer que instala la app en su móvil para recibir alertas si su ser querido se sale de una zona segura.

Un Ecosistema de Colaboración

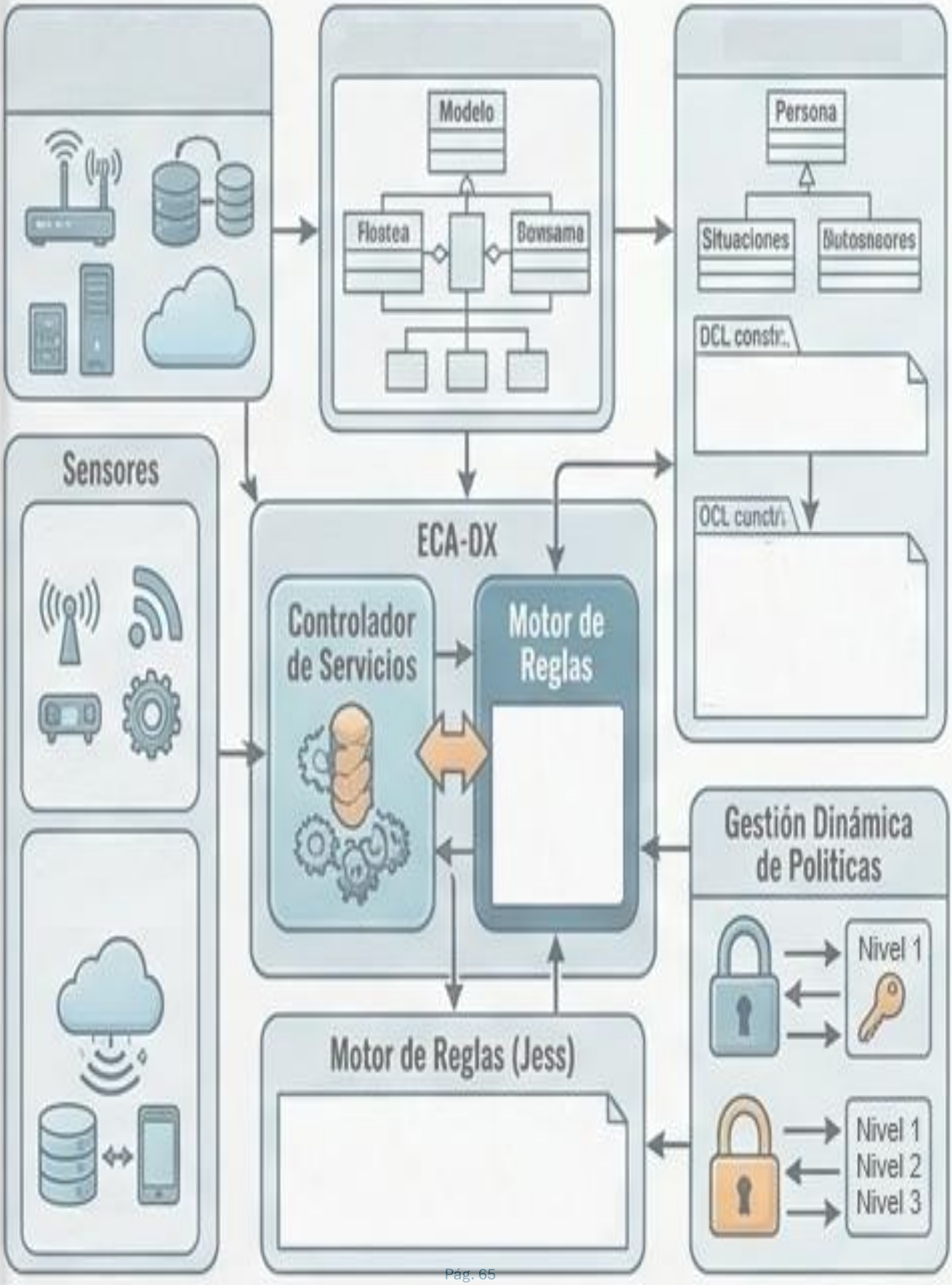
Lo importante de este análisis de stakeholders es darse cuenta de que ninguna entidad puede hacerlo todo sola. Una aplicación verdaderamente sensible al contexto y desplegada a gran escala requiere la colaboración de:

- Diseñadores de plataforma (que ponen los cimientos)
- Proveedores de contexto (que aportan datos)
- Proveedores de acción (que ejecutan respuestas)
- Desarrolladores de aplicaciones (que integran todo y crean valor para el usuario)

La arquitectura ECA-DX que exploraremos en el próximo capítulo está pensada precisamente para facilitar este ecosistema de colaboración, definiendo interfaces claras y responsabilidades bien delimitadas para cada tipo de stakeholder.

En el siguiente capítulo, pondremos todo este conocimiento en práctica y presentaremos la Arquitectura ECA-DX (Dominio Extendido), nuestra propuesta concreta para el desarrollo de aplicaciones sensibles al

contexto. Veremos cómo se integran los tres patrones (ECA, jerarquías de contexto, arquitectura de acciones) en una solución coherente, robusta y escalable.


Modelo
Floatea
Bowsame
Persona
Situaciones
Blutosneores
DCL constr..
OCL constr..
Sensores
ECA-DX
Controlador de Servicios
Motor de Reglas
Gestión Dinámica de Politicas
Nivel 1
Motor de Reglas (Jess)
Nivel 1
Nivel 2
Nivel 3

CAPÍTULO 3: La Arquitectura Propuesta: ECA-DX (Dominio Extendido)

Hasta ahora hemos recorrido el camino conceptual: entendimos el problema del software "ciego" ante el contexto (Capítulo 1) y estudiamos los patrones fundamentales que nos dan las herramientas para construir aplicaciones reactivas y proactivas (Capítulo 2). Llegó el momento de presentar nuestra propuesta concreta: la arquitectura ECA-DX (Evento-Control-Acción con Dominio Extendido).

Este capítulo es el corazón técnico del libro. No te preocupes si aparecen términos como "motor de reglas", "Jess", "UML" u "OCL". Los explicaremos paso a paso, con ejemplos, y verás cómo todas las piezas encajan para formar un sistema robusto, escalable y, sobre todo, útil para construir aplicaciones que realmente entienden el mundo.

3.1 Integrando la Extensión "DominioX" (DX) para Fuentes Híbridas

El patrón ECA clásico, como vimos en el capítulo anterior, es una excelente base para la reactividad. Sin embargo, cuando intentamos llevarlo a la práctica en el mundo real, nos encontramos con un obstáculo: las fuentes de información contextual son muy diversas y a menudo "hablan" en lenguajes distintos.

Imaginemos una aplicación típica de asistencia médica. Necesita integrar:

- **Sensores físicos:** pulsera de frecuencia cardíaca (Bluetooth), sensor de movimiento (acelerómetro del móvil), GPS.
- **Datos de usuario:** perfil del paciente (edad, alergias, medicación), preferencias de notificación (SMS, llamada, push).

- **APIs externas:** servicio de geocodificación (convertir coordenadas en nombre de calle), servicio de mensajería (Twilio, WhatsApp Business).
- Datos históricos: patrones de crisis anteriores, registros médicos.

Cada una de estas fuentes tiene su propio formato, frecuencia, latencia y fiabilidad. Integrarlas directamente en el controlador ECA sería un caos: el controlador se volvería monolítico, difícil de mantener y de extender.

La idea de "DominioX" (DX)

La extensión DominioX (DX) propone añadir una capa intermedia, que llamaremos Plataforma de Gestión de Contexto, situada entre las fuentes de información (el mundo real) y el controlador ECA. Esta capa tiene dos responsabilidades principales:

1) **Estandarizar la comunicación:** todas las fuentes, sean del tipo que sean, se "conectan" a la plataforma mediante interfaces bien definidas. La plataforma traduce los datos específicos de cada fuente a un formato común (por ejemplo, objetos Java o documentos JSON con un esquema predefinido).
2) **Jerarquizar el procesamiento:** aplica el patrón de Fuentes de Contexto y Gestores de Jerarquía (visto en 2.5) dentro de la propia plataforma. De esta forma, la complejidad de agregar, inferir y predecir queda encapsulada en componentes reutilizables, y el controlador solo recibe eventos de alto nivel y estados semánticos.

En la investigación original, esta arquitectura se representa gráficamente (ver siguiente Figura 4):

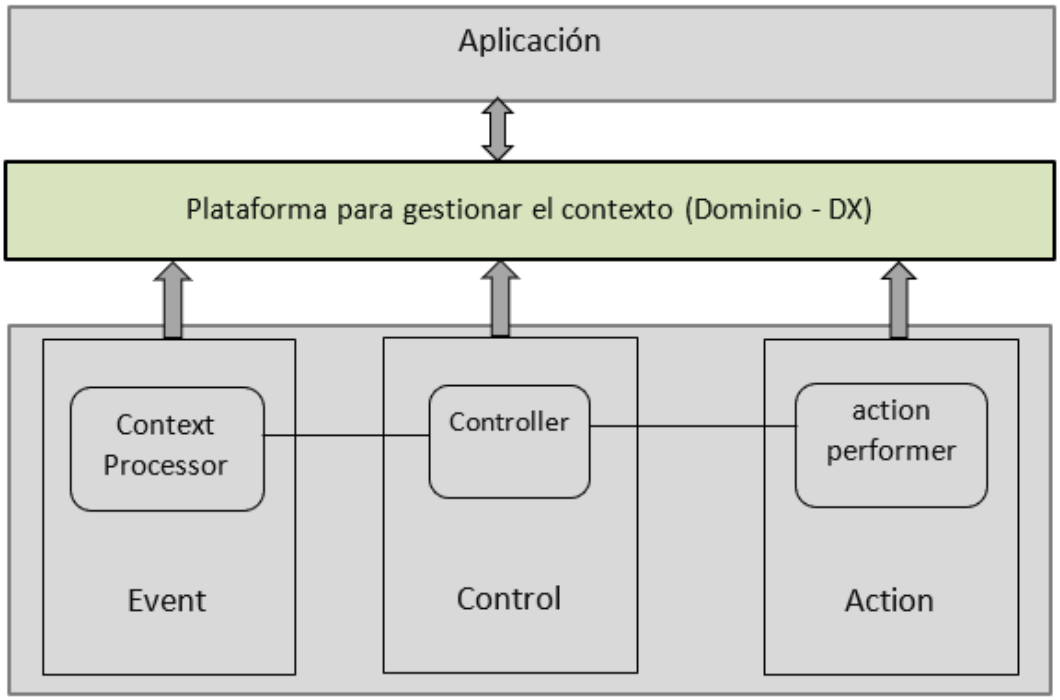


Figura 4: Arquitectura de Software Para el Desarrollo de Aplicaciones Sensibles al Contexto - ECA-DX-Modelo Propuesto (Iparraguirre Villanueva, 2017)

¿Qué aporta esta capa DX?

- **Abstracción**: el desarrollador de la aplicación no necesita saber si un dato proviene de un GPS, de Bluetooth o de la entrada manual del usuario. Solo pide "ubicación del paciente" y la plataforma se la da.
- **Reutilización**: un mismo gestor de contexto (por ejemplo, ActivityRecognizer) puede ser usado por múltiples aplicaciones (salud, seguridad, productividad).

- **Mantenibilidad**: si cambia un sensor (dejas de usar GPS y pasas a usar red Wifi para localización), solo hay que modificar la fuente de contexto correspondiente, sin tocar el controlador ni las reglas de la aplicación.
- **Escalabilidad**: los gestores y fuentes pueden distribuirse en diferentes máquinas o contenedores, como veremos en el apartado de rendimiento.

Fuentes de contexto híbridas: un ejemplo concreto

Retomemos el caso del paciente epiléptico. Algunas fuentes de contexto serán:

- EpilepticAlarmContextSource: no es un sensor físico, sino un componente software que ejecuta un algoritmo médico sobre las señales del corazón. Produce eventos como EpilepticAlarm(paciente).
- GeoLocationContextSource: se conecta al GPS del móvil o a la red de telefonía. Produce mediciones periódicas de latitud/longitud.
- CaregiverStatusContextSource: obtiene el estado "disponible/ocupado" que el cuidador introduce manualmente en su app. Es una fuente híbrida porque combina entrada humana con un sensor de presencia (si el cuidador está en su puesto de trabajo).

Gracias a la capa DX, todas estas fuentes se registran en la plataforma con su interfaz estandarizada (métodos subscribe, unsubscribe, query). El controlador no distingue entre ellas; simplemente se suscribe a los eventos que necesita (por ejemplo, EpilepticAlarm y GeoLocation).

3.2 El Corazón del Sistema: El Controlador de Servicios y el Motor de Reglas (Jess)

El Controlador es el cerebro de toda la arquitectura. Es el componente que recibe los eventos del mundo (a través de la plataforma DX), evalúa las reglas ECA y, cuando se cumplen, ordena la ejecución de acciones (Costa, 2007); (Maatjes, 2008).

En nuestra propuesta, el Controlador no es una caja negra monolítica. Se compone de varios subcomponentes que trabajan en armonía. El más importante de ellos es el Motor de Reglas.

¿Por qué un motor de reglas?

Podríamos implementar la lógica de las reglas ECA con simples sentencias if-then-else en Java o Python. Sin embargo, cuando el número de reglas crece (por ejemplo, una aplicación de hogar inteligente puede tener decenas o cientos de reglas), y cuando las condiciones son complejas (combinaciones de eventos, ventanas de tiempo, dependencias), el código imperativo se vuelve inmanejable (Frühwirth, 2025) .

Un motor de reglas es un software especializado que permite:

- Declarar las reglas de forma declarativa (decir qué debe ocurrir, no cómo evaluarlo).
- Añadir, modificar o eliminar reglas en tiempo de ejecución, sin parar la aplicación.

- Utilizar algoritmos eficientes (como el famoso algoritmo Rete) para evaluar cientos o miles de reglas contra una base de hechos cambiante, con un rendimiento aceptable.

En la investigación original (Iparraguirre Villanueva, 2017) se eligió Jess (Java Expert Shell System) como motor de reglas. Jess es una implementación del algoritmo Rete completamente escrita en Java. Es madura, rápida y se integra muy bien con aplicaciones Java empresariales (Friedman-Hill, 2003); (Prasad, 2022).

El lenguaje de reglas ECA-DL

Para que Jess entienda nuestras reglas ECA, necesitamos un lenguaje intermedio. Se definió un pequeño lenguaje de dominio específico llamado ECA-DL (Event-Condition-Action Domain Language). Una regla ECA-DL típica para el caso médico se ve así:

```
Scope (EpilepticPatient.*; p)
{
  Upon EpilepticAlarm(p)
  When p.hasHazardousActivity.hazardousvalue == true
  Do NotifyPatientApplication(p)
}
```

Traducción: Para cada paciente epiléptico, cuando ocurra una alarma y el paciente esté realizando una actividad peligrosa, notifica a su aplicación.

ECA-DL es muy legible incluso para no programadores, lo que facilita que médicos o cuidadores puedan definir sus propias reglas.

El mapeo a Jess

Internamente, el Controlador transforma cada regla ECA-DL en una o más reglas del lenguaje Jess. Jess tiene su propia sintaxis basada en Lisp, pero el programador no necesita conocerla porque la traducción es automática. Aquí un ejemplo simplificado de cómo se ve una regla Jess equivalente a la anterior:

```
(defrule notify-patient-on-danger
    (EpilepticAlarm {patientID ?p})
    (HazardousActivity {patientID ?p value true})
    =>
    (call ?*action-resolver* notifyPatientApplication ?p)
    )
```

El Controlador se encarga de:

- Recibir la suscripción de la aplicación (por ejemplo, cuando el paciente activa el servicio de monitorización).
- Parsear la regla ECA-DL y generar el código Jess correspondiente.
- Cargar la regla en el motor de Jess.
- Alimentar la memoria de trabajo de Jess con los hechos (eventos y estados) que llegan de las fuentes de contexto.

- Escuchar las ejecuciones de reglas y, cuando una regla se dispara, invocar al Resolvedor de Acciones.

Distribución del motor de reglas (DJess)

Una de las innovaciones de la investigación es el uso de DJess, una extensión de Jess que permite que varios motores de reglas, ejecutándose en distintas máquinas, compartan una memoria de trabajo común. Esto es crucial para la escalabilidad.

Imaginemos que tenemos 10.000 pacientes monitorizados. No podemos tener un único motor de reglas centralizado procesando todos los eventos, porque se convertiría en un cuello de botella. Con DJess, podemos desplegar múltiples controladores, cada uno gestionando un subconjunto de pacientes, y sincronizar la información relevante (por ejemplo, la ubicación de los cuidadores). En la investigación (Iparraguirre Villanueva, 2017) se demostró que esta configuración distribuida mejora drásticamente los tiempos de reacción (ver Figuras 5 y 6).

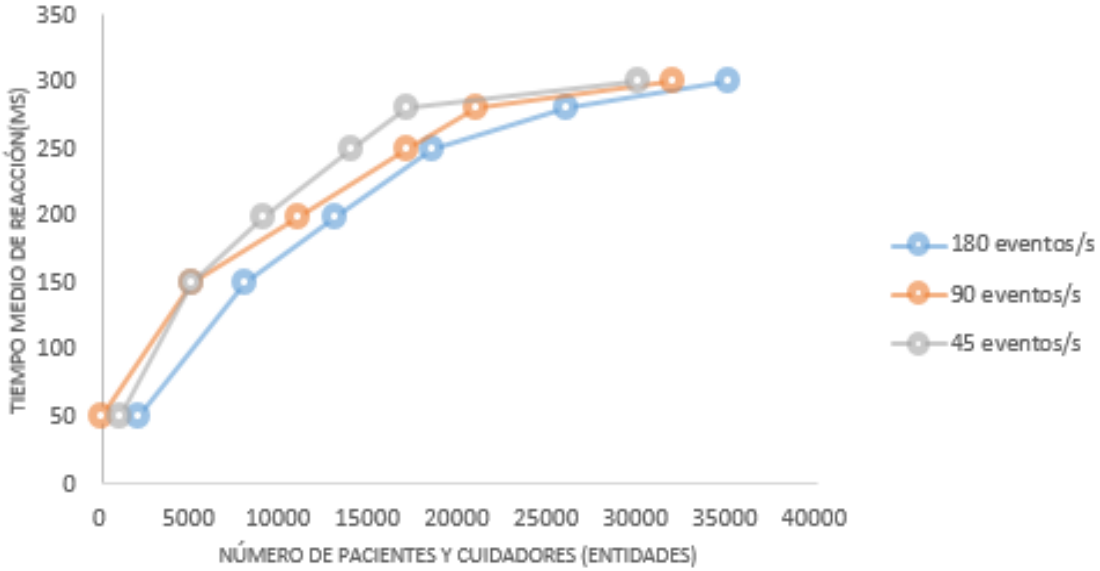


Figura 5: Tiempos de reacción promedio de ECARule2 y ECARule3 en una configuración distribuida (Iparraguirre Villanueva, 2017)

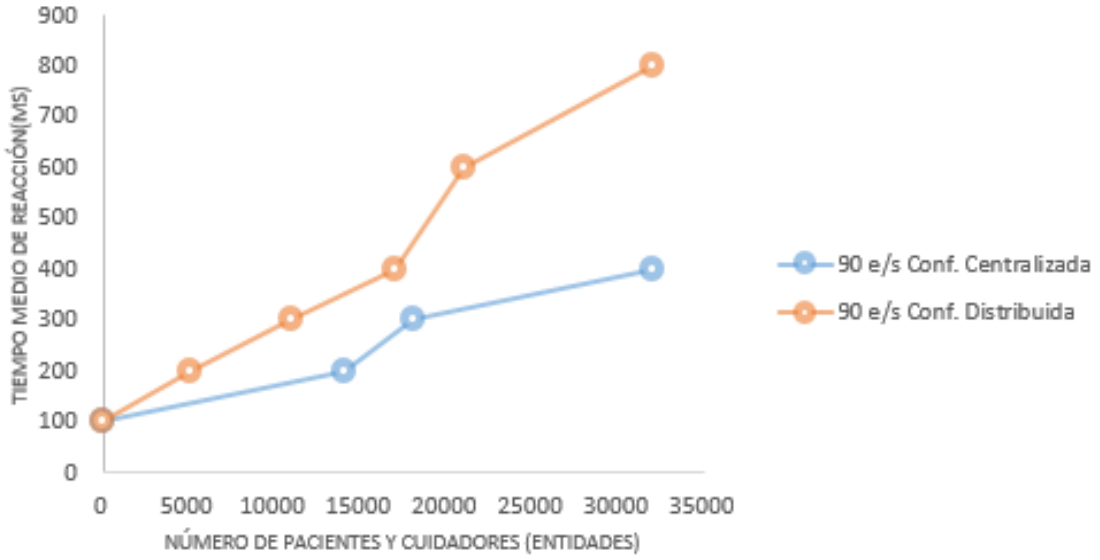


Figura 6: Comparación de los tiempos de reacción entre la configuración Centralizada y Distribuida (Iparraguirre Villanueva, 2017)

3.3 Modelando la Realidad: Cómo Construir un Modelo de Contexto Eficaz

Antes de escribir una sola regla, necesitamos entender qué "cosas" existen en nuestro universo de aplicación y qué relaciones hay entre ellas. Es decir, necesitamos un modelo de contexto.

Un modelo de contexto no es más que un mapa conceptual que representa:

Entidades: los objetos o seres que pueblan nuestro mundo. Ejemplos: PacienteEpileptico, Cuidador, Edificio, Sensor.

- **Contexto intrínseco:** propiedades que describen a una entidad. Ejemplos: GeoLocation (ubicación), CaregiverStatus (estado disponible/ocupado), HazardousActivity (si el paciente está en una actividad peligrosa).
- **Relaciones:** cómo se conectan las entidades. Ejemplos: un paciente recibe tratamiento de un profesional de la salud; un cuidador está asignado a un paciente.

En la investigación, este modelado se realiza con diagramas de clases UML (Lenguaje Unificado de Modelado). UML es un estándar muy conocido en ingeniería de software, por lo que los desarrolladores se sentirán cómodos con él.

Ejemplo: modelo de contexto para la asistencia a epilepsia

La siguiente tabla resume las entidades y sus atributos (basada en la Figura 7):

Entidad / Contexto	Atributos principales	Relaciones
Persona (superclase)	(ninguno directo)	
PacienteEpileptico	hasHazardousActivity: Boolean	Tratamiento (ProfesionalSalud), VoluntaryCare (Cuidador)
Cuidador	hasCareStatus: CaregiverStatusEnum	VoluntaryCare (PacienteEpileptico)
ProfesionalSalud		Tratamiento (PacienteEpileptico)
GeoLocation	coordinates: GeoLocationCoordinates	Asociado a Persona (cada persona tiene una ubicación)
CaregiverStatusEnum (enumerado)	OnCall, NotOnCall, Busy, EmergencyOnly	

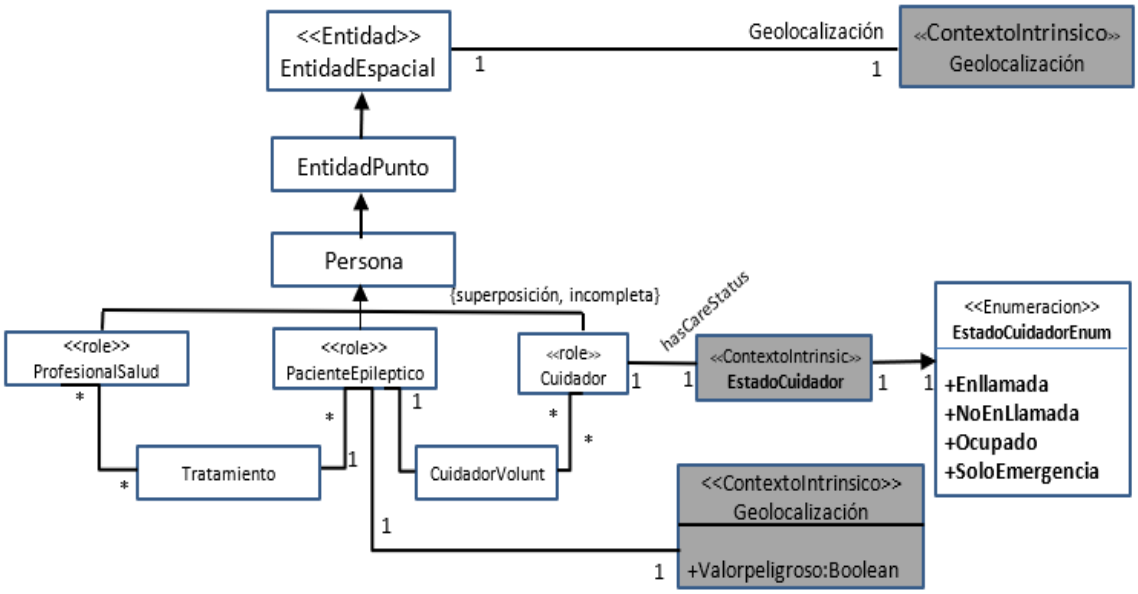


Figura 7: Modelo de contexto de la aplicación (adaptado de (Iparraguirre Villanueva, 2017))

Además, se definen operaciones útiles, como isCaregiverOf(caregiver, patient) para comprobar si un cuidador está asignado a un paciente.

¿Por qué es tan importante este modelo?

Porque todas las reglas ECA-DL, todas las consultas a las fuentes de contexto y todas las inferencias de los gestores se basarán en este modelo. Si modelamos mal, la aplicación funcionará mal. Por ejemplo, si olvidamos incluir el atributo HazardousActivity, no podremos discriminar si el paciente está conduciendo y por tanto las alertas podrían ser menos precisas.

El modelo de contexto actúa como un lenguaje común entre los distintos stakeholders (médicos, diseñadores de la plataforma, programadores). Asegura que todos hablen el mismo idioma.

De lo conceptual a lo implementable

El modelo UML es conceptual, pero necesitamos llevarlo a código. La investigación (Iparraguirre Villanueva, 2017) propone una correspondencia directa:

- Cada clase del diagrama UML se convierte en una clase Java (o en una plantilla Jess).
- Cada atributo se convierte en un campo de la clase.
- Las relaciones (asociaciones) se convierten en referencias entre objetos.

Además, se definen tipos de datos de medición que encapsulan no solo el valor del contexto, sino también metadatos de calidad: antigüedad (freshness), precisión, origen (si viene de un sensor o de entrada manualmente), y probabilidad de corrección. Esto permite que el sistema tome decisiones teniendo en cuenta la fiabilidad de la información.

3.4 De la Especificación a la Realización: Definiendo "Situaciones" con UML y OCL

Un modelo de contexto nos dice qué puede ser cierto (por ejemplo, "un paciente tiene una ubicación"). Pero una aplicación sensible al contexto no reacciona a hechos aislados; reacciona a situaciones que son combinaciones de hechos que tienen un significado especial para el usuario.

Una situación es un estado compuesto de mayor nivel. Ejemplos:

- SituationCaregiverAvailable: un cuidador está disponible (su estado es OnCall o EmergencyOnly).
- SituationCaregiverWithinRange: un cuidador está a menos de 100 metros del paciente.

En la investigación (Iparraguirre Villanueva, 2017), las situaciones se especifican formalmente utilizando diagramas de clases UML (para la estructura) y restricciones OCL (para la lógica). OCL (Object Constraint Language) es un lenguaje declarativo que permite escribir condiciones sobre los objetos UML (Standards Development Organization, 2014).

Ejemplo de especificación de situación

En la Figura 8 del documento original, se define SituationCaregiverAvailable mediante una restricción OCL que dice (en lenguaje natural): *"Una instancia de SituationCaregiverAvailable existe si hay un cuidador cuyo estado (CaregiverStatus) tiene valor 'OnCall' o 'EmergencyOnly'."*

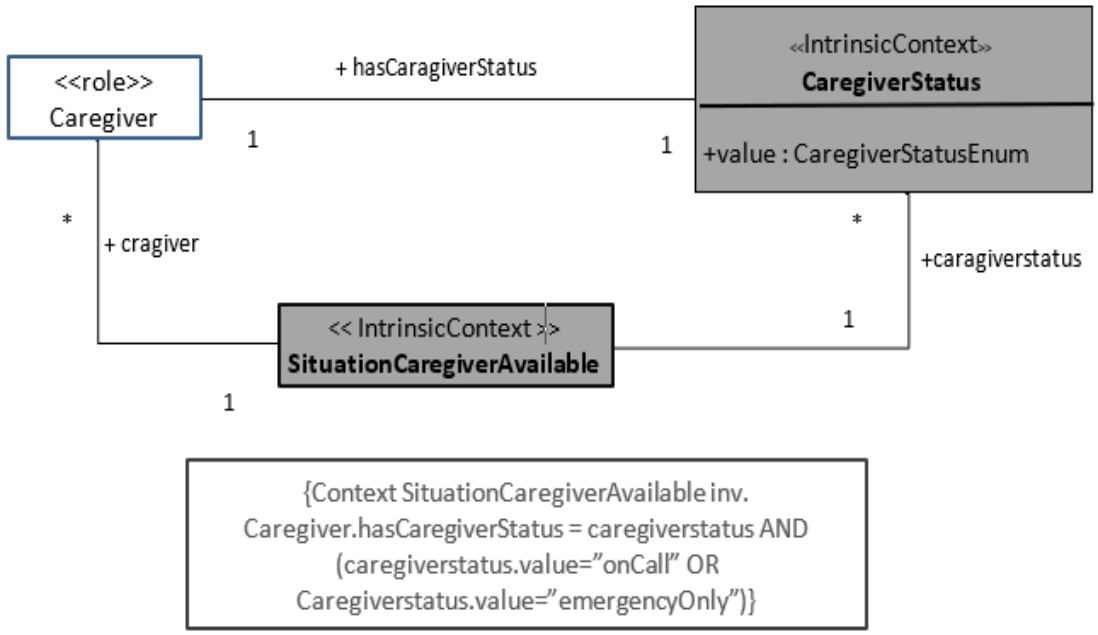


Figura 8: Especificación de la Situación SituationCaregiverAvailable (adaptado de (Iparraguirre Villanueva, 2017))

En OCL sería algo así:

```
context SituationCaregiverAvailable inv:

  self.caregiver.hasCaregiverStatus.value = CaregiverStatusEnum::OnCall or

  self.caregiver.hasCaregiverStatus.value = CaregiverStatusEnum::EmergencyOnly
```

Del modelo a los componentes realizadores

Una vez especificada la situación, debemos realizarla en código. La plataforma DX incluye componentes llamados Gestores de Contexto (Context Managers) que se encargan de evaluar continuamente las condiciones de una situación y notificar a los suscriptores cuando la situación comienza o termina.

Por ejemplo, SituationWithinRangeContextManager se suscribe a las fuentes de ubicación del paciente y de los cuidadores, calcula la distancia mediante una fórmula (ver Figura 9) y genera eventos de transición (Enter, Exit) para la situación.

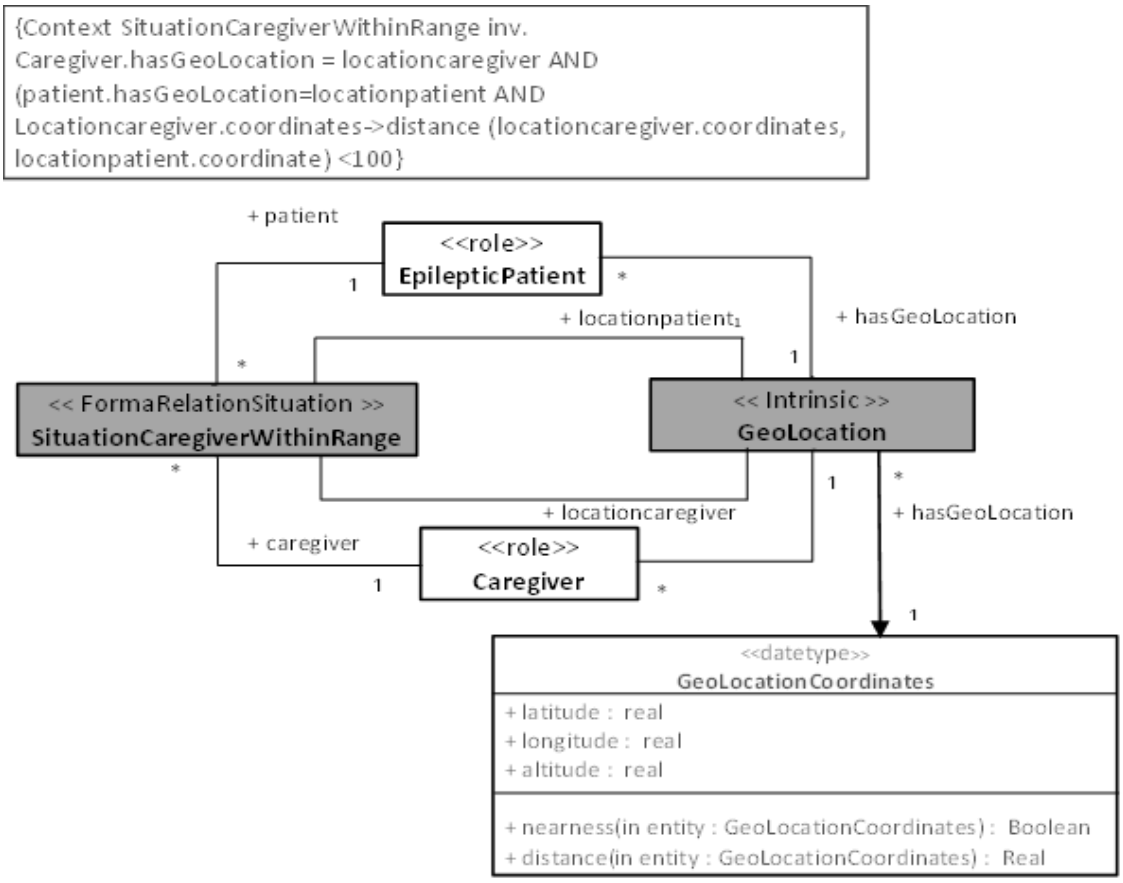


Figura 9: Rango entre paciente y medido: SituationCaregiverWithin (Iparraguirre Villanueva, 2017)

El desarrollador de la aplicación no necesita escribir el código de este gestor. La plataforma puede generarlo a partir de la especificación UML+OCL mediante transformaciones automáticas (siguiendo el enfoque MDA - Model Driven Architecture). En la investigación se cuantifica el esfuerzo ahorrado: 663 líneas de Java y 126 de Jess se derivan sistemáticamente de solo 23 clases UML y 9 líneas de OCL (ver la siguiente tabla). Esto demuestra la poderosa simplificación que aporta el modelado.

Comparación entre la especificación y los esfuerzos de realización

UML clases(units)	OCL(LDC)	ECA-DX (LDC)	Java(LDC)	Jess(LDC)
23	9	21	663	126

**Basado en datos de la investigación original (Iparraguirre Villanueva, 2017)*

3.5 Gestión Dinámica de Políticas: Privacidad y Control de Acceso al Contexto

No basta con saber detectar situaciones y reaccionar. Las aplicaciones sensibles al contexto manejan información muy delicada: ubicación de las personas, datos de salud, estado de ánimo, etc. Es imprescindible garantizar que solo las entidades autorizadas accedan a estos datos y que los usuarios tengan control sobre quién ve qué.

En nuestra arquitectura, la gestión de políticas de acceso y privacidad está integrada desde el diseño, no como un añadido posterior.

¿Qué es una política de contexto?

Una política es una regla que establece qué acciones puede realizar un sujeto sobre un objeto bajo ciertas condiciones contextuales. Por ejemplo:

- *"Un cuidador solo puede acceder a las constantes vitales del paciente durante la ventana de tiempo en que existe una situación de alarma."*
- *"Un investigador no puede ver la ubicación exacta de un paciente, solo una región agregada (distrito, ciudad)."*

Políticas estáticas vs. dinámicas

Los sistemas tradicionales usan políticas estáticas (el administrador define «el usuario X tiene permiso para leer el recurso Y»). En entornos ubicuos, esto es insuficiente, porque las entidades entran y salen del entorno dinámicamente (un cuidador que estaba libre ahora está ocupado, un médico de guardia cambia cada día).

La propuesta ECA-DX introduce políticas dinámicas basadas en contexto. Es decir, las propias reglas ECA se encargan de otorgar o revocar permisos cuando se detectan ciertas situaciones.

Implementación mediante reglas ECA y componentes de enforcement

En el documento se desarrolla una aplicación de gestión de políticas superpuesta a la misma arquitectura. Las reglas ECA-DX no solo invocan acciones de notificación, sino también acciones de

concesión/denegación de acceso sobre las fuentes de contexto. Por ejemplo:

Cuando ocurra una alarma epiléptica, otorga acceso a las bioseñales del paciente a todos los cuidadores que estén disponibles y cerca, y además otorga acceso al estado de esos cuidadores para que el paciente los vea.

Esto se traduce en invocaciones a operaciones como grantAccessBiosignals(paciente, listaCuidadores) sobre la fuente de contexto EpilepticAlarmContextSource. La fuente de contexto, que es quien realmente posee los datos, aplica el filtro de acceso antes de enviar cualquier notificación.

Componentes específicos para políticas

La arquitectura añade interfaces de Policy Enforcement (ver Figura 10) con métodos como:

- grantAccessBiosignals(entities, duration)
- denyAccessBiosignals(entities)
- increaseTrustLevelCaregiver(caregiverId)
- decreaseTrustLevelCaregiver(caregiverId)

Estos métodos son invocados por el Resolvedor de Acciones cuando se disparan reglas de política.

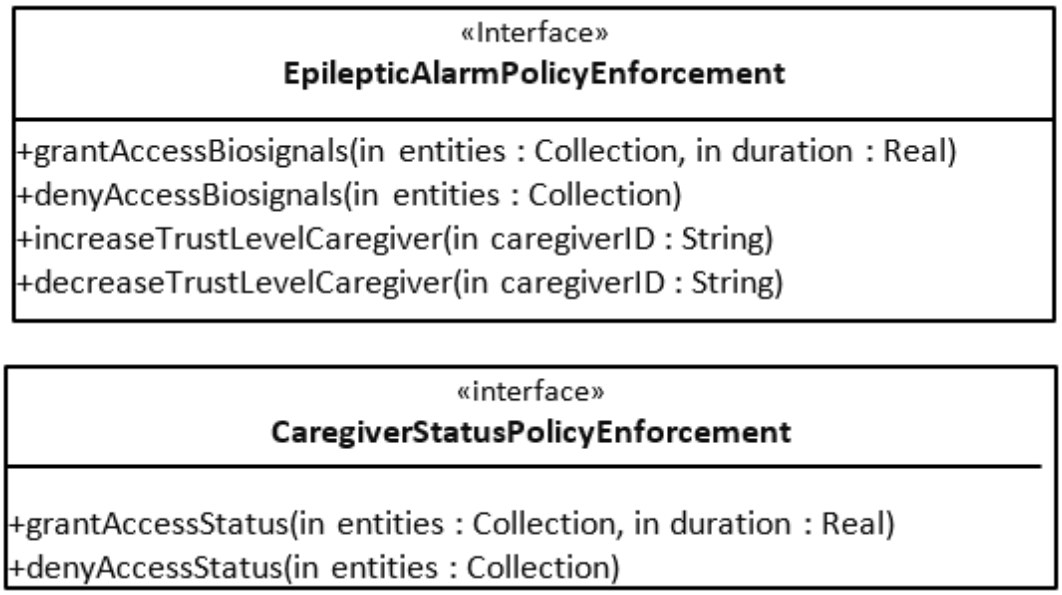


Figura 10: implementación de políticas por componentes (Iparraguirre Villanueva, 2017)

Ventajas de este enfoque

- **Granularidad fina:** los permisos se pueden otorgar por tiempo limitado (p.ej., 30 minutos) o revocar automáticamente al terminar la situación.
- **Transparencia para el desarrollador de aplicaciones**: la lógica de seguridad está en las reglas ECA, no dispersa en el código de la interfaz de usuario.
- **Auditoría:** el mismo sistema puede registrar cada concesión/denegación y notificar al administrador de políticas (ver la interfaz AdministratorActionService en la Figura 11).

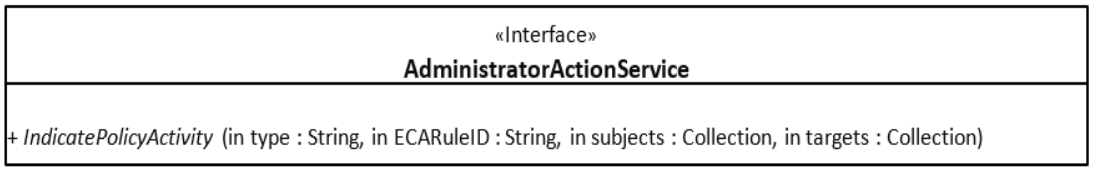


Figura 11: Interfaz de servicio del administrador (Iparraguirre Villanueva, 2017)

En resumen, la extensión DX no solo se ocupa de la eficiencia y la escalabilidad, sino también de los aspectos éticos y legales, tan necesarios en aplicaciones que manejan datos personales.

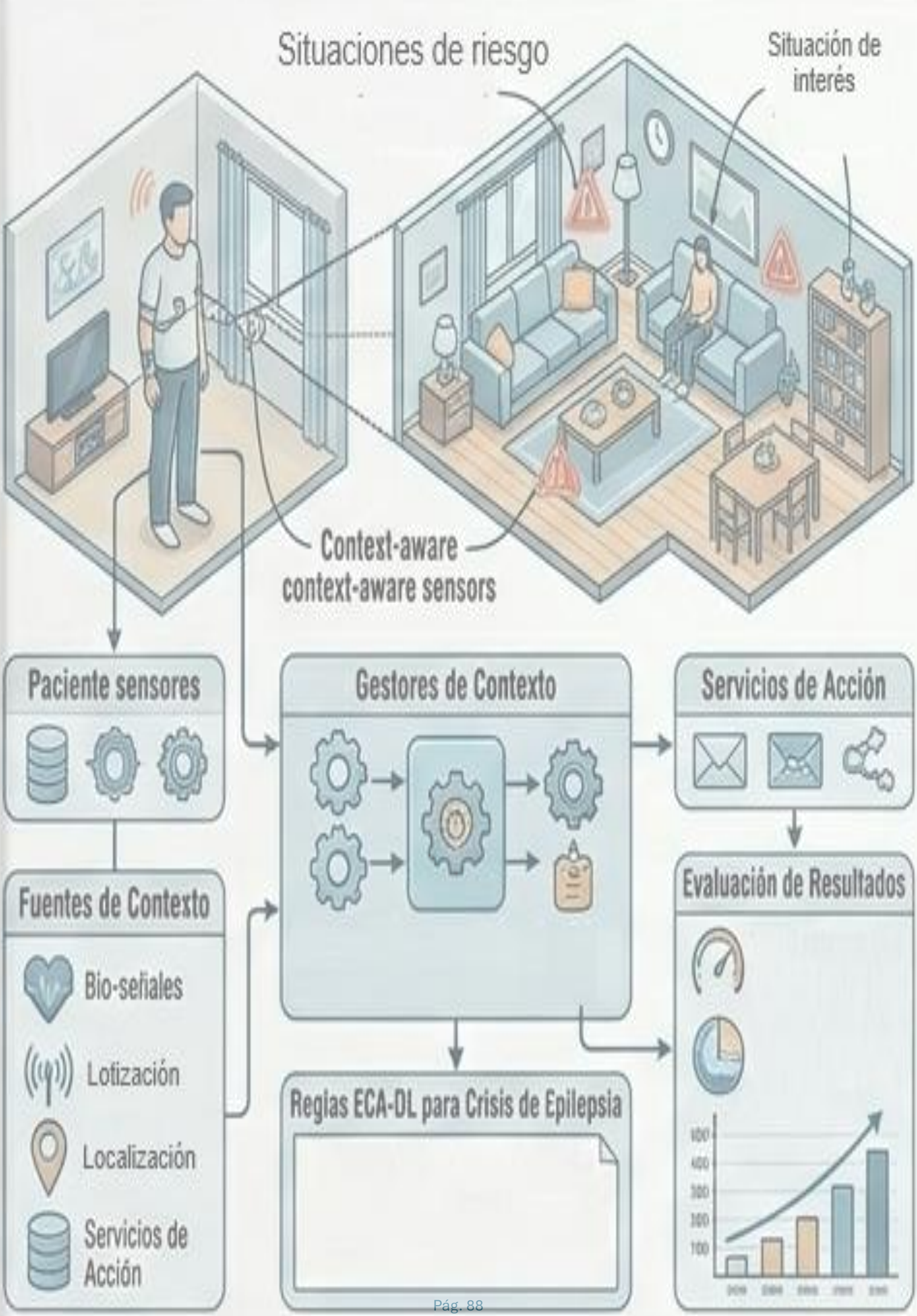
Resumen del capítulo

Hemos presentado la arquitectura ECA-DX como una evolución del patrón ECA clásico, añadiendo:

1) Una plataforma de gestión de contexto (DominioX) que abstrae y jerarquiza las fuentes de información híbridas.
2) Un controlador basado en motor de reglas Jess que ejecuta reglas declarativas ECA-DL, con capacidad de distribución (DJess) para lograr escalabilidad.
3) Un modelado de contexto formal mediante UML, que permite describir entidades, atributos y relaciones.
4) Una especificación de situaciones con restricciones OCL, que se realiza automáticamente en componentes gestores de contexto.

Un sistema de políticas dinámicas que utiliza las propias reglas ECA para gestionar acceso y privacidad, integrándose de forma natural con la arquitectura.

Con todos estos pilares, estamos listos para pasar a la acción. En el Capítulo 4 construiremos paso a paso una aplicación real: un sistema de monitorización de pacientes con epilepsia que desplegaremos, evaluaremos y analizaremos. Verás cómo la teoría se convierte en código funcionando, y cómo los tiempos de reacción y la escalabilidad se comportan en la práctica.


Situaciones de riesgo
Situación de interés
Context-aware
context-aware sensors
Paciente sensores
Gestores de Contexto
Servicios de Acción
Fuentes de Contexto
Bio-señales
Lotización
Localización
Servicios de Acción
Evaluación de Resultados
Reglas ECA-DL para Crisis de Epilepsia

CAPÍTULO 4: Caso Práctico – Monitorización de Salud para Pacientes con Epilepsia

Hasta ahora hemos construido un andamiaje teórico sólido: conocemos los patrones (ECA, jerarquías de contexto, arquitectura de acciones) y hemos diseñado la plataforma ECA-DX. Llegó el momento de poner todas las piezas en movimiento y construir una aplicación real.

En este capítulo te llevaré de la mano por el desarrollo completo de un sistema de asistencia autónoma para personas con epilepsia. Veremos cómo el paciente, sus cuidadores y los profesionales de la salud se convierten en actores de un ecosistema donde el software reacciona por sí solo ante una crisis, sin que nadie tenga que pulsar un botón de emergencia.

Además, mediremos el rendimiento de la aplicación: ¿es lo suficientemente rápida para alertar antes de que ocurra una convulsión? ¿Puede manejar a cientos o miles de pacientes simultáneamente? Las respuestas están en los números.

4.1 El Escenario: Un Paciente, su Entorno y la Necesidad de Asistencia Autónoma

Carlos y su día a día

Carlos tiene 45 años, vive solo y sufre epilepsia refractaria. A pesar de la medicación, cada cierto tiempo puede sufrir una crisis convulsiva. Su mayor miedo es que ocurra mientras conduce o cuando está solo en casa. Hasta ahora dependía de un colgante de alerta que, en caso de ataque, debe pulsar él mismo… pero durante una convulsión es imposible.

Su neuróloga, la Dra. Gómez, le propone probar una aplicación de monitorización continua que se ejecuta en su reloj inteligente y en su móvil. La aplicación recibe señales de su frecuencia cardíaca y de sus movimientos; gracias a un algoritmo predictivo, puede avisar con minutos de antelación de una alta probabilidad de crisis. Además, cuando la crisis es inminente, el sistema contacta automáticamente con los cuidadores más cercanos y con el servicio médico.

Los actores del escenario

En la aplicación intervienen tres tipos de usuarios, cada uno con un rol distinto:

- **Paciente (Carlos):** recibe notificaciones en su reloj/móvil cuando el sistema predice una crisis. Puede confirmar o cancelar la alerta si es un falso positivo.
- **Cuidadores (voluntarios):** pueden ser familiares, vecinos o personal de una residencia. Se ofrecen voluntarios para ayudar. La aplicación les mostrará un mapa con la ubicación de Carlos cuando se active una alarma, y ellos podrán aceptar o rechazar la solicitud de ayuda.
- **Profesionales de la salud (Dra. Gómez y enfermeros):** reciben un registro de los eventos para poder afinar el tratamiento. No intervienen en tiempo real, pero el sistema les notifica si la crisis es grave o si ningún cuidador acepta la ayuda.

¿Qué debe hacer la aplicación?

Los requisitos funcionales, expresados en lenguaje natural, son:

1) **Detección automática:** cuando el algoritmo predictivo (integrado en el reloj) genera una alarma epiléptica, la aplicación debe reaccionar inmediatamente.
2) **Notificación al paciente:** si Carlos está realizando una actividad peligrosa (p.ej., conduciendo), el sistema debe enviarle un aviso personalizado para que detenga la actividad.
3) **Búsqueda de cuidadores:** el sistema debe localizar a los cuidadores que estén disponibles (estado OnCall o EmergencyOnly) y que se encuentren a menos de 100 metros de Carlos. A todos ellos les enviará una solicitud de ayuda con la ubicación.
4) **Gestión de respuestas:** el primer cuidador que acepte deberá ser confirmado. El resto recibirá un mensaje indicando que la ayuda ya está en camino.
5) **Registro médico:** cada alarma y cada respuesta se registran en el historial clínico de Carlos para su posterior análisis.

Todo esto debe ocurrir sin intervención humana directa (salvo la confirmación de los cuidadores, que es una acción explícita pero muy simple). El sistema es autónomo y proactivo.

4.2 Paso a Paso: Modelando el Contexto y las Situaciones de Riesgo

Antes de escribir código, aplicamos la metodología del Capítulo 3: modelar el contexto y definir las situaciones.

Modelo de contexto (UML)

En la Figura 7, mencionada en el capítulo 3.3 se representa el diagrama de clases. Aquí lo resumimos en una tabla:

Clase / Contexto	Atributos y operaciones
PacienteEpileptico	• hasHazardousActivity: Boolean • isPatientOf(profesional): Boolean
Cuidador	• hasCareStatus: CaregiverStatusEnum • isCaregiverOf(paciente): Boolean
ProfesionalSalud	• hasTreatment(paciente): Boolean
GeoLocation	• coordinates: GeoLocationCoordinates (lat, lon, alt) • distance(otraCoordenada): Real
CaregiverStatusEnum	Valores: OnCall, NotOnCall, Busy, EmergencyOnly
HazardousActivity	Valor booleano (true/false)

Además, definimos eventos primitivos:

- EpilepticAlarm(paciente): generado por el algoritmo predictivo.
- RejectEpilepticAlarm(paciente): si el paciente cancela la alerta manualmente.

- AcceptHelpRequest(paciente, cuidador): cuando un cuidador pulsa "aceptar" en su app.
- RejectHelpRequest(paciente, cuidador): cuando un cuidador rechaza.

Situaciones de interés

Identificamos dos situaciones que gatillarán las reglas:

1) SituationCaregiverAvailable

   Un cuidador está disponible si su CaregiverStatus es OnCall o EmergencyOnly.

2) SituationCaregiverWithinRange

   Un cuidador está cerca del paciente si la distancia entre sus coordenadas es inferior a 100 metros.

En la Figura 8 y 9 del subcapítulo 3.4 se muestran las especificaciones OCL. Por ejemplo, para la segunda situación:

```
context SituationCaregiverWithinRange inv:
  patient.hasGeoLocation = locationPatient and
  caregiver.hasGeoLocation = locationCaregiver and
  locationCaregiver.distance(locationPatient) < 100
```

Estas restricciones se traducirán automáticamente a código en los gestores de contexto.

Tipos de datos de medición

No solo almacenamos el valor del contexto, sino también metadatos de calidad: antigüedad, precisión, probabilidad de corrección. Por ejemplo, GeoLocationMeasurement incluye:

- personID
- coordinates
- precision (en metros)
- freshness (cuándo se tomó la medida)
- probabilityOfCorrectness (0..1)

Esto permite que el motor de reglas, si lo desea, pueda ignorar datos poco fiables (p.ej., una ubicación GPS con baja precisión). Con el modelo completo, podemos pasar a la implementación.

4.3 Implementando los Componentes: Fuentes de Contexto, Gestores y Servicios de Acción

La implementación se realizó en Java (JDK 1.6+), usando RMI (Java Remote Method Invocation) para la distribución de componentes y Eclipse como entorno de desarrollo. El motor de reglas es Jess (v7.1). Para la simulación de sensores (porque no teníamos acceso a pacientes reales), se generaron eventos aleatorios pero con patrones realistas (Oracle Java Platform, 2026).

Fuentes de contexto (Context Sources)

Desarrollamos cinco fuentes:

Fuente	Datos que produce	Modo de simulación
EpilepticAlarmContextSource	Evento EpilepticAlarm(paciente)	Generación aleatoria con probabilidad controlada (ej. 1 evento cada 5 min)
GeoLocationContextSource	Mediciones de ubicación periódicas	Coordenadas que varían lentamente (simulan movimiento)
HazardousActivityContextSource	Booleano true/false	Se activa aleatoriamente un 20% del tiempo
CaregiverStatusContextSource	Estado OnCall, Not OnCall, etc.	Simula cambios manualmente (cada 15 min, cambia de estado)
AcceptRequestContextSource	Evento AcceptHelpRequest(paciente, cuidador)	Generado cuando un cuidador simulado pulsa "Aceptar"

Todas las fuentes implementan una interfaz común que permite suscribirse (subscribe) y recibir notificaciones por callback. Las suscripciones se gestionan mediante identificadores únicos.

Gestores de contexto (Context Managers)

Dos gestores se encargan de las situaciones:

- SituationWithinRangeContextManager
  Se suscribe a GeoLocationContextSource de todos los pacientes y cuidadores. Para cada par (paciente, cuidador) calcula la distancia cada vez que alguna de las ubicaciones cambia. Si la distancia baja de 100 m genera el evento SituationCaregiverWithinRangeEnter(paciente, cuidador); si sube por encima, genera el evento Exit.
- SituationAvailableContextManager
  Se suscribe a CaregiverStatusContextSource. Cuando un cuidador cambia su estado a OnCall o EmergencyOnly, genera el evento SituationCaregiverAvailableEnter(cuidador); en caso contrario, Exit.

Cada gestor corre su propio motor Jess interno (una instancia ligera). Esto permite que el razonamiento de la situación esté distribuido y sea eficiente.

Servicios de acción (Action Services)

Son componentes que se ejecutan en los dispositivos de los usuarios. Ofrecen métodos remotos que el Resolvedor de Acciones puede invocar:

- PatientActionService (en el móvil de Carlos)
  indicateEpilepticSeizure() → muestra una alerta vibrante y sonora.

- CaregiverActionService (en los móviles de los cuidadores)
  requestHelp(patientID, coordinates) → muestra un mapa con la ubicación y un botón "Aceptar"/"Rechazar".
  indicateAcceptance(patientID, caregiverID) → notifica que otro cuidador ya ha aceptado.
- ProfessionalActionService (en la tablet de la Dra. Gómez)
  logSeizure(patientID) → registra el evento en la historia clínica.

El controlador central (Healthcare Controller)

Es el componente que orquesta todo. Se suscribe a los eventos de los gestores y a los eventos primarios de las fuentes. Su configuración es la siguiente (ver Figura 12):

- Recibe reglas ECA-DL (las veremos en 4.4) desde la aplicación a través de una interfaz de administración.
- Internamente, contiene un motor de reglas Jess donde se cargan las reglas traducidas.
- Cuando una regla se dispara, invoca al Resolvedor de Acciones Médico (un componente que conoce qué acción llamar y con qué parámetros).

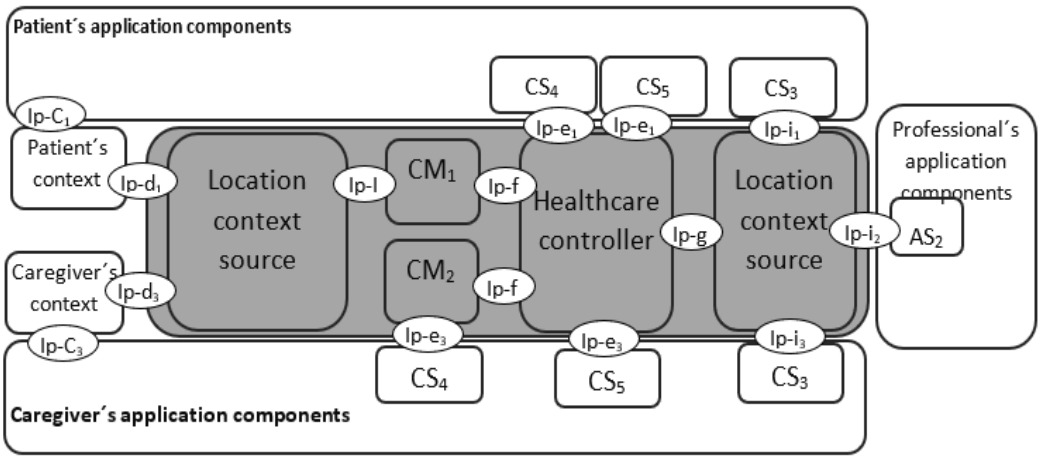


Figura 12: Configuración de los Componentes para la Aplicación (Iparraguirre Villanueva, 2017)

El Resolvedor se encarga de convertir el propósito abstracto (p.ej., NotifyCaregiversApplications) en invocaciones concretas a los servicios de acción de los cuidadores correspondientes.

Simulación de usuarios y eventos

Para las pruebas se creó una población simulada de hasta 35.000 entidades (pacientes, cuidadores y profesionales). Los parámetros de simulación incluyen:

- Número de pacientes (de 10 a 10.000).
- Número de cuidadores por paciente (1 a 5).
- Frecuencia de eventos EpilepticAlarm: de 10 a 450 eventos por segundo (valores extremos para pruebas de estrés).

Todo el sistema se desplegó en máquinas con 3-4 GB de RAM.

4.4 Definiendo las Reglas de Comportamiento (ECA-DL) para la Resolución de Ayuda

Con los componentes implementados, el comportamiento de la aplicación se define mediante reglas ECA-DL. A continuación presento las cuatro reglas principales, explicadas en lenguaje natural y con su equivalente formal.

Regla 1: Avisar al paciente si hay alarma y actividad peligrosa

Lenguaje natural:

Cuando se detecte una alarma epiléptica del paciente Carlos, y además Carlos esté realizando una actividad peligrosa (p.ej., conduciendo), entonces notifica a su aplicación.

ECA-DL:

```
Scope (EpilepticPatient.*; p)
{
  Upon EpilepticAlarm(p)
  When p.hasHazardousActivity.hazardousvalue == true
  Do NotifyPatientApplication(p)
}
```

Qué hace?

El Controlador, al recibir EpilepticAlarm, consulta el estado de HazardousActivity del paciente. Si es true, invoca al Resolvedor con la

acción NotifyPatientApplication, que a su vez llamará a indicateEpilepticSeizure en el móvil de Carlos.

Regla 2: Notificar a cuidadores cercanos y disponibles

Lenguaje natural:

Cuando ocurra una alarma epiléptica, busca a todos los cuidadores que estén asignados a ese paciente, que estén disponibles (SituationCaregiverAvailable) y que se encuentren a menos de 100 metros (SituationWithinRange). Envía a todos ellos una solicitud de ayuda.

ECA-DL:

```
        Scope (EpilepticPatient.*; p)
{
   Upon EpilepticAlarm(p)
   Do NotifyCaregiversApplications(
      p,
      p.hasGeoLocation.coordinates,
      Select(Caregiver.*; care;
          isCaregiverOf(care, p) and
          SituationWithinRange(p, care) and
          SituationCaregiverAvailable(care))
   )
}
```

La cláusula Select es una consulta que se evalúa en el momento de la alarma, usando el conocimiento actual de las situaciones (suministradas por los gestores). Devuelve una colección de identificadores de cuidadores. La acción NotifyCaregiversApplications enviará a cada uno una notificación push con la ubicación.

Regla 3: Informar a los demás cuidadores cuando alguien acepta

Lenguaje natural:

Después de una alarma, si un cuidador acepta la solicitud, se debe notificar al resto de cuidadores que la ayuda ya está cubierta, para que no se dupliquen esfuerzos.

ECA-DL:

```
Scope (EpilepticPatient.*; p)
{
  Upon Ev1: EpilepticAlarm(p); Ev2: AcceptHelpRequest(p)
  Do notifyAcceptanceCaregivers(
    Select(Caregiver.*; care;
        isCaregiverOf(care, p) and
        SituationWithinRange(care, p) and
        care != Ev2.caregiverID),
    p,
```

```
        Ev2.caregiverID
      )
    }
```

Observa la composición de eventos: Ev1 seguido de Ev2. La regla solo se dispara si el evento AcceptHelpRequest ocurre después de la alarma para el mismo paciente. El Select excluye al cuidador que ya aceptó. La acción notifyAcceptanceCaregivers llamará a indicateAcceptance en los móviles de los demás cuidadores.

Regla 4: Registrar la alarma en el historial médico

Lenguaje natural:

Cuando ocurra una alarma, notifica a todos los profesionales de la salud que tratan a ese paciente para que quede registro.

ECA-DL:

```
Scope (EpilepticPatient.*; p)
{
  Upon EpilepticAlarm(p)
  Do logEpilepticAlarm(p,
    Select(HealthProfessional.*; prof;
        isHealthProfessionalOf(prof, p))
  )
```

```
}
```

Esta regla es útil para estudios clínicos y para mejorar los algoritmos predictivos.

Ventajas de usar ECA-DL

- **Legibilidad**: un médico o familiar con nociones básicas puede entender la intención.
- **Modularidad**: se pueden añadir o quitar reglas sin recompilar toda la aplicación.
- **Ejecución eficiente**: el motor de reglas Jess optimiza el orden de evaluación.

4.5 Evaluación de Resultados: ¿Qué tan rápida y escalable es la propuesta?

No basta con que la aplicación funcione; debe hacerlo a tiempo y con muchos usuarios. Para evaluarlo, se definió una métrica principal: el tiempo de reacción.

Tiempo de reacción = intervalo entre que ocurre un evento (p.ej., EpilepticAlarm) y que se invoca la primera acción correspondiente (p.ej., NotifyPatientApplication).

Configuraciones de prueba

- **Configuración centralizada:** todos los componentes (fuentes, gestores, controlador) se ejecutan en una única máquina con 4 GB de RAM.
- **Configuración distribuida:** las fuentes de contexto se ejecutan en una máquina (3 GB) y los gestores + controlador en otra (4 GB). La comunicación vía RMI.

Se variaron dos parámetros:

- **Número de entidades:** hasta 35.000 entre pacientes, cuidadores y profesionales.
- **Frecuencia de eventos:** entre 45 y 450 eventos por segundo (tasa de alarmas generadas).

Resultados en configuración centralizada

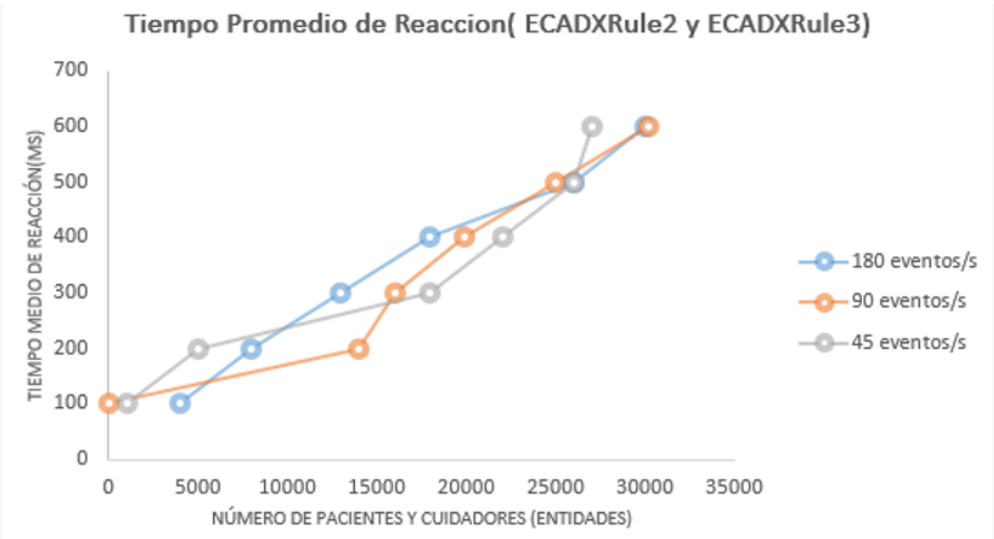


Figura 13: Tiempos de reacción promedio de ECARule2 y ECARule3 en una configuración centralizada (Iparraguirre Villanueva, 2017)

- Con 45 eventos/s y 90 eventos/s, el tiempo de reacción crece linealmente con el número de entidades: desde unos 12 ms para 500 entidades hasta aproximadamente 120 ms para 10.000 entidades.
- Con 180 eventos/s, se observa un punto de saturación alrededor de 7.500 entidades: el tiempo de reacción se vuelve errático (saltos de 200 ms a más de 500 ms). A partir de ahí, el motor Jess no puede procesar los eventos tan rápido como llegan, y comienza a encolarlos. Si la cola crece demasiado, algunos eventos se pierden (caen fuera de la ventana de detección).

Análisis: el 85% del tiempo de reacción se debe al procesamiento interno de Jess (algoritmo Rete). El otro 15% es la comunicación entre componentes. La saturación se debe a que la memoria y la CPU no son infinitas en una sola máquina.

Resultados en configuración distribuida

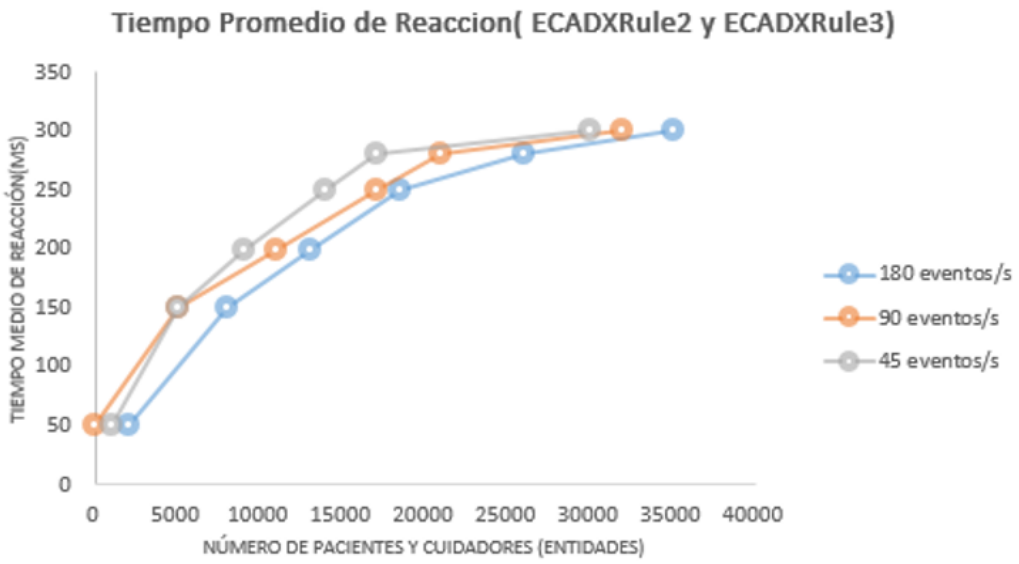


Figura 14: Tiempos de reacción promedio de ECARule2 y ECARule3 en una configuración distribuida (Iparraguirre Villanueva, 2017)

- Las curvas son más suaves y predecibles. Incluso con 35.000 entidades y 180 eventos/s, el tiempo de reacción se mantiene por debajo de 200 ms, sin picos de saturación.
- La mejora se explica porque la carga de simular las fuentes de contexto (que consumen mucha memoria) está en una máquina separada, dejando más recursos para los motores Jess. Además, la red de inferencia distribuida (DJess) permite compartir hechos sin necesidad de pasar eventos uno a uno.

Comparación directa

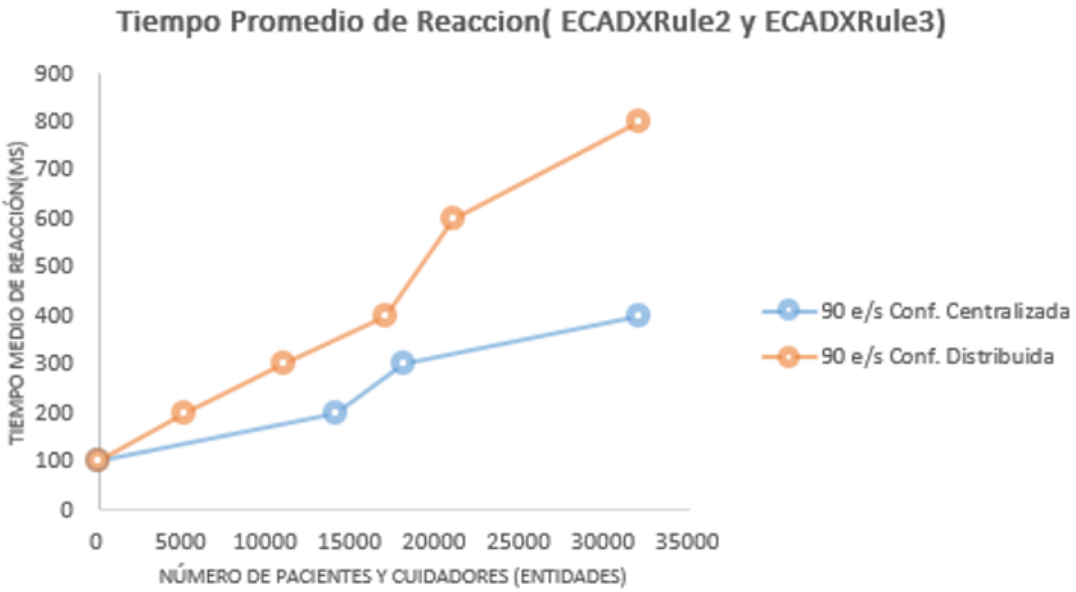


Figura 15: Comparación de los tiempos de reacción entre la configuración Centralizada y Distribuida (Iparraguirre Villanueva, 2017)

Para 90 eventos/s, la configuración centralizada es más rápida hasta 7.500 entidades (~45 ms frente a ~60 ms). La latencia de red en la configuración distribuida penaliza ligeramente. Sin embargo, a partir de 10.000 entidades, la configuración distribuida se vuelve claramente

superior: el tiempo centralizado se dispara a 300 ms, mientras el distribuido sigue estable en ~100 ms.

Conclusión práctica: para aplicaciones con menos de 5.000 usuarios concurrentes, una configuración centralizada es suficiente y más sencilla. Para sistemas de gran escala (miles de pacientes), la arquitectura distribuida es esencial.

¿Son aceptables estos tiempos?

En una crisis epiléptica, los segundos cuentan. Nuestro sistema, incluso en el peor caso escalable, reacciona en menos de 200 milisegundos (0,2 segundos). La notificación push al cuidador tarda un poco más (depende de la red), pero el procesamiento interno es prácticamente instantáneo. Esto permite avisar al paciente con antelación suficiente (el algoritmo predictivo ya ha dado una ventana de 2-3 minutos) y activar la ayuda antes del inicio de la convulsión.

Verificación de la corrección funcional

Además de las pruebas de rendimiento, se validó que las reglas se disparaban correctamente mediante casos de prueba unitarios. Por ejemplo:

- Se inyecta un EpilepticAlarm con HazardousActivity=true. El sistema verifica que PatientActionService ha sido invocado.

- Se simula un cuidador dentro del rango y disponible. Al ocurrir la alarma, se comprueba que CaregiverActionService.requestHelp se llama para ese cuidador.
- Se simula la aceptación de un cuidador, y se comprueba que los demás cuidadores reciben indicateAcceptance.

Todos los tests fueron superados.

Reflexión final del capítulo

Hemos demostrado, con un caso médico real, que la arquitectura ECA-DX es:

- **Realizable**: se puede implementar con tecnologías estándar (Java, Jess, RMI).
- **Expresiva**: las reglas ECA-DL capturan el comportamiento complejo de forma clara.
- **Escalable**: la distribución de los componentes permite manejar decenas de miles de usuarios.
- **Rápida**: los tiempos de reacción son del orden de milisegundos, suficientes para aplicaciones de tiempo real suave (asistencia sanitaria, domótica, etc.).

En el siguiente y último capítulo discutiremos las lecciones aprendidas, las limitaciones de la propuesta y los caminos abiertos para el futuro del software sensible al contexto.


Evolución Contextual
Predicción
Aprendizaje Automático
Ética de Datos
Evolución Contextual
Recomendaciones para Desarrolladores e Investigadores
Conclusiones Clave
Lecciones Aprendidas
El Camino por Delante

CAPÍTULO 5: Más Allá del Código: Logros, Lecciones y el Próximo Paso del Software Contextual

Hemos recorrido un largo camino. Comenzamos preguntándonos por qué el software actual es “ciego” al mundo que lo rodea, exploramos los patrones que nos permiten construir aplicaciones reactivas y proactivas, diseñamos una arquitectura robusta (ECA-DX) y la pusimos a prueba en un caso real de asistencia médica. Llegó el momento de mirar atrás, evaluar lo aprendido y, sobre todo, mirar hacia delante.

Este capítulo es una conversación abierta. No encontrarás fórmulas cerradas, sino reflexiones sobre lo que funcionó, los obstáculos que aún persisten y las oportunidades que se abren para los desarrolladores, los investigadores y, en última instancia, para los usuarios que se beneficiarán de un software que realmente les entiende.

5.1 Lecciones Aprendidas: ¿Qué Funcionó y Qué Fue un Desafío?

Lo que salió bien

1. La separación de responsabilidades funciona.

La arquitectura ECA-DX divide claramente tres aspectos: la captura y abstracción del contexto (fuentes y gestores), la lógica reactiva (reglas ECA en el controlador) y la ejecución de acciones (resolvedor y proveedores). Esta separación ha demostrado ser una bendición para la mantenibilidad y la extensibilidad. En el caso práctico, pudimos añadir nuevas reglas (por ejemplo, una regla que avisara también al portero de la finca) sin tocar ni una línea de las fuentes de contexto ni los gestores.

2. El modelado formal evita el caos.

Dedicar tiempo a modelar el contexto con UML y OCL, aunque al principio pueda parecer un esfuerzo adicional, se paga con creces. El modelo actúa como un contrato compartido entre los médicos (que definen qué es una situación de riesgo) y los programadores (que implementan las reglas). En el prototipo, una restricción OCL mal interpretada habría provocado que los cuidadores a 101 metros no recibieran alertas… pero precisamente por tener el modelo escrito, pudimos detectar y corregir el error antes de pasar a código.

3. Los motores de reglas (Jess) son ideales para la reactividad.

Jess demostró ser capaz de evaluar cientos de reglas contra miles de hechos en tiempos del orden de milisegundos. Su implementación del algoritmo Rete, que recuerda resultados de evaluaciones previas, evita recompilar constantemente las condiciones. Además, la extensión DJess permitió distribuir el razonamiento, algo que resultó crítico para la escalabilidad.

4. La escalabilidad se logra distribuyendo, no centralizando.

Las pruebas de esfuerzo (hasta 35.000 entidades y 450 eventos/segundo) mostraron que una configuración centralizada se satura alrededor de 7.500 entidades. En cambio, la configuración distribuida (fuentes de contexto en una máquina, gestores y controlador en otra) mantuvo tiempos de reacción aceptables incluso con 35.000

entidades. La lección es clara: el contexto masivo excede las capacidades de una única máquina; las arquitecturas de software sensible al contexto deben ser distribuidas por diseño.

Los desafíos que nos mantienen despiertos

1. La calidad del contexto es un problema no resuelto del todo.

En nuestros experimentos simulamos sensores perfectos o con ruido controlado. En la realidad, los sensores fallan, las mediciones llegan tarde o son imprecisas. Aunque introdujimos metadatos de calidad (freshness, probabilidad de corrección), no implementamos mecanismos sofisticados para fusionar fuentes redundantes o para recuperarse de fallos. Mejorar la tolerancia a la incertidumbre sigue siendo una tarea pendiente.

2. La gestión de políticas dinámicas es compleja.

La aplicación de gestión de políticas (descrita en el Capítulo 3) demostró que se pueden otorgar y revocar permisos basándose en reglas ECA. Sin embargo, en un despliegue real con cientos de pacientes y cuidadores, el número de reglas de política podría crecer exponencialmente. Además, la interacción entre políticas (por ejemplo, una política de privacidad europea GDPR y una política local del hospital) no se abordó. Se necesitan lenguajes de políticas más expresivos y herramientas de verificación.

3. El rendimiento en el peor caso aún preocupa.

Aunque los tiempos medios eran buenos, en la configuración centralizada con alta carga observamos colas de eventos y pérdida de notificaciones. Para aplicaciones de salud, perder un EpilepticAlarm es inaceptable. Se requieren mecanismos de colas persistentes y monitorización proactiva que detecten la saturación antes de que ocurra.

4. La seguridad y la privacidad son responsabilidad de todos.

Ninguna arquitectura es segura por sí sola. Aunque añadimos interfaces de control de acceso, la implementación de la seguridad (autenticación, cifrado, auditoría) quedó fuera del alcance. En un producto real, habría que integrar estándares como OAuth 2.0 y TLS, y probablemente usar un gestor de identidades descentralizado (p.ej., blockchain para consentimientos).

5.2 Conclusiones Clave: Hacia un Estándar para el Desarrollo Contextual

Tras todo el trabajo de investigación, diseño, implementación y evaluación, podemos extraer varias conclusiones que aspiramos a que sirvan de base para futuros estándares.

Conclusión 1: El desarrollo de aplicaciones sensibles al contexto necesita una arquitectura de referencia

Las arquitecturas tradicionales (cliente-servidor, MVC, SOA) no fueron concebidas para manejar flujos continuos de eventos del mundo real. La arquitectura ECA-DX (Evento-Control-Acción con Dominio Extendido) ha demostrado ser una candidata sólida para llenar ese vacío. Proporciona:

- Un modelo de componentes claro (fuentes, gestores, controlador, resolvedor, proveedores de acción).
- Un lenguaje declarativo (ECA-DL) para definir comportamientos reactivos.
- Un proceso de desarrollo que va del modelado conceptual (UML+OCL) a la realización en código (Java+Jess).
- Una estrategia de escalabilidad mediante distribución.

Conclusión 2: El modelado del contexto debe preceder a la implementación

No se puede construir una aplicación sensible al contexto "sobre la marcha". Es imprescindible un modelo conceptual que describa entidades, atributos contextuales y relaciones. Este modelo actúa como el "andamio" sobre el que se levanta todo el resto: fuentes de contexto, gestores de situaciones, reglas ECA y políticas de acceso.

La experiencia con el caso de epilepsia muestra que la inversión en modelado se traduce en menos errores y mayor facilidad para incorporar nuevos requisitos.

Conclusión 3: La separación de la lógica reactiva en reglas ECA acelera el desarrollo

En una aplicación tradicional, añadir un nuevo comportamiento implica modificar y recompilar el código. Con ECA-DL, basta con escribir (o incluso generar desde una interfaz de usuario) una nueva regla y enviarla al controlador en tiempo de ejecución. Esto abre la puerta a que los propios usuarios (con las debidas salvaguardas) personalicen el comportamiento de sus aplicaciones.

Conclusión 4: La escalabilidad masiva exige distribución y motores de reglas eficientes

Los experimentos con Jess y DJess muestran que es posible alcanzar un rendimiento aceptable para aplicaciones con decenas de miles de usuarios si se distribuye la carga. El algoritmo Rete, diseñado originalmente para sistemas expertos, resulta sorprendentemente adecuado para la evaluación continua de reglas contextuales.

Conclusión 5: La gestión de políticas contextuales es posible y necesaria

La privacidad y el control de acceso no son añadidos opcionales. Integrar políticas dinámicas basadas en reglas ECA permite que los permisos se adapten al contexto (¿quién está cerca? ¿hay una emergencia? ¿el usuario ha dado su consentimiento explícito?). Esta aproximación es más respetuosa con la autonomía del usuario que las políticas estáticas tradicionales.

5.3 El Camino por Delante: Predicción, Aprendizaje Automático y Ética de Datos

Nuestra arquitectura se centra en la reactividad (reaccionar a eventos presentes) y en un tipo limitado de proactividad (disparar acciones cuando se cumplen condiciones). Pero el futuro del software contextual va mucho más allá.

De la reacción a la predicción

Imagina que el sistema no solo reacciona cuando ya se ha producido la alarma epiléptica, sino que aprende los patrones previos de Carlos: quizás su frecuencia cardíaca aumenta de forma característica 20 minutos antes de una crisis. Un módulo de aprendizaje automático (machine learning) podría analizar los datos históricos y predecir la crisis con mayor antelación, incluso antes de que el algoritmo clínico actual la detecte.

Incorporar ML a la arquitectura ECA-DX es uno de los trabajos futuros más prometedores. La idea sería añadir un nuevo tipo de componente: un gestor de contexto predictivo que utilice modelos entrenados (redes neuronales, árboles de decisión, etc.) para generar eventos de "alerta temprana". El controlador podría suscribirse a esos eventos igual que a los eventos sensoriales.

Razonamiento con incertidumbre

Las predicciones nunca son 100% seguras. Un enfoque interesante es la lógica difusa o las redes bayesianas. En lugar de eventos booleanos (EpilepticAlarm = true/false), se podrían manejar grados de creencia (ProbabilidadCrisis = 0.85). Las reglas ECA podrían entonces tener condiciones como When probabilidadCrisis > 0.7. Esto haría el sistema más robusto ante sensores ruidosos.

Ética de datos y cumplimiento normativo

El manejo de datos de salud (ubicación, constantes vitales) está fuertemente regulado (GDPR en Europa, HIPAA en EE.UU., leyes locales). Nuestra arquitectura debe evolucionar para garantizar:

- **Consentimiento granular:** el paciente debe poder autorizar qué datos se comparten y con quién, y poder revocarlo en cualquier momento.
- **Minimización de datos:** el sistema debe solicitar solo los datos estrictamente necesarios para cada regla.

- **Explicabilidad:** si el sistema deniega una acción por una política, debe poder explicar por qué (por ejemplo, “no se notificó al cuidador porque el paciente había desactivado temporalmente el permiso de ubicación”).
- **Auditoría automática:** registro inmutable de todas las decisiones sensibles.

Un área de investigación emergente es el uso de blockchain para gestionar los consentimientos de forma descentralizada y auditable.

Integración con el Internet de las Cosas (IoT) y el Edge Computing

Nuestro prototipo se ejecutaba en servidores (en la nube o en máquinas locales). Pero cada vez más el procesamiento de contexto se desplaza al edge (en el propio router doméstico, en la puerta de enlace del hospital o incluso en el dispositivo móvil). El beneficio es doble: menor latencia y mayor privacidad (los datos no salen del hogar). La arquitectura ECA-DX se adapta bien a este modelo: las fuentes de contexto pueden residir en el edge, los gestores de situaciones pueden desplegarse en la nube central, y el controlador puede repartirse jerárquicamente.

5.4 Recomendaciones para Desarrolladores e Investigadores

Si después de leer este libro sientes la tentación de construir tus propias aplicaciones sensibles al contexto (¡ojalá así sea!), aquí tienes algunas recomendaciones prácticas basadas en nuestra experiencia.

Para desarrolladores de software

1) **Empieza con un modelo, no con el código.**

   Usa diagramas de clases UML (o incluso pizarra y papel) para identificar entidades, atributos y relaciones. Pregunta a los usuarios (médicos, familiares, etc.) si el modelo captura lo que consideran importante. Solo cuando el modelo esté validado, empieces a implementar.

2) **Elige un motor de reglas maduro.**

   Jess es excelente pero ya tiene unos años. Alternativas modernas en el ecosistema Java son Drools (muy activo, soporte para reglas declarativas) y EasyRules (más ligero). En otros lenguajes, explora CLIPS (de donde nació Jess) o PyKnow (Python). La clave es que soporte evaluación continua y memoria de trabajo.

3) **Diseña para la distribución desde el primer día.**

   Aunque tu prototipo inicial sea centralizado, usa interfaces claras (REST, gRPC, mensajería asíncrona) que permitan después separar los componentes. Esto te ahorrará un gran dolor de cabeza cuando la aplicación crezca.

4) **No olvides los metadatos de calidad.**

   Cada medición contextual debe llevar al menos: marca de tiempo (freshness), origen (sensor, usuario, inferido) y, si es posible, un indicador de fiabilidad. El motor de reglas podrá usar estos metadatos para ignorar datos obsoletos o poco fiables.

5) **Implementa una política de privacidad por defecto restrictiva.**

   Mejor pecar de conservador: denegar el acceso hasta que una regla explícita lo conceda. Y siempre informar al usuario de qué datos se están usando y para qué.

Para investigadores

1) **Explora la combinación de ECA con aprendizaje automático.**
   ¿Cómo se pueden aprender automáticamente condiciones complejas (ej. “patrón de movimientos que precede a una caída”) y transformarlas en reglas ECA? ¿Cómo mantener el rendimiento del algoritmo Rete cuando los hechos incluyen valores continuos y probabilísticos?
2) **Investiga la gestión de conflictos entre políticas.**
   Cuando múltiples reglas (unas del paciente, otras del hospital, otras del cuidador) entran en conflicto, ¿qué política prevalece? Se necesitan mecanismos de priorización y negociación.
3) **Desarrolla herramientas de verificación formal para reglas contextuales.**
   Es muy fácil escribir una regla ECA que sea lógicamente incorrecta (p.ej., “notificar al cuidador si está a menos de 100 metros”, pero si el cuidador está a 101 metros, nunca se notifica). Herramientas de model checking podrían detectar esas situaciones antes del despliegue.
4) **Estudia la experiencia de usuario (UX) de sistemas proactivos.**
   Un sistema que actúa por sí mismo puede ser percibido como útil o intrusivo. ¿Cómo debe informar al usuario de las acciones automáticas? ¿Con qué grado de control debe contar? La psicología cognitiva tiene mucho que aportar aquí.
5) **Amplía el caso de estudio a otros dominios.**
   La epilepsia es solo un ejemplo. El mismo enfoque puede aplicarse a ciudades inteligentes (gestión de tráfico proactiva), agricultura de precisión (riego automático según humedad y previsión meteorológica)

o comercio electrónico (ofertas personalizadas según contexto). Cada nuevo dominio planteará desafíos únicos que enriquecerán la arquitectura.

Epílogo: El software que nos entiende

Hace unas décadas, Mark Weiser soñó con una tecnología que desapareciera en el fondo de nuestra vida cotidiana. Hoy, ese sueño está más cerca que nunca, pero aún nos falta un componente esencial: la capacidad del software de entender el mundo sin que se lo expliquemos constantemente.

La arquitectura ECA-DX es un paso en esa dirección. No es la solución definitiva no lo pretende pero sí un andamio sólido sobre el que construir aplicaciones que reaccionan, predicen, se adaptan y respetan a sus usuarios. Ahora la pelota está en tu tejado. Toma estos conceptos, experimenta, falla, aprende y mejora.

El futuro del software sensible al contexto no lo escribiremos nosotros desde un laboratorio; lo escribirás tú, querido lector, cuando diseñes la próxima aplicación que salve una vida, que haga más cómodo un hogar o que conecte a las personas de forma más inteligente.

BIBLIOGRAFÍA

Bardram, J. E. (2005). The java context awareness framework (JCAF) - A service infrastructure and programming framework for context-aware applications. *Lecture Notes in Computer Science*, *3468 LNCS*, 98–115. https://doi.org/10.1007/11428572_7/SAVE-RESEARCH

Bimpas, A., Violos, J., Leivadeas, A., & Varlamis, I. (2024). Leveraging pervasive computing for ambient intelligence: A survey on recent advancements, applications and open challenges. *Computer Networks*, *239*, 110156. https://doi.org/10.1016/J.COMNET.2023.110156

Cajas Ordóñez, S. A., Samanta, J., Suárez-Cetrulo, A. L., & Carbajo, R. S. (2025). Intelligent Edge Computing and Machine Learning: A Survey of Optimization and Applications. *Future Internet 2025, Vol. 17, Page 417*, *17*(9), 417. https://doi.org/10.3390/FI17090417

Chen, H., Finin, T., & Joshi, A. (2003). An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, *18*(3), 197–207. https://doi.org/10.1017/S0269888904000025

Costa, P. D. (2007). *Architectural support for context-aware applications: from context models to services platforms*. https://research.utwente.nl/en/publications/architectural-support-for-context-aware-applications-from-context/

Dai, W., Adeli, E., Luo, Z., Dash, D., Lakshmikanth, S., Durante, Z., Tang, P., Kaushal, A., Milstein, A., Fei-Fei, L., & Schulman, K. (2025). Developing ICU Clinical Behavioral Atlas Using Ambient Intelligence and Computer Vision. *NEJM AI*, *2*(2). https://doi.org/10.1056/AIOA2400590

Dey, A. K., Abowd, G. D., & Salber, D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, *16*(2–4), 97–166. https://doi.org/10.1207/S15327051HCI16234_02;PAGE:STRING:ARTICLE/CHAPTER

Fahy, P., & Clarke, S. (2004). *CASS-Middleware for Mobile Context-Aware Applications*.

Ficili, I., Giacobbe, M., Tricomi, G., & Puliafito, A. (2025). From Sensors to Data Intelligence: Leveraging IoT, Cloud, and Edge Computing with AI. *Sensors 2025, Vol. 25, Page 1763*, *25*(6), 1763. https://doi.org/10.3390/S25061763

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, *19*(1), 17–37. https://doi.org/10.1016/0004-3702(82)90020-0

Friedman-Hill, Ernest. (2003). *Jess in Action: Java Rule-Based Systems*. https://books.google.com/books/about/Jess_in_Action.html?hl=es&id=LjszEAAAQBAJ

Frühwirth, T. (2025). *Principles of Rule-Based Programming*. https://books.google.com/books/about/Principles_of_Rule_Based_Programming.html?hl=es&id=ciVOEQAAQBAJ

García, R. F. (2023). MVC: Model–View–Controller. *IOS Architecture Patterns*, 45–106. https://doi.org/10.1007/978-1-4842-9069-9_2

Gereda Hernandez, D. (2024). *Revisión Sistemática de la Literatura Sobre Comparativa de los Estilos Arquitectónicos SOA y Microservicios en Entornos de Desarrollo ágil - ProQuest*. https://www.proquest.com/openview/dbf5c1dad9e9a067cd007ff6e53b7b29/1?pq-origsite=gscholar&cbl=2026366&diss=y

González Quiroga, M. (2011). *Estudio de arquitecturas de redes orientadas a servicio*. Universitat Politècnica de Catalunya. https://hdl.handle.net/2099.1/12312

Hofer, T., Schwinger, W., Pichler, M., Leonhartsberger, G., Altmann, J., & Retschitzegger, W. (2003). Context-awareness on mobile devices - The hydrogen approach. *Proceedings of the 36th Annual Hawaii International Conference on System Sciences, HICSS 2003*, 10–19. https://doi.org/10.1109/HICSS.2003.1174831

Iparraguirre Villanueva, O. C. (2017). *Arquitectura de software para el desarrollo de aplicaciones sensibles al contexto-propuesta*. Universidad Nacional Federico Villarreal. https://hdl.handle.net/20.500.13084/1669

Kumar, S. (2017). *Technological and Business perspective of Wearable technology*. http://www.theseus.fi/handle/10024/122553

Lituma Sarmiento, A. F. (2023). *Análisis y diseño de una propuesta de sistema integral de gestión empresarial basado en una arquitectura cliente-servidor*. Universidad Católica de Cuenca. https://dspace.ucacue.edu.ec/handle/ucacue/14464

Maatjes, N. (2008, November). *Automated Transformations from ECA Rules to Jess | Guide books*. https://dl.acm.org/doi/abs/10.5555/1522508

Mouhim, S., & Lachhab, F. (2025). Towards a Context Awareness System Using IoT, AI, and Big Data Technologies. *IEEE Access*, *13*, 40302–40315. https://doi.org/10.1109/ACCESS.2025.3546865

Oracle Java Platform. (2026). *Java Remote Method Invocation API (Java RMI)*. https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/

Prasad, G. N. R. (2022). A review on open source and Expert System Shells. In *A multidisciplinary Research Journal)* (Vol. 1). www.poonamshodh.in

Raento, M., Oulasvirta, A., Petit, R., & Toivonen, H. (2005). ContextPhone: A prototyping platform for context-aware mobile applications. *IEEE Pervasive Computing*, *4*(2), 51–59. https://doi.org/10.1109/MPRV.2005.29

Ruiz del Val, A. (2025). *Geolocalización en el ámbito empresarial*. https://uvadoc.uva.es/handle/10324/77750

Standards Development Organization. (2014, February). *About the Object Constraint Language Specification Version 2.4*. https://www.omg.org/spec/OCL/

Vijayvargia, K., Saxena, P., & Bhilare, D. S. (2025). Context Management Life Cycle for Internet of Things: Tools, Techniques, and Open Issues. *Engineering, Technology & Applied Science Research*, *15*(1), 19449–19459. https://doi.org/10.48084/ETASR.9117

Weiser, M. (1991). *The Computer for the 21 st Century on JSTOR*. https://www.jstor.org/stable/24938718

Weiser, M. (1993). *Some computer science issues in ubiquitous computing*. https://doi.org/10.1145/329124.329127

Zohra Trabelsi, F., Khtira, A., & Asri, B. El. (2024). *Algorithmic Business Process Optimization: Empowering Operational Excellence with Service-Oriented Architecture (SOA) and Microservices*. https://doi.org/10.18280/isi.290627

ANEXO

A. Glosario de Términos Clave

Rete: El algoritmo Rete es un algoritmo de reconocimiento de patrones eficiente para implementar un sistema de producción de reglas. Fue creado por el Dr. Charles L. Forgy en la Carnegie Mellon University. Su primera referencia escrita data de 1974, y apareció de forma más detallada en su tesis doctoral en 1979 y un artículo científico (Forgy, 1982). Rete es hoy en día la base de muchos sistemas expertos muy famosos, incluyendo CLIPS, Jess, Drools y Soar.

OCL (Object Constraint Language) es un lenguaje para la descripción de formas de expresión en los modelos UML. Sus expresiones pueden representar invariantes, precondiciones, postcondiciones, inicializaciones, guardias, reglas de derivación, así como consultas a objetos para determinar sus condiciones de estado (Standards Development Organization, 2014).

MVC: Es el patrón Modelo, Vista y Controlador (MVC) es el más extendido para el desarrollo de aplicaciones donde se deben manejar interfaces de usuarios, este se centra en la separación de los datos o modelo, y la vista, mientras que el controlador es el encargado de relacionar a estos dos (García, 2023).

FRAMEWORK: es un término compuesto, en donde Frame se traduce como marco y Work como trabajo.

SOA: Arquitectura orientada a Servicios (SOA, siglas en inglés Service Oriented Architecture) es un paradigma de arquitectura para diseñar y desarrollar sistemas distribuidos.

ECA: Event-Control-Action (Evento – Control - Acción).

DL: Lenguaje de dominio.

GUI: Es una interfaz gráfica, conocida también como GUI (del inglés Graphical User Interface), es un programa informático que actúa de interfaz de usuario, utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz.

WEARABLE: (del inglés wearable device) se conoce como tecnología vestible que describen aquellas prendas de vestir, y complementos que incorporan elementos tecnológicos, electrónicos, etc (Kumar, 2017).

SYMBIAN: fue un sistema operativo propiedad de Nokia, y que en el pasado fue producto de la alianza de varias empresas de telefonía móvil, entre las que se encontraban Nokia, Sony Mobile Communications, Samsung, Siemens, Arima, Lenovo, LG, Motorola, Mitsubishi Electric, Panasonic, Sharp, etc.

SENSOR: Es un objeto cuyo propósito es detectar eventos o cambios en su entorno y envía la información a la computadora que proporcione la salida correspondiente. Un sensor es un dispositivo que convierte los datos del mundo real (analógico).

GEOLOCALIZACIÓN: Entendemos por geolocalización al conjunto de técnicas que permiten determinar la posición geográfica de un elemento (un ordenador, un teléfono móvil o cualquier dispositivo capaz de ser

detectado) en el mundo real y hacer uso de esa información. Esta tecnología requiere de la perfecta sincronización entre hardware y software, es necesario un dispositivo con GPS o conexión a internet y un software que permita hacer uso de ellos en esta dirección (Ruiz del Val, 2025).

IOS: Es un sistema operativo móvil, originalmente desarrollado para iPhone (iPhone OS), después se ha usado en dispositivos de iPod touch y el iPad.

ANDROID: es un sistema operativo basado en el núcleo Linux, fue diseñado principalmente para dispositivos móviles con pantalla táctil, como teléfonos inteligentes, tablets; y también para relojes inteligentes, televisores y automóviles.

SMS: servicio de mensajes cortos o servicio de mensajes simples, más conocido como SMS (por las siglas del inglés Short Message Service), es un servicio disponible en los teléfonos móviles que permite el envío de mensajes cortos, conocidos como mensajes de texto.

GPS: Sistema de Posicionamiento Global, más conocido por sus siglas en inglés, GPS (siglas de Global Positioning System), es un sistema que permite determinar en toda la Tierra la posición de un objeto (una persona, un vehículo) con una precisión de hasta centímetros (si se utiliza GPS diferencial).

JDK: Java Development Kit, es una implementación para las plataformas java: edición estándar, Edición empresarial o Edición micro, realizado por la corporación de Oracle.

LISP: Es una familia de lenguajes de programación de computadora de tipo multiparadigma con una larga historia y una sintaxis completamente entre paréntesis.

Dirección legal: Urb. Paseo del Mar
Nuevo Chimbote, Santa, Ancash
Correo electrónico: ed.honexus@gmail.com
Teléfono: 978653152