Una introducción a la arquitectura de software

David Garlan y Mary Shaw
enero de 1994

CMU-CS-94-166

Escuela de Ciencias de la Computación
Universidad de Carnegie mellon
Pittsburgh, Pensilvania 15213-3890

También publicado como "Introducción a la arquitectura de software", *Avances en ingeniería de software e ingeniería del conocimiento, volumen I,* editado por V. Ambriola y G. Tortora, World Scientific Publishing Company, Nueva Jersey, 1993.

También aparece como CMU Software Engineering Institute Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21.

Este trabajo fue financiado en parte por la Agencia de Proyectos de Investigación Avanzada del Departamento de Defensa bajo la subvención MDA972-92-J-1002, por las Subvenciones de la Fundación Nacional de Ciencias CCR-9109469 y CCR-9112880, y por una subvención de Siemens Corporate Research. También fue financiado en parte por la Escuela de Ciencias de la Computación y el Instituto de Ingeniería de Software de la Universidad Carnegie Mellon (que está patrocinado por el Departamento de Defensa de EE. UU.). Las opiniones y conclusiones contenidas en este documento son las de los autores y no deben interpretarse como representativas de las políticas oficiales, expresas o implícitas, del gobierno de los EE. UU., el Departamento de Defensa, la Fundación Nacional de Ciencias, Siemens Corporation o Carnegie Mellon. Universidad.

Palabras clave: arquitectura de software, diseño de software, ingeniería de software

Resumen

A medida que aumenta el tamaño de los sistemas de software, los algoritmos y las estructuras de datos de la computación ya no constituyen los principales problemas de diseño. Cuando los sistemas se construyen a partir de muchos componentes, la organización del sistema general, la arquitectura del software, presenta un nuevo conjunto de problemas de diseño. Este nivel de diseño se ha abordado de varias maneras, incluidos diagramas informales y términos descriptivos, lenguajes de interconexión de módulos, plantillas y marcos para sistemas que satisfacen las necesidades de dominios específicos y modelos formales de mecanismos de integración de componentes.

En este artículo proporcionamos una introducción al campo emergente de la arquitectura de software. Comenzamos considerando una serie de estilos arquitectónicos comunes en los que se basan actualmente muchos sistemas y mostramos cómo se pueden combinar diferentes estilos en un solo diseño. Luego presentamos seis casos de estudio para ilustrar cómo las representaciones arquitectónicas pueden mejorar nuestra comprensión de los sistemas de software complejos. Finalmente, examinamos algunos de los problemas sobresalientes en el campo y consideramos algunas de las direcciones de investigación prometedoras.

Contenido

Lista de Figuras

1. Introducción

A medida que aumenta el tamaño y la complejidad de los sistemas de software, el problema del diseño va más allá de los algoritmos y las estructuras de datos de la computación: el diseño y la especificación de la estructura general del sistema emerge como un nuevo tipo de problema. Los problemas estructurales incluyen la organización bruta y la estructura de control global; protocolos de comunicación, sincronización y acceso a datos; asignación de funcionalidad a elementos de diseño; distribución física; composición de elementos de diseño; escalado y rendimiento; y selección entre alternativas de diseño.

Este es el nivel de diseño de la arquitectura de software. Existe una cantidad considerable de trabajo sobre este tema, que incluye lenguajes de interconexión de módulos, plantillas y marcos para sistemas que satisfacen las necesidades de dominios específicos y modelos formales de mecanismos de integración de componentes. Además, existe un cuerpo de trabajo implícito en forma de términos descriptivos usados informalmente para describir sistemas. Y aunque actualmente no existe una terminología o notación bien definida para caracterizar las estructuras arquitectónicas, los buenos ingenieros de software hacen un uso común de los principios arquitectónicos cuando diseñan software complejo. Muchos de los principios representan reglas generales o patrones idiomáticos que han surgido de manera informal a lo largo del tiempo. Otros están más cuidadosamente documentados como estándares científicos y de la industria.

Cada vez es más claro que la ingeniería de software efectiva requiere facilidad en el diseño de software arquitectónico. En primer lugar, es importante poder reconocer paradigmas comunes para que se puedan entender las relaciones de alto nivel entre los sistemas y para que se puedan construir nuevos sistemas como variaciones de los sistemas antiguos.
En segundo lugar, obtener la arquitectura correcta suele ser crucial para el éxito del diseño de un sistema de software; el incorrecto puede conducir a resultados desastrosos. En tercer lugar, la comprensión detallada de las arquitecturas de software permite al ingeniero tomar decisiones basadas en principios entre las alternativas de diseño. En cuarto lugar, la representación de un sistema arquitectónico suele ser esencial para el análisis y la descripción de las propiedades de alto nivel de un sistema complejo.

En este artículo proporcionamos una introducción al campo de la arquitectura de software. El propósito es ilustrar el estado actual de la disciplina y examinar las formas en que el diseño arquitectónico puede afectar el diseño de software.
El material presentado aquí se seleccionó de un curso semestral, Arquitecturas para Sistemas de Software, impartido en CMU por los autores [1]. Naturalmente, un documento breve como este solo puede resaltar brevemente las principales características del terreno.
Esta selección enfatiza descripciones informales omitiendo gran parte del material del curso sobre especificación, evaluación y selección entre alternativas de diseño.
Esperamos, sin embargo, que esto sirva para iluminar la naturaleza y el significado de este campo emergente.

En la siguiente sección, describimos una serie de estilos arquitectónicos comunes en los que se basan actualmente muchos sistemas y mostramos cómo

estilos heterogéneos se pueden combinar en un solo diseño. A continuación, utilizamos seis estudios de casos para ilustrar cómo las representaciones arquitectónicas de un sistema de software pueden mejorar nuestra comprensión de los sistemas complejos. Finalmente, examinamos algunos de los problemas sobresalientes en el campo y consideramos algunas de las direcciones de investigación prometedoras.

El texto que constituye la mayor parte de este artículo ha sido extraído de muchas otras publicaciones de los autores. La taxonomía de los estilos arquitectónicos y los estudios de casos han incorporado partes de varios artículos publicados [1, 2, 3, 4]. En menor medida, el material se ha extraído de otros artículos de los autores [5, 6, 7].

2. De los lenguajes de programación a la arquitectura de software

Una caracterización del progreso en lenguajes y herramientas de programación ha sido el aumento regular en el nivel de abstracción, o el tamaño conceptual de los componentes básicos de los diseñadores de software. Para poner el campo de la arquitectura de software en perspectiva, comencemos por observar el desarrollo histórico de las técnicas de abstracción en la informática.

2.1. Lenguajes de programación de alto nivel

Cuando surgieron las computadoras digitales en la década de 1950, el software se escribía en lenguaje de máquina; los programadores colocaron instrucciones y datos de forma individual y explícita en la memoria de la computadora. La inserción de una nueva instrucción en un programa puede requerir la verificación manual de todo el programa para actualizar las referencias a los datos y las instrucciones que se movieron como resultado de la inserción.

Eventualmente, se reconoció que el diseño de la memoria y la actualización de las referencias podrían automatizarse, y también que los nombres simbólicos podrían usarse para códigos de operación y direcciones de memoria. Los ensambladores simbólicos fueron el resultado. Pronto les siguieron los macroprocesadores, que permitían que un solo símbolo representara una secuencia de instrucciones de uso común. La sustitución de códigos de operación de máquina por símbolos simples, direcciones de máquina aún por definir y secuencias de instrucciones fue quizás la forma más temprana de abstracción en el software.

A fines de la década de 1950, quedó claro que ciertos patrones de ejecución eran comúnmente útiles; de hecho, se entendían tan bien que era posible crearlos automáticamente a partir de una notación más parecida a las matemáticas que al lenguaje de máquina. El primero de estos patrones fue para la evaluación de expresiones aritméticas, para la invocación de procedimientos y para bucles y sentencias condicionales. Estas ideas fueron capturadas en una serie de lenguajes tempranos de alto nivel, de los cuales Fortran fue el principal sobreviviente.

Los lenguajes de alto nivel permitieron desarrollar programas más sofisticados y surgieron patrones en el uso de datos. Mientras que en Fortran los tipos de datos servían principalmente como pistas para seleccionar las instrucciones de máquina adecuadas,

Los tipos de datos en Algol y sus sucesores sirven para indicar las intenciones del programador sobre cómo deben usarse los datos. Los compiladores de estos lenguajes podrían aprovechar la experiencia con Fortran y abordar problemas de compilación más sofisticados. Entre otras cosas, verificaron el cumplimiento de estas intenciones, proporcionando así incentivos para que los programadores utilicen el mecanismo de tipos.

El progreso en el diseño del lenguaje continuó con la introducción de módulos para brindar protección a los procedimientos y estructuras de datos relacionados, con la separación de la especificación de un módulo de su implementación y con la introducción de tipos de datos abstractos.

2.2. Tipos de datos abstractos

A fines de la década de 1960, los buenos programadores compartieron una intuición sobre el desarrollo de software: si obtiene las estructuras de datos correctas, el esfuerzo hará que el desarrollo del resto del programa sea mucho más fácil. El trabajo de tipos de datos abstractos de la década de 1970 puede verse como un esfuerzo de desarrollo que convirtió esta intuición en una teoría real. La conversión de una intuición a una teoría implicó la comprensión

- *la estructura del software* (que incluía una representación empaquetada con sus operadores primitivos),
- *especificaciones* (expresadas matemáticamente como modelos abstractos o modelos algebraicos) axiomas),
- *problemas de idioma* (módulos, alcance, tipos definidos por el usuario),
- *integridad del resultado* (invariantes de las estructuras de datos y protección contra otras manipulaciones),
- *reglas para combinar tipos* (declaraciones),
- *ocultación de información* (protección de propiedades no incluidas explícitamente en las especificaciones).

El efecto de este trabajo fue elevar el nivel de diseño de ciertos elementos de los sistemas de software, a saber, los tipos de datos abstractos, por encima del nivel de las declaraciones del lenguaje de programación o algoritmos individuales. Esta forma de abstracción condujo a la comprensión de una buena organización para un módulo completo que sirve para un propósito particular. Esto implicó combinar representaciones, algoritmos, especificaciones e interfaces funcionales de manera uniforme. Por supuesto, se requería cierto apoyo del lenguaje de programación, pero el paradigma de tipos de datos abstractos permitió que algunas partes de los sistemas se desarrollaran a partir de un vocabulario de tipos de datos en lugar de un vocabulario de construcciones del lenguaje de programación.

2.3. Arquitectura de software

Así como los buenos programadores reconocieron estructuras de datos útiles a fines de la década de 1960, los buenos diseñadores de sistemas de software ahora reconocen organizaciones de sistemas útiles.

Uno de ellos se basa en la teoría de los tipos de datos abstractos. Pero esta no es la única manera de organizar un sistema de software.

Muchas otras organizaciones se han desarrollado de manera informal a lo largo del tiempo y ahora forman parte del vocabulario de los diseñadores de sistemas de software. Por ejemplo, las descripciones típicas de las arquitecturas de software incluyen sinopsis como (la cursiva es nuestra):

- "Camelot se basa en el *modelo cliente-servidor* y utiliza llamadas a procedimientos remotos tanto local como remotamente para proporcionar comunicación entre aplicaciones y servidores."[8]
- "*La estratificación de abstracción* y la descomposición del sistema brindan la apariencia de uniformidad del sistema a los clientes, pero permiten que Helix se adapte a una diversidad de dispositivos autónomos. La arquitectura fomenta un *modelo de servidor cliente* para la estructuración de aplicaciones."[9]
- "Hemos elegido un *enfoque distribuido y orientado a objetos* para administrar la información". [10]
- "La forma más fácil de convertir el compilador secuencial canónico en un compilador concurrente es *canalizar* la ejecución de las fases del compilador en varios procesadores. . . . Una forma más efectiva [es] dividir el código fuente en muchos segmentos, que se procesan simultáneamente a través de las diversas fases de compilación [mediante múltiples procesos de compilación] antes de que un pase de fusión final vuelva a combinar el código objeto en un solo programa."[11]

Otras arquitecturas de software están cuidadosamente documentadas y, a menudo, ampliamente difundidas. Los ejemplos incluyen el modelo de referencia de interconexión de sistemas abiertos de la Organización Internacional de Normalización (una arquitectura de red en capas) [12], el modelo de referencia NIST/ECMA (una arquitectura de entorno de ingeniería de software genérica basada en sustratos de comunicación en capas) [13, 14] y X Window System (una arquitectura de interfaz de usuario con ventanas distribuidas basada en activación de eventos y devoluciones de llamadas) [15].

Todavía estamos lejos de tener una taxonomía bien aceptada de tales paradigmas arquitectónicos, y mucho menos una teoría completamente desarrollada de la arquitectura del software. Pero ahora podemos identificar claramente una serie de patrones o estilos arquitectónicos que actualmente forman el repertorio básico de un arquitecto de software.

3. Estilos arquitectónicos comunes

Ahora examinamos algunos de estos estilos arquitectónicos representativos y ampliamente utilizados. Nuestro propósito es ilustrar el rico espacio de opciones arquitectónicas e indicar cuáles son algunas de las ventajas y desventajas de elegir un estilo sobre otro.

Para dar sentido a las diferencias entre estilos, es útil tener un marco común desde el cual verlos. El marco que adoptaremos es tratar una arquitectura de un sistema específico como una colección de computacional

componentes, o simplemente componentes, junto con una descripción de las interacciones entre estos componentes, los *conectores.* Hablando gráficamente, esto conduce a una vista de una descripción arquitectónica abstracta como un gráfico en el que los nodos representan los componentes y los arcos representan los conectores. Como veremos, los conectores pueden representar interacciones tan variadas como llamadas a procedimientos, transmisiones de eventos, consultas a bases de datos y conductos.

Un estilo arquitectónico, entonces, define una familia de tales sistemas en términos de un patrón de organización estructural. Más específicamente, un estilo arquitectónico determina el vocabulario de componentes y conectores que se pueden usar en instancias de ese estilo, junto con un conjunto de *restricciones* sobre cómo se pueden combinar. Estos pueden incluir restricciones topológicas en las descripciones arquitectónicas (por ejemplo, sin ciclos). Otras restricciones, por ejemplo, que tienen que ver con la semántica de ejecución, también pueden ser parte de la definición del estilo.

Dado este marco, podemos entender qué es un estilo al responder las siguientes preguntas: ¿Cuál es el patrón estructural: los componentes, los conectores y las restricciones? ¿Cuál es el modelo computacional subyacente?
¿Cuáles son las invariantes esenciales del estilo? ¿Cuáles son algunos ejemplos comunes de su uso? ¿Cuáles son las ventajas y desventajas de usar ese estilo? ¿Cuáles son algunas de las especializaciones comunes?

3.1. Tuberías y Filtros

En un estilo de tubería y filtro, cada componente tiene un conjunto de entradas y un conjunto de salidas. Un componente lee flujos de datos en sus entradas y produce flujos de datos en sus salidas, entregando una instancia completa del resultado en un orden estándar. Esto generalmente se logra aplicando una transformación local a los flujos de entrada y computando de manera incremental para que la salida comience antes de que se consuma la entrada. Por lo tanto, los componentes se denominan "filtros". Los conectores de este estilo sirven como conductos para los flujos, transmitiendo las salidas de un filtro a las entradas de otro. Por lo tanto, los conectores se denominan "tuberías".

Entre las invariantes importantes del estilo, los filtros deben ser entidades independientes: en particular, no deben compartir estado con otros filtros. Otra invariante importante es que los filtros no conocen la identidad de sus filtros aguas arriba y aguas abajo. Sus especificaciones pueden restringir lo que aparece en las tuberías de entrada o dar garantías sobre lo que aparece en las tuberías de salida, pero es posible que no identifiquen los componentes en los extremos de esas tuberías. Además, la corrección de la salida de una red de tuberías y filtros no debe depender del orden en que los filtros realizan su procesamiento incremental, aunque se puede suponer una programación justa. (Ver [5] para una discusión en profundidad de este estilo y sus propiedades formales.) La figura 1 ilustra este estilo.

Las especializaciones comunes de este estilo incluyen *canalizaciones,* que restringen las topologías a secuencias lineales de filtros; tuberías limitadas, que restringen la cantidad de datos que pueden residir en una tubería; y canalizaciones tipadas, que requieren que los datos pasados entre dos filtros tengan un tipo bien definido.

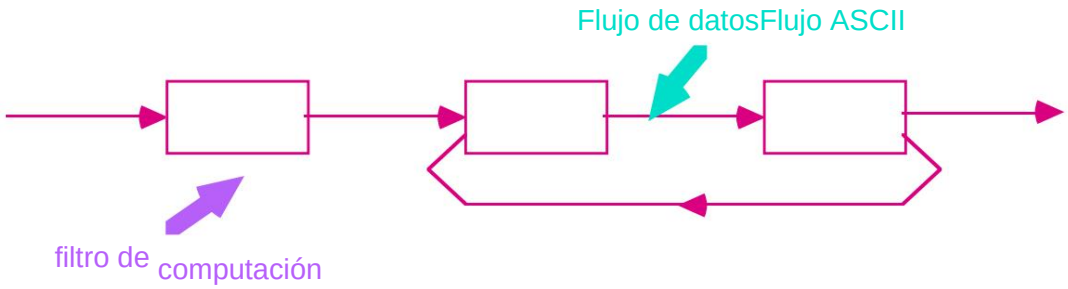

Figura 1: Tuberías y filtros

Un caso degenerado de una arquitectura de canalización ocurre cuando cada filtro procesa todos sus datos de entrada como una sola entidad.1 En este caso, la arquitectura se convierte en un sistema "secuencial por lotes". En estos sistemas, las tuberías ya no cumplen la función de proporcionar un flujo de datos y, por lo tanto, son en gran medida vestigiales.
Por lo tanto, tales sistemas se tratan mejor como instancias de un estilo arquitectónico separado.

Los ejemplos más conocidos de arquitecturas de tuberías y filtros son los programas escritos en el shell de Unix [16]. Unix admite este estilo proporcionando una notación para conectar componentes (representados como procesos de Unix) y proporcionando mecanismos de tiempo de ejecución para implementar conductos. Como otro ejemplo bien conocido, tradicionalmente los compiladores han sido vistos como sistemas de tubería (aunque las fases a menudo no son incrementales). Las etapas en la tubería incluyen análisis léxico, análisis sintáctico, análisis semántico, generación de código. (Volveremos a este ejemplo en los casos de estudio.) Otros ejemplos de conductos y filtros ocurren en dominios de procesamiento de señales [17], programación funcional [18] y sistemas distribuidos [19].

Los sistemas de tuberías y filtros tienen una serie de buenas propiedades. En primer lugar, permiten al diseñador comprender el comportamiento general de entrada/salida de un sistema como una simple composición de los comportamientos de los filtros individuales. En segundo lugar, admiten la reutilización: se pueden conectar dos filtros cualquiera, siempre que coincidan en los datos que se transmiten entre ellos. En tercer lugar, los sistemas se pueden mantener y mejorar fácilmente: se pueden agregar nuevos filtros a los sistemas existentes y los filtros antiguos se pueden reemplazar por otros mejorados. En cuarto lugar, permiten ciertos tipos de análisis especializados, como el análisis de rendimiento y punto muerto. Finalmente, admiten naturalmente la ejecución concurrente. Cada filtro puede implementarse como una tarea separada y potencialmente ejecutarse en paralelo con otros filtros.

Pero estos sistemas también tienen sus desventajas.2 En primer lugar, los sistemas de tuberías y filtros a menudo conducen a una organización del procesamiento por lotes. Aunque los filtros pueden

---

1En general, encontramos que los límites de los estilos pueden superponerse. Esto no debería disuadirnos de identificar las características principales de un estilo con sus ejemplos centrales de uso.
2Esto es cierto a pesar del hecho de que las pipas y los filtros, como todos los estilos, tienen un grupo de seguidores religiosos devotos, personas que creen que todos los problemas que vale la pena resolver pueden resolverse mejor usando ese estilo en particular.

procesar los datos de forma incremental, dado que los filtros son inherentemente independientes, el diseñador se ve obligado a pensar que cada filtro proporciona una transformación completa de los datos de entrada en datos de salida. En particular, debido a su carácter transformacional, los sistemas de tuberías y filtros normalmente no son buenos para manejar aplicaciones interactivas. Este problema es más grave cuando se requieren actualizaciones de visualización incrementales, porque el patrón de salida para las actualizaciones incrementales es radicalmente diferente del patrón para la salida del filtro. En segundo lugar, pueden verse obstaculizados por tener que mantener correspondencias entre dos flujos separados pero relacionados. En tercer lugar, dependiendo de la implementación, pueden forzar una combinación más baja.

denominador común en la transmisión de datos, lo que genera un trabajo adicional para que cada filtro analice y anule el análisis de sus datos. Esto, a su vez, puede provocar una pérdida de rendimiento y una mayor complejidad al escribir los filtros.

3.2. Abstracción de datos y organización orientada a objetos

En este estilo, las representaciones de datos y sus operaciones primitivas asociadas se encapsulan en un objeto o tipo de datos abstracto. Los componentes de este estilo son los objetos o, si se prefiere, instancias de los tipos de datos abstractos. Los objetos son ejemplos de una especie de componente que llamamos administrador porque es responsable de preservar la integridad de un recurso (aquí la representación). Los objetos interactúan a través de invocaciones de funciones y procedimientos. Dos aspectos importantes de este estilo son (a) que un objeto es responsable de preservar la integridad de su representación (generalmente manteniendo alguna invariante sobre él) y (b) que la representación está oculta de otros objetos. La Figura 2 ilustra este estilo.3

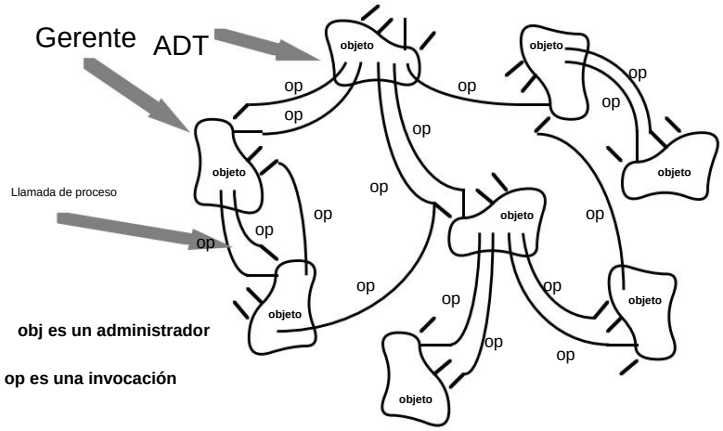

Figura 2: Objetos y tipos de datos abstractos

3No hemos mencionado la herencia en esta descripción. Si bien la herencia es un principio organizador importante para definir los tipos de objetos en un sistema, no tiene una función arquitectónica directa. En particular, desde nuestro punto de vista, una relación de herencia no es un conector, ya que no define la interacción entre los componentes de un sistema. Además, en un entorno arquitectónico, la herencia de propiedades no se limita a los tipos de objetos, sino que puede incluir conectores e incluso estilos arquitectónicos.

El uso de tipos de datos abstractos y, cada vez más, el uso de sistemas orientados a objetos está, por supuesto, muy extendido. Hay muchas variaciones. Por ejemplo, algunos sistemas permiten que los "objetos" sean tareas concurrentes; otros permiten que los objetos tengan múltiples interfaces [20, 21].

Los sistemas orientados a objetos tienen muchas buenas propiedades, la mayoría de las cuales son bien conocidas. Debido a que un objeto oculta su representación a sus clientes, es posible cambiar la implementación sin afectar a esos clientes. Además, la agrupación de un conjunto de rutinas de acceso con los datos que manipulan permite a los diseñadores descomponer los problemas en colecciones de agentes que interactúan.

Pero los sistemas orientados a objetos también tienen algunas desventajas. El más significativo es que para que un objeto interactúe con otro (a través de una llamada a procedimiento) debe conocer la identidad de ese otro objeto. Esto contrasta, por ejemplo, con los sistemas de tuberías y filtros, donde los filtros no necesitan saber qué otros filtros hay en el sistema para interactuar con ellos. La importancia de esto es que cada vez que cambia la identidad de un objeto, es necesario modificar todos los demás objetos que lo invocan explícitamente. En un lenguaje orientado a módulos, esto se manifiesta como la necesidad de cambiar la lista de "importación" de cada módulo que usa el módulo modificado. Además, puede haber problemas de efectos secundarios: si A usa el objeto B y C también usa B, entonces los efectos de C en B parecen efectos secundarios inesperados para A, y viceversa.

3.3. Invocación implícita basada en eventos

Tradicionalmente, en un sistema en el que las interfaces de los componentes proporcionan una colección de procedimientos y funciones, los componentes interactúan entre sí invocando explícitamente esas rutinas. Sin embargo, recientemente ha habido un interés considerable en una técnica de integración alternativa, denominada invocación implícita, integración reactiva y difusión selectiva. Este estilo tiene raíces históricas en los sistemas basados en actores [22], satisfacción de restricciones, demonios y redes de conmutación de paquetes.

La idea detrás de la invocación implícita es que en lugar de invocar un procedimiento directamente, un componente puede anunciar (o transmitir) uno o más eventos. Otros componentes del sistema pueden registrar un interés en un evento asociando un procedimiento con el evento. Cuando se anuncia el evento, el propio sistema invoca todos los procedimientos que se han registrado para el evento. Así, un anuncio de evento "implícitamente" provoca la invocación de procedimientos en otros módulos.

Por ejemplo, en el sistema Field [23], herramientas como editores y monitores de variables se registran para los eventos de punto de interrupción de un depurador. Cuando un depurador se detiene en un punto de interrupción, anuncia un evento que permite que el sistema invoque automáticamente métodos en esas herramientas registradas. Estos métodos pueden desplazar un editor a la línea de origen adecuada o volver a mostrar el valor de las variables monitoreadas. En este esquema, el depurador simplemente anuncia un evento, pero no

No sé qué otras herramientas (si las hay) están relacionadas con ese evento, o qué harán cuando se anuncie ese evento.

Desde el punto de vista arquitectónico, los componentes de un estilo de invocación implícita son módulos cuyas interfaces proporcionan tanto una colección de procedimientos (como con los tipos de datos abstractos) como un conjunto de eventos. Los trámites podrán ser convocados de la forma habitual. Pero además, un componente puede registrar algunos de sus procedimientos con eventos del sistema. Esto hará que estos procedimientos se invoquen cuando esos eventos se anuncien en tiempo de ejecución. Por lo tanto, los conectores en un sistema de invocación implícita incluyen llamadas a procedimientos tradicionales, así como enlaces entre anuncios de eventos y llamadas a procedimientos.

La principal invariante de este estilo es que los anunciantes de eventos no saben qué componentes se verán afectados por esos eventos. Por lo tanto, los componentes no pueden hacer suposiciones sobre el orden de procesamiento, o incluso sobre qué procesamiento ocurrirá como resultado de sus eventos. Por esta razón, la mayoría de los sistemas de invocación implícita también incluyen la invocación explícita (es decir, la llamada a un procedimiento normal) como una forma complementaria de interacción.

Abundan los ejemplos de sistemas con mecanismos de invocación implícitos [7]. Se utilizan en entornos de programación para integrar herramientas [23, 24], en sistemas de gestión de bases de datos para garantizar restricciones de coherencia [22, 25], en interfaces de usuario para separar la presentación de datos de las aplicaciones que gestionan los datos [26, 27] y por editores dirigidos por la sintaxis para admitir la verificación semántica incremental [28, 29].

Un beneficio importante de la invocación implícita es que proporciona un fuerte soporte para la reutilización. Cualquier componente se puede introducir en un sistema simplemente registrándolo para los eventos de ese sistema. Un segundo beneficio es que la invocación implícita facilita la evolución del sistema [30]. Los componentes pueden ser reemplazados por otros componentes sin afectar las interfaces de otros componentes en el sistema.

Por el contrario, en un sistema basado en una invocación explícita, cada vez que se cambia la identidad de un que proporciona alguna función del sistema, todos los demás módulos que importan ese módulo también deben cambiar.

La principal desventaja de la invocación implícita es que los componentes renuncian al control sobre el cálculo realizado por el sistema. Cuando un componente anuncia un evento, no tiene idea de qué otros componentes responderán a él. Peor aún, incluso si sabe qué otros componentes están interesados en los eventos que anuncia, no puede confiar en el orden en que se invocan. Tampoco puede saber cuándo están terminados. Otro problema se refiere al intercambio de datos. A veces, los datos se pueden pasar con el evento. Pero en otras situaciones, los sistemas de eventos deben depender de un repositorio compartido para la interacción. En estos casos, el rendimiento global y la gestión de recursos pueden convertirse en un problema grave. Finalmente, razonar acerca de la corrección puede ser problemático, ya que el significado de un procedimiento que anuncia eventos será

dependen del contexto de los enlaces en los que se invoca. Esto contrasta con el razonamiento tradicional sobre las llamadas a procedimientos, que solo necesitan considerar las condiciones previas y posteriores de un procedimiento al razonar sobre una invocación del mismo.

3.4. Sistemas en capas

Un sistema en capas está organizado jerárquicamente, cada capa brinda servicio a la capa superior y sirve como cliente a la capa inferior. En algunos sistemas en capas, las capas internas están ocultas para todos, excepto para la capa externa adyacente, excepto para ciertas funciones cuidadosamente seleccionadas para exportar. Por lo tanto, en estos sistemas, los componentes implementan una máquina virtual en alguna capa de la jerarquía. (En otros sistemas en capas, las capas pueden ser solo parcialmente opacas). Los conectores están definidos por los protocolos que determinan cómo interactuarán las capas.
Las restricciones topológicas incluyen la limitación de interacciones a capas adyacentes. La Figura 3 ilustra este estilo.

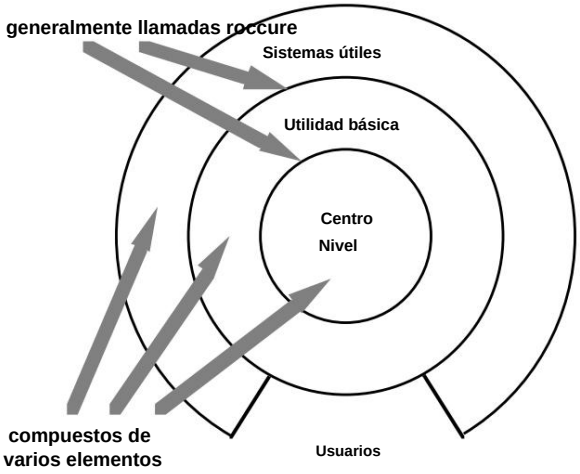

Figura 3: Sistemas en capas

Los ejemplos más conocidos de este tipo de estilo arquitectónico son los protocolos de comunicación en capas [31]. En esta área de aplicación, cada capa proporciona un sustrato para la comunicación en algún nivel de abstracción. Los niveles más bajos definen niveles más bajos de interacción, siendo el más bajo típicamente definido por conexiones de hardware. Otras áreas de aplicación para este estilo incluyen sistemas de bases de datos y sistemas operativos [9, 32, 33].

Los sistemas en capas tienen varias propiedades deseables. Primero, apoyan el diseño basado en niveles crecientes de abstracción. Esto permite a los implementadores dividir un problema complejo en una secuencia de pasos incrementales. En segundo lugar, apoyan la mejora. Al igual que las tuberías, debido a que cada capa interactúa como máximo con las capas inferiores y superiores, los cambios en la función de una capa afectan como máximo a otras dos capas. En tercer lugar, admiten la reutilización. Al igual que los tipos de datos abstractos, las diferentes implementaciones de la misma capa se pueden usar indistintamente, siempre que admitan las mismas interfaces para sus capas adyacentes. Esto lleva a la posibilidad de definir interfaces de capa estándar a las que se

los implementadores pueden construir. (Un buen ejemplo es el modelo OSI ISO y algunos de los protocolos del sistema X Window).

Pero los sistemas en capas también tienen desventajas. No todos los sistemas se estructuran fácilmente en capas. (Veremos un ejemplo de esto más adelante en los estudios de casos). E incluso si un sistema *puede* estructurarse lógicamente como capas, las consideraciones de rendimiento pueden requerir un acoplamiento más estrecho entre las funciones de alto nivel lógico y sus implementaciones de nivel inferior. Además, puede ser bastante difícil encontrar los niveles correctos de abstracción. Esto es particularmente cierto para los modelos en capas estandarizados. Uno observa que la comunidad de comunicaciones ha tenido algunas dificultades para mapear los protocolos existentes en el marco ISO: muchos de esos protocolos unen varias capas.

En cierto sentido, esto es similar a los beneficios de ocultar la implementación que se encuentran en los tipos de datos abstractos. Sin embargo, aquí hay múltiples niveles de abstracción e implementación. También son similares a las canalizaciones, en el sentido de que los componentes se comunican como máximo con otro componente a cada lado. Pero en lugar de un simple protocolo de lectura/escritura de tuberías, los sistemas en capas pueden proporcionar formas de interacción mucho más ricas. Esto dificulta la definición de capas independientes del sistema (como con los filtros), ya que una capa debe admitir protocolos específicos en sus límites superior e inferior. Pero también permite una interacción mucho más cercana entre las capas y permite la transmisión bidireccional de información.

3.5. Repositorios

En un estilo de repositorio, hay dos tipos de componentes bastante distintos: una estructura de datos central representa el estado actual y una colección de componentes independientes que operan en el almacén de datos central. Las interacciones entre el repositorio y sus componentes externos pueden variar significativamente entre sistemas.

La elección de la disciplina de control conduce a subcategorías principales. Si los tipos de transacciones en un flujo de entrada de transacciones activan la selección de procesos para ejecutar, el repositorio puede ser una base de datos tradicional. Si el estado actual de la estructura de datos central es el principal desencadenante de la selección de procesos para ejecutar, el repositorio puede ser una pizarra.

La Figura 4 ilustra una vista simple de una arquitectura de pizarra. (Examinaremos modelos más detallados en los estudios de caso). El modelo de pizarra generalmente se presenta con tres partes principales:

Las fuentes de conocimiento: parcelas separadas e independientes de conocimiento dependiente de la aplicación. La interacción entre fuentes de conocimiento se da únicamente a través de la pizarra.

La estructura de datos de pizarra: datos de estado de resolución de problemas, organizados en una jerarquía dependiente de la aplicación. Las fuentes de conocimiento realizan cambios en la pizarra que conducen gradualmente a una solución al problema.

Control: accionado íntegramente por estado de pizarra. Las fuentes de conocimiento responden de manera oportunista cuando los cambios en la pizarra las hacen aplicables.

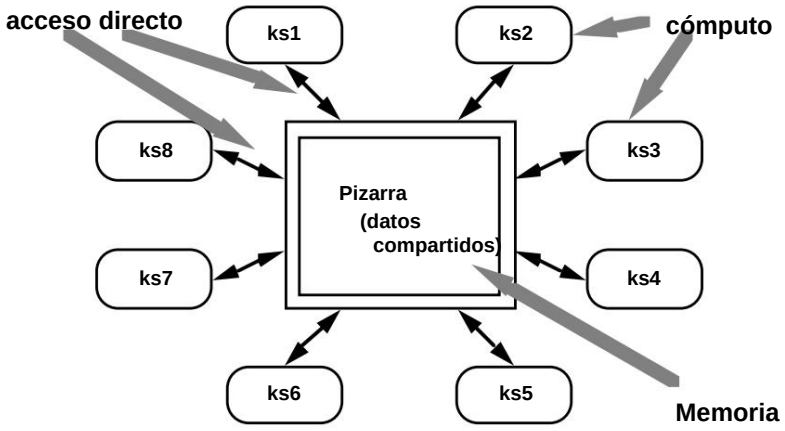

Figura 4: La pizarra

En el diagrama no hay una representación explícita del componente de control. La invocación de una fuente de conocimiento se desencadena por el estado de la pizarra. El lugar de control real y, por lo tanto, su implementación, puede estar en las fuentes de conocimiento, la pizarra, un módulo separado o alguna combinación de estos.

Los sistemas de pizarra se han utilizado tradicionalmente para aplicaciones que requieren interpretaciones complejas del procesamiento de señales, como reconocimiento de voz y patrones. Varios de estos son examinados por Nii [34]. También han aparecido en otros tipos de sistemas que implican acceso compartido a datos con agentes débilmente acoplados [35].

Hay, por supuesto, muchos otros ejemplos de sistemas de depósito. Los sistemas secuenciales por lotes con bases de datos globales son un caso especial. Los entornos de programación a menudo se organizan como una colección de herramientas junto con un repositorio compartido de programas y fragmentos de programas [36]. Incluso las aplicaciones que tradicionalmente se han visto como arquitecturas de canalización, pueden interpretarse con mayor precisión como sistemas de repositorio. Por ejemplo, como veremos más adelante, mientras que la arquitectura de un compilador se ha presentado tradicionalmente como una canalización, las “fases” de la mayoría de los compiladores modernos operan sobre una base de información compartida (tablas de símbolos, árbol de sintaxis abstracta, etc.).

3.6. Intérpretes controlados por tablas

En una organización de intérpretes se produce una máquina virtual en software. Un intérprete incluye el pseudoprograma que se está interpretando y el propio motor de interpretación. El pseudoprograma incluye el propio programa y el análogo del intérprete de su estado de ejecución (registro de activación). El motor de interpretación incluye tanto la definición del intérprete como el estado actual de *su* ejecución. Así, un intérprete generalmente tiene cuatro componentes: un motor de interpretación para hacer el trabajo, una memoria que contiene

el pseudocódigo a interpretar, una representación del estado de control del motor de interpretación y una representación del estado actual del programa que se está simulando. (Consulte la Figura 5.)

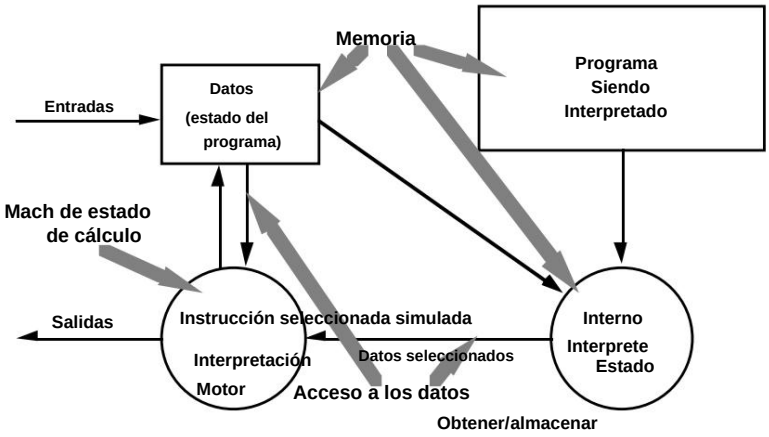

Figura 5: Intérprete

Los intérpretes se usan comúnmente para construir máquinas virtuales que cierran la brecha entre el motor de cómputo esperado por la semántica del programa y el motor de cómputo disponible en el hardware. De vez en cuando hablamos de un lenguaje de programación que proporciona, digamos, una "máquina virtual de Pascal".

Volveremos a los intérpretes con más detalle en los estudios de casos.

3.7. Otras arquitecturas familiares

Hay muchos otros estilos y patrones arquitectónicos. Algunos están muy extendidos y otros son específicos de dominios particulares. Si bien un tratamiento completo de estos está más allá del alcance de este documento, señalamos brevemente algunas de las categorías importantes.

- **Procesos** distribuidos: los sistemas distribuidos han desarrollado una serie de organizaciones comunes para sistemas multiproceso [37]. Algunos se pueden caracterizar principalmente por sus características topológicas, como las organizaciones en anillo y en estrella. Otros se caracterizan mejor en términos de los tipos de protocolos entre procesos que se utilizan para la comunicación (por ejemplo, algoritmos de latido).

Una forma común de arquitectura de sistema distribuido es una organización de “cliente-servidor” [38]. En estos sistemas un servidor representa un proceso que brinda servicios a otros procesos (los clientes). Por lo general, el servidor no conoce de antemano las identidades o la cantidad de clientes que accederán a él en tiempo de ejecución. Por otro lado, los clientes conocen la identidad de un servidor (o pueden averiguarlo a través de algún otro servidor) y acceden a él mediante una llamada a procedimiento remoto.

- **Organizaciones principales de programas/subrutinas:** la organización principal de muchos sistemas refleja el lenguaje de programación en el que se ejecuta el sistema.

está escrito. Para lenguajes sin soporte para modularización, esto a menudo resulta en un sistema organizado alrededor de un programa principal y un conjunto de subrutinas. El programa principal actúa como controlador de las subrutinas y, por lo general, proporciona un bucle de control para secuenciar las subrutinas en algún orden.

- **Arquitecturas de software de dominio específico:** Recientemente ha habido un interés considerable en el desarrollo de arquitecturas de "referencia" para dominios específicos [39]. Estas arquitecturas proporcionan una estructura organizativa adaptada a una familia de aplicaciones, como aviónica, mando y control o sistemas de gestión de vehículos. Al especializar la arquitectura al dominio, es posible aumentar el poder descriptivo de las estructuras. De hecho, en muchos casos la arquitectura está lo suficientemente restringida como para generar un sistema ejecutable de forma automática o semiautomática a partir de la propia descripción de la arquitectura.

- **Sistemas de transición de estado:** una organización común para muchos sistemas reactivos es el sistema de transición de estado [40]. Estos sistemas se definen en términos de un conjunto de estados y un conjunto de transiciones con nombre que mueven un sistema de un estado a otro.

- **Sistemas de control de procesos:** Los sistemas destinados a proporcionar un control dinámico de un entorno físico a menudo se organizan como sistemas de control de procesos [41]. Estos sistemas se caracterizan aproximadamente como un circuito de retroalimentación en el que el sistema de control de procesos utiliza las entradas de los sensores para determinar un conjunto de salidas que producirán un nuevo estado del entorno.

3.8. Arquitecturas heterogéneas

Hasta ahora hemos estado hablando principalmente de estilos arquitectónicos “puros”. Si bien es importante comprender la naturaleza individual de cada uno de estos estilos, la mayoría de los sistemas generalmente involucran una combinación de varios estilos.

Hay diferentes formas de combinar los estilos arquitectónicos. Una forma es a través de la jerarquía. Un componente de un sistema organizado en un estilo arquitectónico puede tener una estructura interna desarrollada en un estilo completamente diferente. Por ejemplo, en una canalización de Unix, los componentes individuales pueden representarse internamente utilizando prácticamente cualquier estilo, incluido, por supuesto, otro sistema de canalización y filtro.

Lo que quizás sea más sorprendente es que los conectores también pueden descomponerse jerárquicamente. Por ejemplo, un conector de tubería puede implementarse internamente como una cola FIFO a la que se accede mediante operaciones de inserción y eliminación.

Una segunda forma de combinar estilos es permitir que un solo componente use una combinación de conectores arquitectónicos. Por ejemplo, un componente puede acceder a un repositorio a través de parte de su interfaz, pero interactuar a través de conductos con otros componentes en un sistema y aceptar información de control a través de

otra parte de su interfaz. (De hecho, los sistemas de canalización y filtro de Unix hacen esto, el sistema de archivos desempeña el papel de repositorio y los interruptores de inicialización desempeñan el papel de control).

Otro ejemplo es una “base de datos activa”. Este es un repositorio que activa componentes externos a través de una invocación implícita. En esta organización, los componentes externos registran interés en porciones de la base de datos.
La base de datos invoca automáticamente las herramientas adecuadas en función de esta asociación. (Las pizarras a menudo se construyen de esta manera; las fuentes de conocimiento se asocian con tipos específicos de datos y se activan cada vez que se modifica ese tipo de datos).

Una tercera forma de combinar estilos es elaborar completamente un nivel de descripción arquitectónica en un estilo arquitectónico completamente diferente. Veremos ejemplos de esto en los casos de estudio.

4. Estudios de casos

Ahora presentamos seis ejemplos para ilustrar cómo se pueden usar los principios arquitectónicos para aumentar nuestra comprensión de los sistemas de software. El primer ejemplo muestra cómo diferentes soluciones arquitectónicas para el mismo problema brindan diferentes beneficios. El segundo estudio de caso resume la experiencia en el desarrollo de un estilo arquitectónico de dominio específico para una familia de productos industriales. El tercer caso de estudio examina la arquitectura familiar del compilador bajo una nueva luz.
Los tres estudios de caso restantes presentan ejemplos del uso de arquitecturas heterogéneas.

4.1. Estudio de caso 1: palabra clave en contexto

En su artículo de 1972, Parnas propuso el siguiente problema [42]:

> El sistema de índice KWIC [Palabra clave en contexto] acepta un conjunto ordenado de líneas, cada línea es un conjunto ordenado de palabras y cada palabra es un conjunto ordenado de caracteres. Cualquier línea se puede "desplazar circularmente" eliminando repetidamente la primera palabra y agregándola al final de la línea. El sistema de índice KWIC genera una lista de todos los turnos circulares de todas las líneas en orden alfabético.

Parnas usó el problema para contrastar diferentes criterios para descomponer un sistema en módulos. Describe dos soluciones, una basada en la descomposición funcional con acceso compartido a las representaciones de datos y una segunda basada en una descomposición que oculta las decisiones de diseño. Desde su introducción, el problema se ha vuelto muy conocido y es ampliamente utilizado como dispositivo de enseñanza en ingeniería de software. Garlan, Kaiser y Notkin también utilizan el problema para ilustrar esquemas de modularización basados en la invocación implícita [7].

Si bien KWIC se puede implementar como un sistema relativamente pequeño, no es simplemente de interés pedagógico. Ejemplos prácticos de ello son ampliamente utilizados por

científicos de la computación. Por ejemplo, el índice "permutado" [sic] para las páginas Man de Unix es esencialmente un sistema de este tipo.

Desde el punto de vista de la arquitectura del software, el problema deriva su atractivo del hecho de que puede usarse para ilustrar el efecto de los cambios en el diseño del software. Parnas muestra que las diferentes descomposiciones de problemas varían mucho en su capacidad para soportar cambios de diseño. Entre los cambios que considera están:

- Cambios en el algoritmo de procesamiento: por ejemplo, el cambio de línea se puede realizar en cada línea a medida que se lee desde el dispositivo de entrada, en todas las líneas después de leerlas o bajo demanda cuando la alfabetización requiere un nuevo conjunto de líneas cambiadas.
- Cambios en la representación de datos: por ejemplo, las líneas se pueden almacenar de varias maneras. De manera similar, los cambios circulares se pueden almacenar explícita o implícitamente (como pares de índice y desplazamiento).

Garlan, Kaiser y Notkin amplían el análisis de Parnas considerando:

- Mejora de la función del sistema: por ejemplo, modifique el sistema para que las líneas desplazadas eliminen los desplazamientos circulares que comienzan con ciertas palabras irrelevantes (como “a”, “an”, “y”, etc.). Cambie el sistema para que sea interactivo y permita que el usuario elimine líneas de las listas originales (o, alternativamente, de las listas desplazadas circularmente).
- Performance: Tanto en el espacio como en el tiempo.
- Reutilización: en qué medida los componentes pueden servir como entidades reutilizables.

Ahora describimos cuatro diseños arquitectónicos para el sistema KWIC. Los cuatro se basan en soluciones publicadas (incluidas las implementaciones). Los dos primeros son los considerados en el artículo original de Parnas. La tercera solución se basa en el uso de un estilo de invocación implícito y representa una variante de la solución examinada por Garlan, Kaiser y Notkin. La cuarta es una solución de tubería inspirada en la utilidad de índice de Unix.

Después de presentar cada solución y resumir brevemente sus fortalezas y debilidades, contrastamos las diferentes descomposiciones arquitectónicas en una tabla organizada a lo largo de las cinco dimensiones de diseño enumeradas anteriormente.

Solución 1: programa principal/ subrutina con datos compartidos

La primera solución descompone el problema de acuerdo con las cuatro funciones básicas realizadas: entrada, desplazamiento, alfabetización y salida. Estos componentes computacionales están coordinados como subrutinas por un programa principal que los secuencia a su vez. Los datos se comunican entre los componentes a través del almacenamiento compartido ("almacenamiento central"). La comunicación entre los componentes computacionales y los datos compartidos es una lectura sin restricciones.

escribir protocolo. Esto es posible gracias a que el programa coordinador garantiza el acceso secuencial a los datos. (Consulte la Figura 6.)

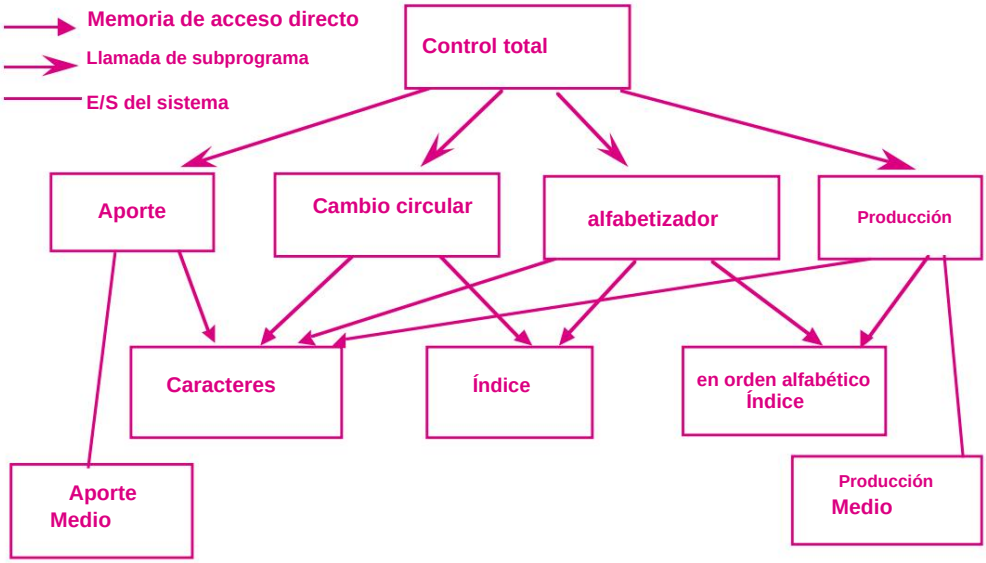

Figura 6: KWIC: solución de datos compartidos

Con esta solución, los datos se pueden representar de manera eficiente, ya que los cálculos pueden compartir el mismo almacenamiento. La solución también tiene cierto atractivo intuitivo, ya que distintos aspectos computacionales están aislados en diferentes módulos.

Sin embargo, como argumenta Parnas, tiene una serie de inconvenientes serios en términos de su capacidad para manejar cambios. En particular, un cambio en el formato de almacenamiento de datos afectará a casi todos los módulos. De manera similar, los cambios en el algoritmo de procesamiento general y las mejoras en la función del sistema no se adaptan fácilmente. Finalmente, esta descomposición no es particularmente favorable a
reutilizar.

Solución 2: tipos de datos abstractos

La segunda solución descompone el sistema en un conjunto similar de cinco módulos.
Sin embargo, en este caso los datos ya no son compartidos directamente por los componentes computacionales. En cambio, cada módulo proporciona una interfaz que permite que otros componentes accedan a los datos solo mediante la invocación de procedimientos en esa interfaz. (Consulte la Figura 7, que ilustra cómo cada uno de los componentes ahora tiene un conjunto de procedimientos que determinan la forma de acceso de otros componentes en el sistema).

Esta solución proporciona la misma descomposición lógica en módulos de procesamiento que la primera. Sin embargo, tiene una serie de ventajas sobre la primera solución cuando se consideran cambios de diseño. En particular, tanto los algoritmos como las representaciones de datos se pueden cambiar en módulos individuales sin afectar a otros. Además, la reutilización está mejor soportada que en la primera solución.

porque los módulos hacen menos suposiciones sobre los demás con los que interactúan.

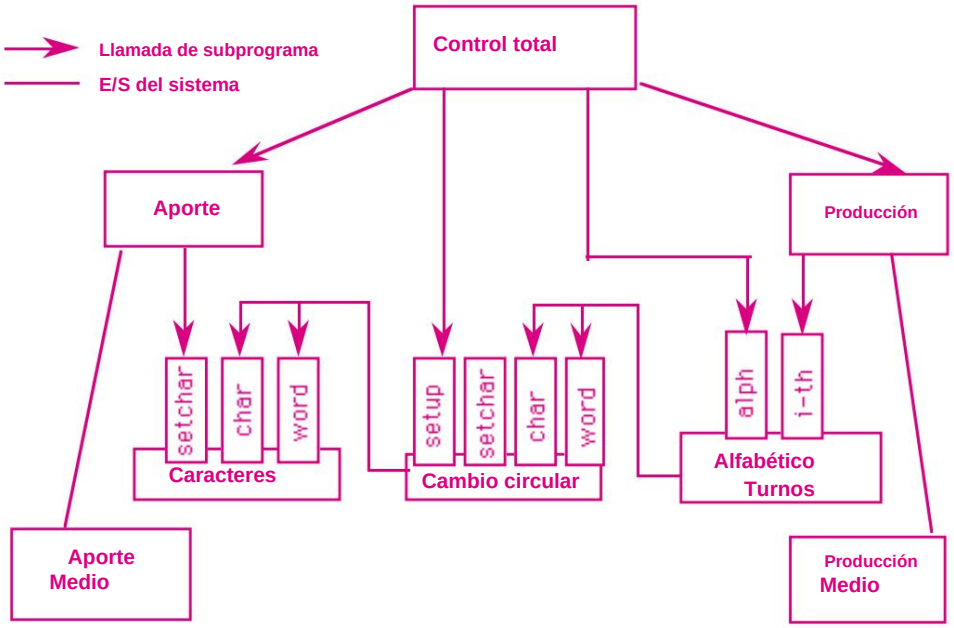

Figura 7: KWIC: solución de tipos de datos abstractos

Por otro lado, como comentan Garlan, Kaiser y Notkin, la solución no es particularmente adecuada para las mejoras. El principal problema es que para agregar nuevas funciones al sistema, el implementador debe modificar los módulos existentes, comprometiendo su simplicidad e integridad, o agregar nuevos módulos que conducen a penalizaciones en el rendimiento. (Ver [7] para una discusión detallada.)

Solución 3: Invocación implícita

La tercera solución utiliza una forma de integración de componentes basada en datos compartidos similar a la primera solución. Sin embargo, hay dos diferencias importantes. Primero, la interfaz con los datos es más abstracta. En lugar de exponer los formatos de almacenamiento a los módulos informáticos, se accede a los datos de forma abstracta (por ejemplo, como una lista o un conjunto). En segundo lugar, los cálculos se invocan implícitamente a medida que se modifican los datos. Por lo tanto, la interacción se basa en un modelo de datos activo. Por ejemplo, el acto de agregar una nueva línea al almacenamiento de línea hace que se envíe un evento al módulo de turno. Esto le permite producir turnos circulares (en un almacén de datos compartido abstracto separado). Esto, a su vez, hace que el alfabetizador se invoque implícitamente para que pueda alfabetizar las líneas.

Esta solución admite fácilmente mejoras funcionales en el sistema: se pueden adjuntar módulos adicionales al sistema registrándolos para que se invoquen en eventos de cambio de datos. Debido a que se accede a los datos de forma abstracta, también aísla los cálculos de los cambios en la representación de los datos. También se admite la reutilización, ya que los módulos invocados implícitamente solo se basan en la existencia de ciertos eventos activados externamente.

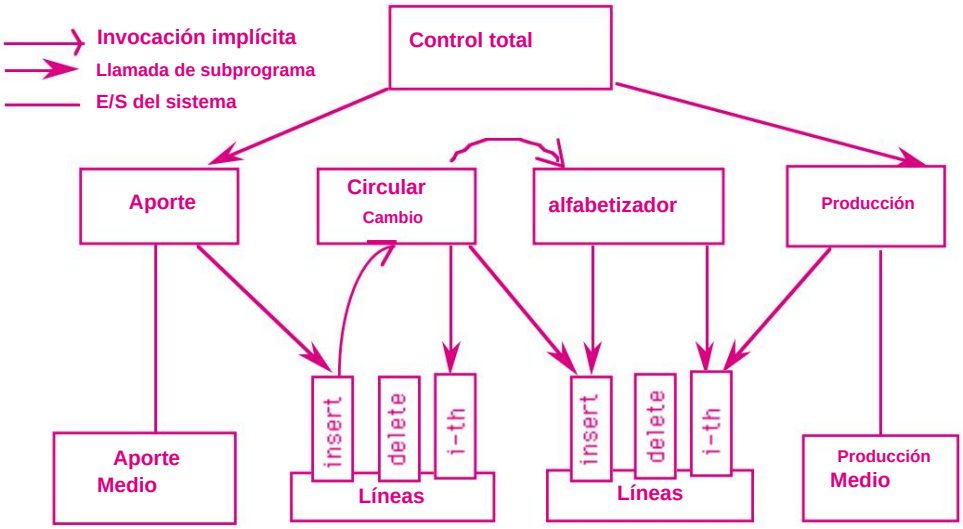

Figura 8: KWIC: solución de invocación implícita

Sin embargo, la solución adolece del hecho de que puede ser difícil controlar el orden de procesamiento de los módulos invocados implícitamente. Además, debido a que las invocaciones están basadas en datos, las implementaciones más naturales de este tipo de descomposición tienden a usar más espacio que las descomposiciones consideradas anteriormente.

Solución 4: Tuberías y Filtros

La cuarta solución utiliza una solución de canalización. En este caso hay cuatro filtros: entrada, desplazamiento, orden alfabético y salida. Cada filtro procesa los datos y los envía al siguiente filtro. El control está distribuido: cada filtro puede ejecutarse siempre que tenga datos sobre los cuales calcular. El intercambio de datos entre filtros está estrictamente limitado a los que se transmiten en las tuberías. (Consulte la Figura 9.)

Esta solución tiene varias propiedades agradables. En primer lugar, mantiene el flujo intuitivo de procesamiento. En segundo lugar, admite la reutilización, ya que cada filtro puede funcionar de forma aislada (siempre que los filtros ascendentes produzcan datos en la forma esperada). Las nuevas funciones se agregan fácilmente al sistema mediante la inserción de filtros en el punto adecuado de la secuencia de procesamiento. En tercer lugar, admite la facilidad de modificación, ya que los filtros son lógicamente independientes de otros filtros.

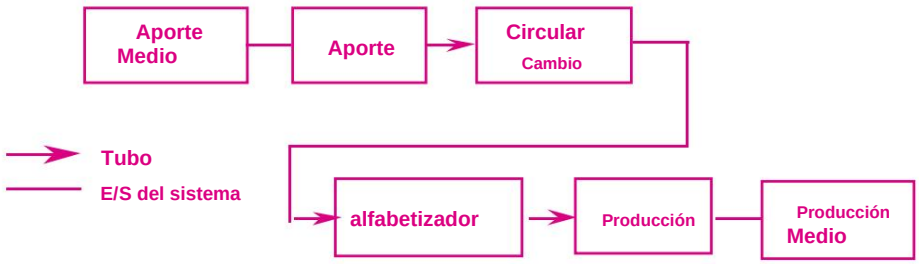

Figura 9: KWIC: solución de tubería y filtro

Por otro lado, tiene una serie de inconvenientes. Primero, es virtualmente imposible modificar el diseño para soportar un sistema interactivo. Por ejemplo, para eliminar una línea, tendría que haber algún almacenamiento compartido persistente, violando un principio básico de este enfoque. En segundo lugar, la solución es ineficiente en términos de uso de espacio, ya que cada filtro debe copiar todos los datos a sus puertos de salida.

comparaciones

Las soluciones se pueden comparar tabulando su capacidad para abordar las consideraciones de diseño detalladas anteriormente. Una comparación detallada tendría que implicar la consideración de una serie de factores relacionados con el uso previsto del sistema: por ejemplo, si es por lotes o interactivo, intensivo en actualizaciones o consultas, etc.

La Figura 10 proporciona una aproximación a dicho análisis, basada en la discusión de los estilos arquitectónicos presentada anteriormente. Como señaló Parnas, la solución de datos compartidos es particularmente débil en su compatibilidad con los cambios en el algoritmo de procesamiento general, las representaciones de datos y la reutilización. Por otro lado, puede lograr un rendimiento relativamente bueno, en virtud del intercambio directo de datos. Además, es relativamente fácil agregar un nuevo componente de procesamiento (también accediendo a los datos compartidos). La solución de tipos de datos abstractos permite cambios en la representación de datos y admite la reutilización, sin comprometer necesariamente el rendimiento. Sin embargo, las interacciones entre los componentes de esa solución están conectadas a los propios módulos, por lo que cambiar el algoritmo de procesamiento general o agregar nuevas funciones puede implicar una gran cantidad de cambios en el sistema existente.

Figura 10: KWIC – Comparación de Soluciones

La solución de invocación implícita es particularmente buena para agregar nuevas funciones. Sin embargo, adolece de algunos de los problemas del enfoque de datos compartidos: soporte deficiente para el cambio en la representación y reutilización de datos. Además, puede introducir una sobrecarga de ejecución adicional. La solución de tubería y filtro permite colocar nuevos filtros en el flujo de procesamiento de texto.
Por lo tanto, admite cambios en el algoritmo de procesamiento, cambios en la función,

y reutilizar. Por otro lado, las decisiones sobre la representación de datos estarán conectadas con las suposiciones sobre el tipo de datos que se transmiten a lo largo de las tuberías. Además, según el formato de intercambio, puede haber una sobrecarga adicional involucrada en el análisis y desanálisis de los datos en las canalizaciones.

4.2. Estudio de caso 2: software de instrumentación

Nuestro segundo estudio de caso describe el desarrollo industrial de una arquitectura de software en Tektronix, Inc. Este trabajo se llevó a cabo como un esfuerzo de colaboración entre varias divisiones de productos de Tektronix y el Computer Research Laboratory durante un período de tres años [6].

El propósito del proyecto era desarrollar una arquitectura de sistema reutilizable para osciloscopios. Un osciloscopio es un sistema de instrumentación que muestrea señales eléctricas y muestra imágenes (llamadas trazas) de ellas en una pantalla. Además, los osciloscopios realizan mediciones en las señales y también las muestran en la pantalla. Mientras que los osciloscopios alguna vez fueron simples dispositivos analógicos que requerían poco software, los osciloscopios modernos se basan principalmente en tecnología digital y tienen un software bastante complejo. No es raro que un osciloscopio moderno realice docenas de mediciones, suministre megabytes de almacenamiento interno, se conecte a una red de estaciones de trabajo y otros instrumentos y proporcione una interfaz de usuario sofisticada que incluye una pantalla táctil con menús, funciones de ayuda integradas y pantallas a color

Al igual que muchas empresas que han tenido que depender cada vez más del software para respaldar sus productos, Tektronix se enfrentó a una serie de problemas. En primer lugar, hubo poca reutilización en diferentes productos de osciloscopio. En cambio, diferentes divisiones de productos construyeron diferentes osciloscopios, cada uno con sus propias convenciones de desarrollo, organización de software, lenguaje de programación y herramientas de desarrollo. Además, incluso dentro de una sola división de productos, cada osciloscopio nuevo generalmente requería un rediseño desde cero para adaptarse a los cambios en la capacidad del hardware y los nuevos requisitos en la interfaz de usuario Este problema se vio agravado por el hecho de que tanto los requisitos de hardware como de interfaz cambiaban cada vez más rápidamente. Además, se percibía la necesidad de abordar los “mercados especializados”. Para hacer esto, tendría que ser posible adaptar un instrumento de uso general a un conjunto específico de usos.

En segundo lugar, había cada vez más problemas de rendimiento porque el software no se podía configurar rápidamente dentro del instrumento. Este problema surge porque un osciloscopio se puede configurar en muchos modos diferentes, dependiendo de la tarea del usuario. En los osciloscopios antiguos, la reconfiguración se manejaba simplemente cargando un software diferente para manejar el nuevo modo. Pero a medida que aumentaba el tamaño total del software, esto generaba demoras entre la solicitud de un usuario de un nuevo modo y un instrumento reconfigurado.

El objetivo del proyecto era desarrollar un marco arquitectónico para osciloscopios que abordara estos problemas. El resultado de ese trabajo fue una arquitectura de software de dominio específico que constituyó la base del siguiente

generación de osciloscopios Tektronix. Desde entonces, el marco se ha ampliado y adaptado para dar cabida a una clase de sistema más amplia y, al mismo tiempo, se ha adaptado mejor a las necesidades específicas del software de instrumentación.

En el resto de esta sección, describimos las etapas de este desarrollo arquitectónico.

Un modelo orientado a objetos

El primer intento de desarrollar una arquitectura reutilizable se centró en producir un modelo orientado a objetos del dominio del software. Esto condujo a una aclaración de los tipos de datos utilizados en los osciloscopios: formas de onda, señales, medidas, modos de disparo, etc. (Consulte la Figura 11).

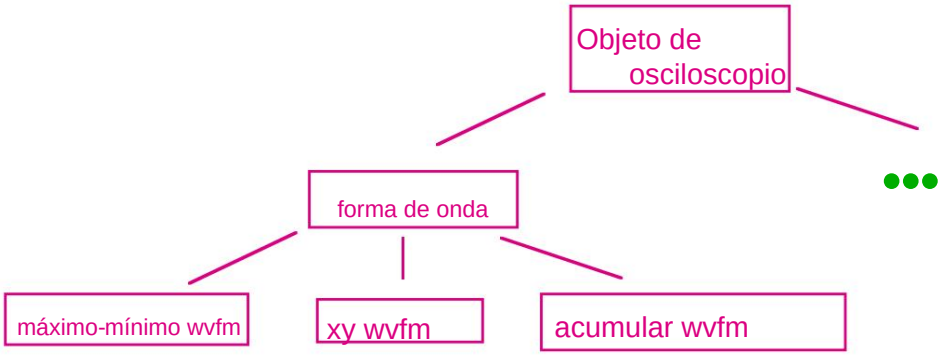

Figura 11: Osciloscopios: un modelo orientado a objetos

Si bien este fue un ejercicio útil, estuvo lejos de producir los resultados esperados. Aunque se identificaron muchos tipos de datos, no había un modelo general que explicara cómo encajaban los tipos. Esto llevó a la confusión sobre la partición de la funcionalidad. Por ejemplo, ¿las mediciones deben asociarse con los tipos de datos que se miden o deben representarse externamente?
¿Con qué objetos debería hablar la interfaz de usuario?

Un modelo en capas

La segunda fase intentó corregir estos problemas proporcionando un modelo en capas de un osciloscopio. (Consulte la Figura 11). En este modelo, la capa central representaba las funciones de manipulación de señales que filtran las señales a medida que ingresan al osciloscopio. Estas funciones normalmente se implementan en hardware. La siguiente capa representaba la adquisición de formas de onda. Dentro de esta capa, las señales se digitalizan y almacenan internamente para su posterior procesamiento. La tercera capa consistía en la manipulación de formas de onda, incluida la medición, la adición de formas de onda, la transformación de Fourier, etc. La cuarta capa consistía en funciones de visualización.
Esta capa fue responsable de mapear formas de onda y medidas digitalizadas a representaciones visuales. La capa más externa era la interfaz de usuario. Esta capa se encargaba de interactuar con el usuario y de decidir qué datos mostrar en pantalla. (Consulte la Figura 12.)

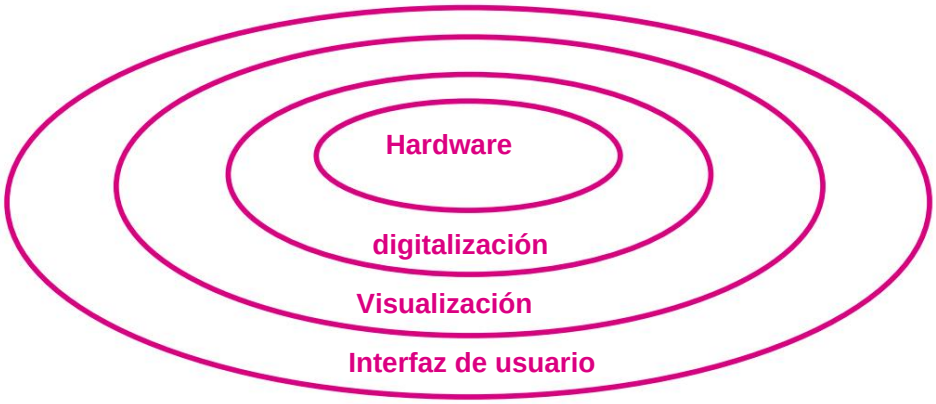

Figura 12: Osciloscopios: un modelo en capas

Este modelo en capas era intuitivamente atractivo ya que dividía las funciones de un osciloscopio en agrupaciones bien definidas. Desafortunadamente, era el modelo incorrecto para el dominio de la aplicación. El principal problema era que los límites de abstracción impuestos por las capas entraban en conflicto con las necesidades de interacción entre las diversas funciones. Por ejemplo, el modelo sugiere que todas las interacciones del usuario con un osciloscopio deben ser en términos de representaciones visuales. Pero en la práctica, los usuarios reales de osciloscopios necesitan afectar directamente las funciones en todas las capas, como establecer la atenuación en la capa de manipulación de señales, elegir el modo de adquisición y los parámetros en la capa de adquisición o crear formas de onda derivadas en la capa de manipulación de formas de onda.

Un modelo de tubería y filtro

El tercer intento produjo un modelo en el que las funciones del osciloscopio se veían como transformadores incrementales de datos. Los transformadores de señal sirven para acondicionar señales externas. Los transformadores de adquisición obtienen formas de onda digitalizadas a partir de estas señales. Los transformadores de visualización convierten estas formas de onda en datos visuales. (Consulte la Figura 13.)

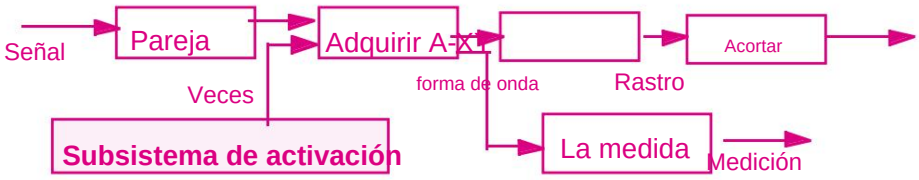

Figura 13: Osciloscopios: un modelo de tubería y filtro

Este modelo arquitectónico fue una mejora significativa sobre el modelo en capas, ya que no aisló las funciones en particiones separadas. Por ejemplo, nada en este modelo evitaría que los datos de la señal se introduzcan directamente en los filtros de visualización. Además, el modelo se correspondía bien con la visión de los ingenieros del procesamiento de señales como un problema de flujo de datos. El principal problema del modelo era que no estaba claro cómo debía interactuar el usuario con él. Si el usuario estuviera simplemente en un extremo del sistema, esto representaría una descomposición aún peor que el sistema en capas.

Un modelo modificado de tubería y filtro

La cuarta solución tuvo en cuenta las entradas del usuario al asociar con cada filtro una interfaz de control que permite que una entidad externa establezca parámetros de operación para el filtro. Por ejemplo, el filtro de adquisición puede tener parámetros que determinen la frecuencia de muestreo y la duración de la forma de onda. Estas entradas sirven como parámetros de configuración para el osciloscopio. Formalmente, los filtros se pueden modelar como funciones de "orden superior", para las cuales los parámetros de configuración determinan qué transformación de datos realizará el filtro. (Ver [17] para esta interpretación de la arquitectura.) La Figura 14 ilustra esta arquitectura.

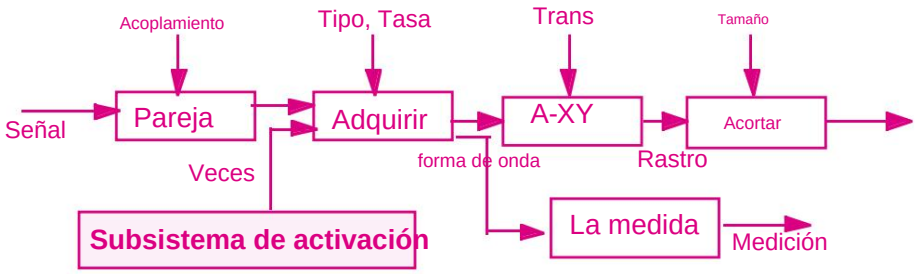

Figura 14: Osciloscopios: un modelo modificado de tubería y filtro

La introducción de una interfaz de control resuelve gran parte del problema de la interfaz de usuario. Primero, proporciona una colección de configuraciones que determinan qué aspectos del osciloscopio pueden ser modificados dinámicamente por el usuario. También explica cómo se pueden lograr cambios en la función del osciloscopio mediante ajustes incrementales en el software. En segundo lugar, desacopla las funciones de procesamiento de señales del osciloscopio de la interfaz de usuario real: el software de procesamiento de señales no hace suposiciones sobre cómo el usuario realmente comunica los cambios en sus parámetros de control. Por el contrario, la interfaz de usuario real puede tratar las funciones de procesamiento de señales únicamente en términos de parámetros de control. Esto permitió a los diseñadores cambiar la implementación del software y el hardware de procesamiento de señales sin afectar la interfaz, siempre que la interfaz de control permaneciera sin cambios.

Especialización adicional

El modelo de tubería y filtro adaptado fue una gran mejora. Pero también tenía algunos problemas. El problema más significativo fue que el modelo computacional de tubería y filtro condujo a un rendimiento deficiente. En particular, las formas de onda pueden ocupar una gran cantidad de almacenamiento interno: simplemente no es práctico que cada filtro copie las formas de onda cada vez que las procesa. Además, diferentes filtros pueden funcionar a velocidades radicalmente diferentes: es inaceptable reducir la velocidad de un filtro porque otro filtro todavía está procesando sus datos.

Para manejar estos problemas, el modelo se especializó aún más. En lugar de tener un solo tipo de tubería, se introdujeron varios "colores" de tuberías. Algunos de estos permitían procesar los datos sin copiarlos. Otros permitieron que los filtros lentos ignoraran los datos si estaban procesando otros datos.
Estos tubos adicionales aumentaron el vocabulario estilístico y permitieron que la

cálculos de tubería/filtro para adaptarse más directamente a las necesidades de rendimiento del producto.

Resumen

Este estudio de caso ilustra los problemas involucrados en el desarrollo de una arquitectura estilo para un dominio de aplicación real. Subraya el hecho de que diferentes estilos arquitectónicos tienen diferentes efectos sobre la capacidad de resolver un conjunto de problemas. Además, ilustra que los diseños arquitectónicos para la industria El software generalmente debe adaptarse de formas puras a estilos especializados que satisfacer las necesidades del dominio específico. En este caso, el resultado final dependía mucho en las propiedades de las arquitecturas de tuberías y filtros, pero encontró formas de adaptar ese estilo genérico para que también pueda satisfacer las necesidades de rendimiento del familia de productos.

4.3. Caso 3: una nueva visión de los compiladores

La arquitectura de un sistema puede cambiar en respuesta a las mejoras en tecnología. Esto se puede ver en la forma en que pensamos acerca de los compiladores.

En la década de 1970, la compilación se consideraba un proceso secuencial y la La organización de un compilador se dibujó típicamente como en la Figura 15. El texto ingresa en el extremo izquierdo y se transforma en una variedad de formas: en flujo de fichas léxicas, árbol de análisis, código intermedio, antes de emerger como código de máquina a la derecha. A menudo nos referimos a este modelo de compilación como una canalización, aunque era (en menos originalmente) más cerca de una arquitectura secuencial por lotes en la que cada la transformación ("pasar") se completó antes de que comenzara la siguiente.

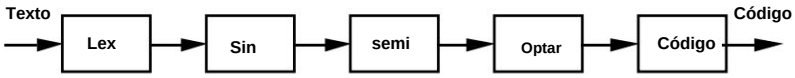

Figura 15: Modelo de compilador tradicional

De hecho, incluso la versión secuencial por lotes de este modelo no fue completamente preciso. La mayoría de los compiladores crearon una tabla de símbolos separada durante el análisis léxico y lo usó o actualizó durante pases posteriores. No era parte de los datos. que fluía de un paso a otro sino que más bien existía fuera de todos los pasos. Entonces, la estructura del sistema se dibujó más correctamente como en la Figura 16.

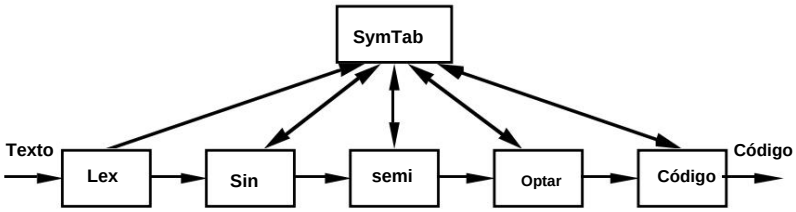

Figura 16: Modelo de compilador tradicional con tabla de símbolos compartidos

Con el paso del tiempo, la tecnología de compilación se volvió más sofisticada. los los algoritmos y las representaciones de la compilación se hicieron más complejos, y atención creciente dirigida a la representación intermedia del programa

durante la compilación. La comprensión teórica mejorada, como los gramáticos de atributos, aceleró esta tendencia. La consecuencia fue que, a mediados de la década de 1980, la representación intermedia (por ejemplo, un árbol de análisis sintáctico atribuido) era el centro de atención. Fue creado temprano durante la compilación y manipulado durante el resto; la estructura de datos podría cambiar en detalle, pero se mantuvo sustancialmente como una estructura en crecimiento en todo momento. Sin embargo, continuamos (a veces hasta el presente) modelando el compilador con flujo de datos secuencial como en la Figura 17.

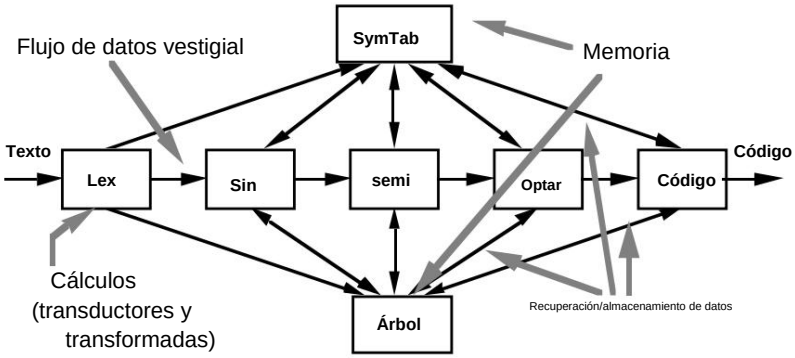

Figura 17: Compilador canónico moderno

De hecho, una visión más adecuada de esta estructura redirigiría la atención de la secuencia de pases a la representación central compartida. Cuando declara que el árbol es el lugar de la información de compilación y los pases definen operaciones en el árbol, se vuelve natural volver a dibujar la arquitectura como en la Figura 18. Ahora las conexiones entre los pases denotan el flujo de control, que es una representación más precisa ; las conexiones más fuertes entre los pases y la estructura de tabla de árbol/ símbolo denotan acceso y manipulación de datos. De esta manera, la arquitectura se ha convertido en un repositorio y, de hecho, esa es una forma más apropiada de pensar en un compilador de esta clase.

Afortunadamente, esta nueva vista también acomoda varias herramientas que operan en la representación interna en lugar de la forma textual de un programa; estos incluyen editores dirigidos por sintaxis y varias herramientas de análisis.

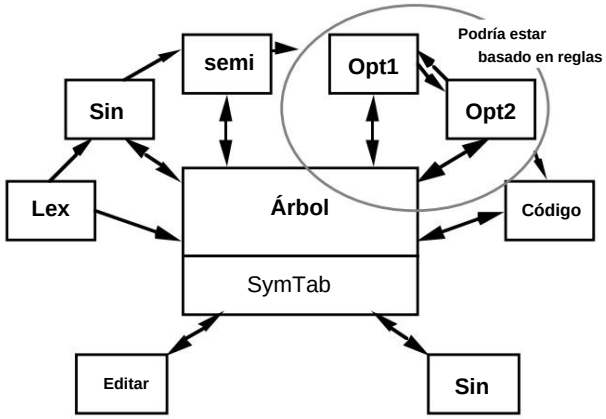

Figura 18: Compilador canónico, revisado

Tenga en cuenta que este repositorio se parece a una pizarra en algunos aspectos y difiere en otros. Como una pizarra, la información del cómputo se ubica centralmente y es operada por cómputos independientes que interactúan solo a través de los datos compartidos. Sin embargo, mientras que el orden de ejecución de las operaciones en una pizarra está determinado por los tipos de modificaciones de la base de datos entrantes, el orden de ejecución del compilador está predeterminado.

4.4. Caso 4: un diseño en capas con diferentes estilos para las capas

El sistema PROVOX® de Fisher Controls ofrece control de proceso distribuido para procesos de producción química [43]. Las capacidades de control de procesos van desde simples lazos de control que controlan la presión, el flujo o los niveles hasta estrategias complejas que involucran lazos de control interrelacionados. Se toman medidas para la integración con los sistemas de gestión e información de la planta en apoyo de la fabricación integrada por computadora. La arquitectura del sistema integra el control de procesos con la gestión de la planta y los sistemas de información en una jerarquía de cinco niveles. La Figura 19 muestra esta jerarquía: el lado derecho es la vista del software y el lado izquierdo es la vista del hardware.

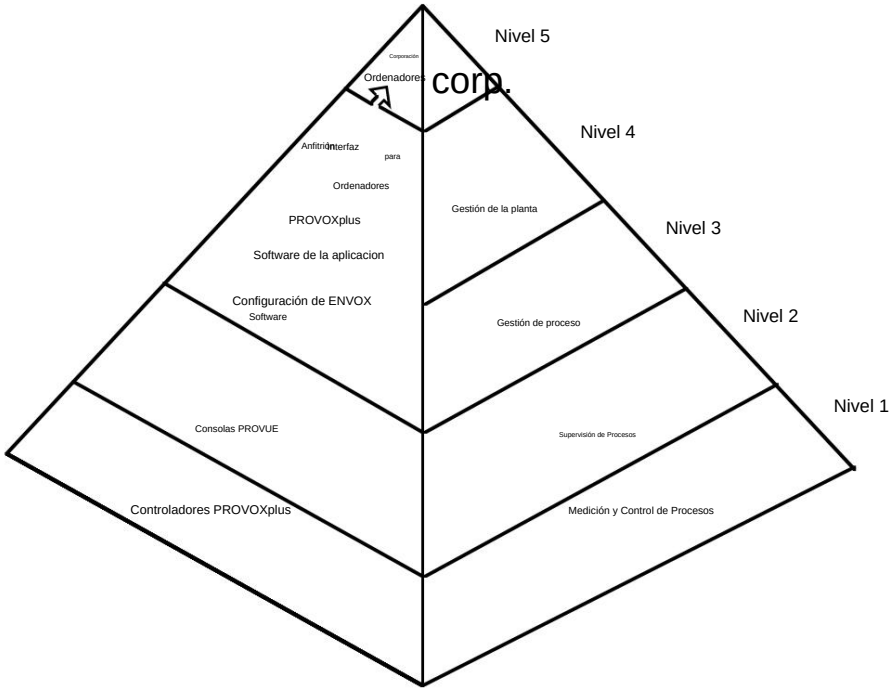

Figura 19: PROVOX – Nivel superior jerárquico

Cada nivel corresponde a una función de gestión de procesos diferente con sus propios requisitos de apoyo a la toma de decisiones.

- Nivel 1: Medida y control de procesos: ajuste directo de los elementos finales de control.
- Nivel 2: Supervisión de procesos: consola de operaciones para monitorear y controlar el Nivel 1.

- Nivel 3: Gestión de procesos: automatización de planta basada en computadora, incluidos informes de gestión, estrategias de optimización y orientación a la consola de operaciones.
- Niveles 4 y 5: Gestión corporativa y de planta: funciones de nivel superior como contabilidad de costos, control de inventario y procesamiento/programación de pedidos.

Se requieren diferentes tipos de tiempos de cálculo y respuesta en diferentes niveles de esta jerarquía. En consecuencia, se utilizan diferentes modelos computacionales.
Los niveles 1 a 3 están orientados a objetos; Los niveles 4 y 5 se basan en gran medida en modelos de depósito de procesamiento de datos convencionales. Para los fines presentes, basta con examinar el modelo orientado a objetos del Nivel 2 y los repositorios de los Niveles 4 y 5.

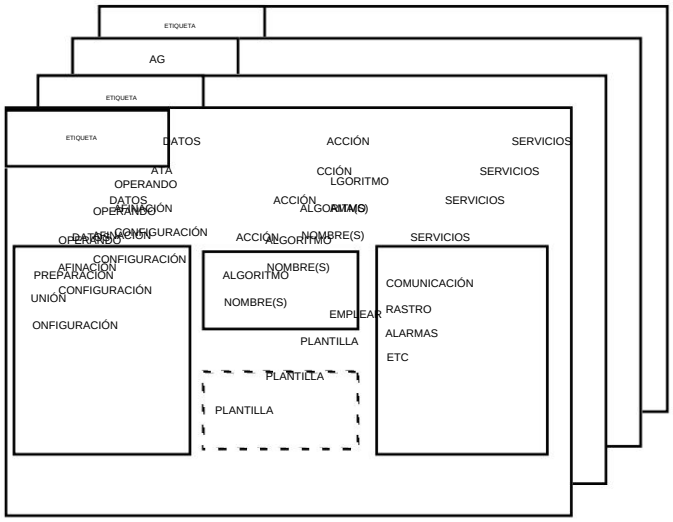

Figura 20: PROVOX – Elaboración Orientada a Objetos

Para las funciones de control y seguimiento del Nivel 2, PROVOX utiliza un conjunto de puntos o loci de control de procesos. La Figura 20 muestra la forma canónica de una definición de punto; siete formas especializadas admiten los tipos de control más comunes. Los puntos son, en esencia, elementos de diseño orientados a objetos que encapsulan información sobre los puntos de control del proceso. Los puntos se configuran individualmente para lograr la estrategia de control deseada. Los datos asociados con un punto incluyen: Parámetros operativos, incluido el valor de proceso actual, el punto de referencia (valor objetivo), la salida de la válvula y el modo (automático o manual).
Parámetros de ajuste, como ganancia, reinicio, derivada y puntos de disparo de alarma.
Parámetros de configuración, incluida la etiqueta (nombre) y los canales de E/S.

Además, los datos del punto pueden incluir una plantilla para una estrategia de control. Como cualquier buen objeto, un punto también incluye definiciones de procedimientos, como algoritmos de control, conexiones de comunicación, servicios de informes e instalaciones de rastreo. Una colección de puntos implementa la estrategia de control de procesos deseada a través de los servicios de comunicación y a través de la dinámica real

del proceso (por ejemplo, si un punto aumenta el flujo hacia un tanque, el valor actual de un punto que detecta el nivel del tanque reflejará este cambio). Aunque la comunicación a través del estado del proceso se desvía del control habitual de objetos basado en mensajes o procedimientos, los puntos son conceptualmente muy parecidos a los objetos en su encapsulación de información esencial sobre el estado y la acción.

Los informes de los puntos aparecen como transacciones de entrada a los procesos de recopilación y análisis de datos en niveles de diseño más altos. El diseñador puede definir la organización de los puntos en procesos de control para que coincida con la estrategia de control de procesos. Estos pueden agregarse aún más en Áreas de proceso de planta (puntos relacionados con un conjunto de equipos, como una torre de enfriamiento) y, de ahí, en Áreas de gestión de planta (segmentos de una planta que estarían controlados por operadores individuales).

PROVOX hace provisiones para la integración con los sistemas comerciales y de gestión de planta en los Niveles 4 y 5. La selección de esos sistemas a menudo es independiente del diseño de control de procesos; PROVOX en sí mismo no proporciona sistemas MIS directamente, pero proporciona la integración de una computadora host convencional con la gestión de bases de datos convencionales. Las instalaciones de recopilación de datos del Nivel 3, las instalaciones de informes del Nivel 2 y la red que admite la implementación distribuida son suficientes para proporcionar información de proceso como transacciones a estas bases de datos. Tales bases de datos se diseñan comúnmente como repositorios, con funciones de procesamiento de transacciones que respaldan un almacén de datos central, un estilo bastante diferente del diseño orientado a objetos del Nivel 2.

El uso de capas jerárquicas en el nivel superior de un sistema es bastante común.
Esto permite una fuerte separación de diferentes clases de funciones e interfaces limpias entre las capas. Sin embargo, dentro de cada capa, las interacciones entre los componentes suelen ser demasiado complejas para permitir una estratificación estricta.

4.5. Caso 5: Un intérprete que usa diferentes modismos para los componentes

Los sistemas basados en reglas proporcionan un medio para codificar los conocimientos de resolución de problemas de los expertos humanos. Estos expertos tienden a captar las técnicas de resolución de problemas como conjuntos de reglas de situación-acción cuya ejecución o activación se secuencia en respuesta a las condiciones de la computación en lugar de un esquema predeterminado. Dado que estas reglas no son ejecutables directamente por las computadoras disponibles, se deben proporcionar sistemas para interpretar tales reglas.
Hayes-Roth estudió la arquitectura y el funcionamiento de los sistemas basados en reglas [44].

Las características básicas de un sistema basado en reglas, que se muestran en la representación de Hayes-Roth como Figura 21, son esencialmente las características de un intérprete basado en tablas, como se describió anteriormente.

- El *pseudocódigo* a ejecutar, en este caso la base de conocimiento
- El *motor de interpretación,* en este caso el intérprete de reglas, el corazón del motor de inferencia

- El *estado de control del motor de interpretación,* en este caso la regla y selector de elementos de datos

- El *estado actual del programa* que se ejecuta en la máquina virtual, en este caso la memoria de trabajo.

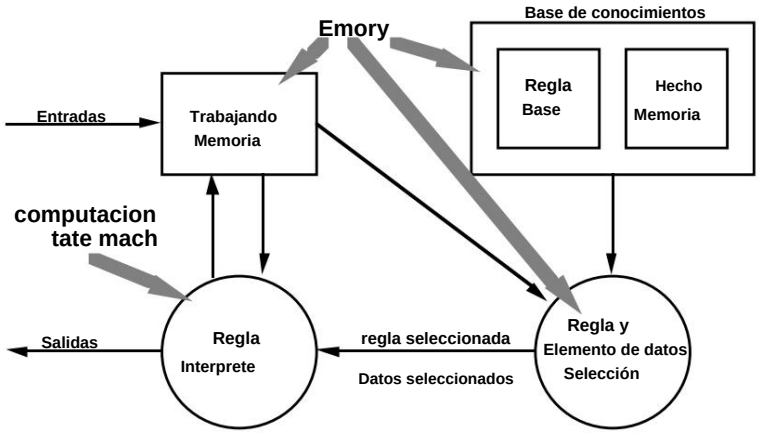

Figura 21: Sistema Básico Basado en Reglas

Los sistemas basados en reglas hacen un uso intensivo de la coincidencia de patrones y el contexto (reglas actualmente relevantes). Agregar mecanismos especiales para estas instalaciones al diseño conduce a la vista más complicada que se muestra en la Figura 22. Al agregar esta complejidad, el intérprete simple original se desvanece en un mar de nuevas interacciones y flujos de datos. Aunque las interfaces entre los módulos originales permanecen, no se distinguen de las interfaces recién agregadas.

Sin embargo, el modelo del intérprete se puede redescubrir identificando los componentes de la Figura 22 con sus antecedentes de diseño en la Figura 21. Esto se hace en la Figura 23. Visto de esta manera, la elaboración del diseño se vuelve mucho más fácil de explicar y comprender. Por ejemplo, vemos que:

- La base de conocimiento sigue siendo una estructura de memoria relativamente simple, simplemente ganando subestructura para distinguir los contenidos activos de los inactivos.

- El intérprete de reglas se amplía con el idioma del intérprete (es decir, el motor de interpretación del sistema basado en reglas se implementa como un intérprete basado en tablas), con procedimientos de control que desempeñan el papel del pseudocódigo que se ejecutará y el pila de ejecución el papel del estado actual del programa.

- La “selección de reglas y elementos de datos” se implementa principalmente como una canalización que transforma progresivamente reglas y hechos activos en activaciones priorizadas; en esta canalización, el tercer filtro ("nominadores") también utiliza una base de datos fija de metarreglas.

- La memoria de trabajo no se elabora más.

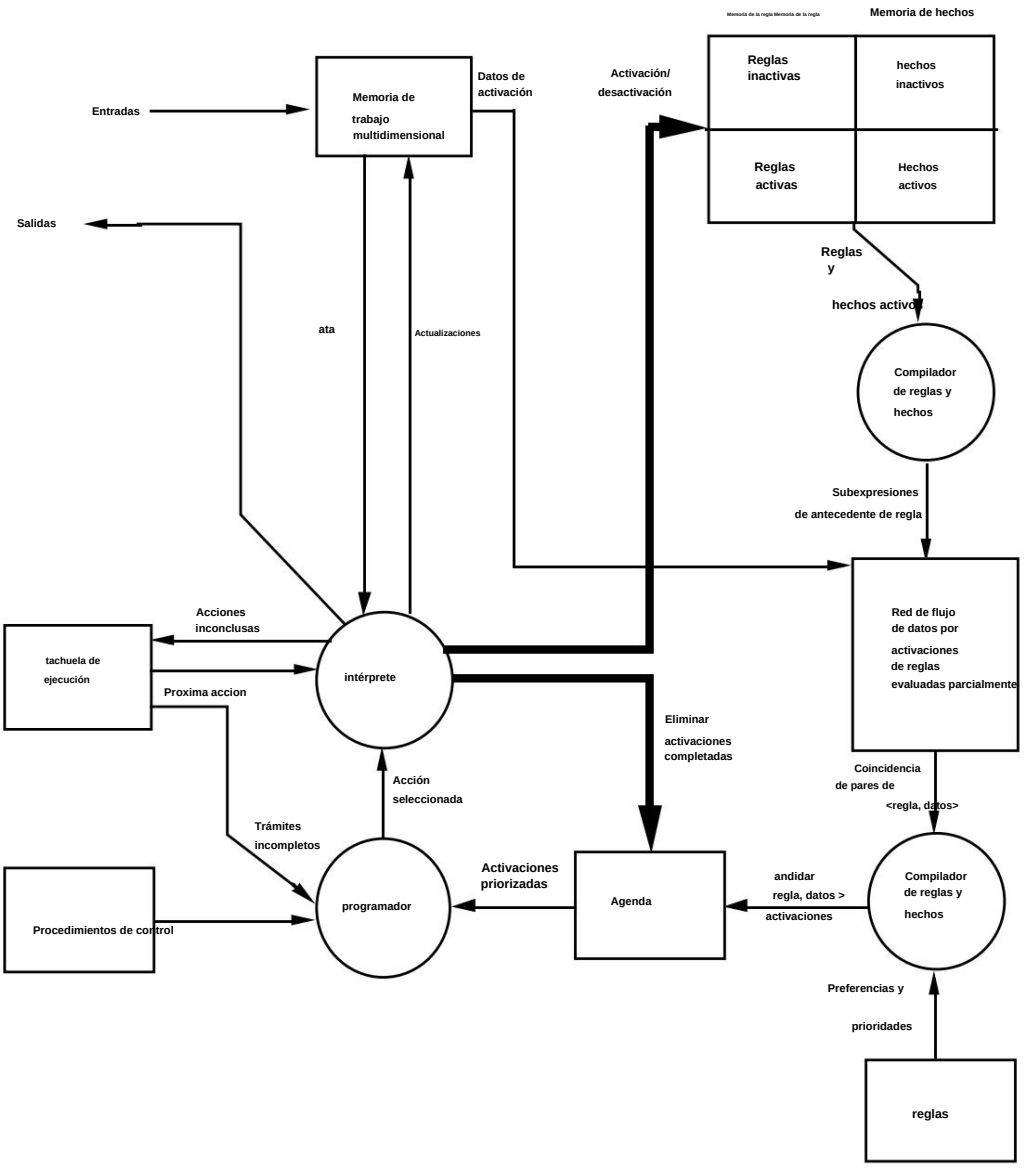

Figura 23: Sistema basado en reglas sofisticado

Las interfaces entre los componentes redescubiertos no han cambiado con respecto al modelo simple, excepto por las dos líneas en negrita sobre las cuales el intérprete controla las activaciones.

Este ejemplo ilustra dos puntos. Primero, en un sistema sofisticado basado en reglas, los elementos del sistema simple basado en reglas se elaboran en respuesta a las características de ejecución de la clase particular de lenguajes que se interpretan. Si el diseño se presenta de esta forma, se mantiene el concepto original para guiar la comprensión y el mantenimiento posterior. En segundo lugar, a medida que se elabora el diseño, se pueden elaborar diferentes componentes del modelo simple con diferentes expresiones idiomáticas.

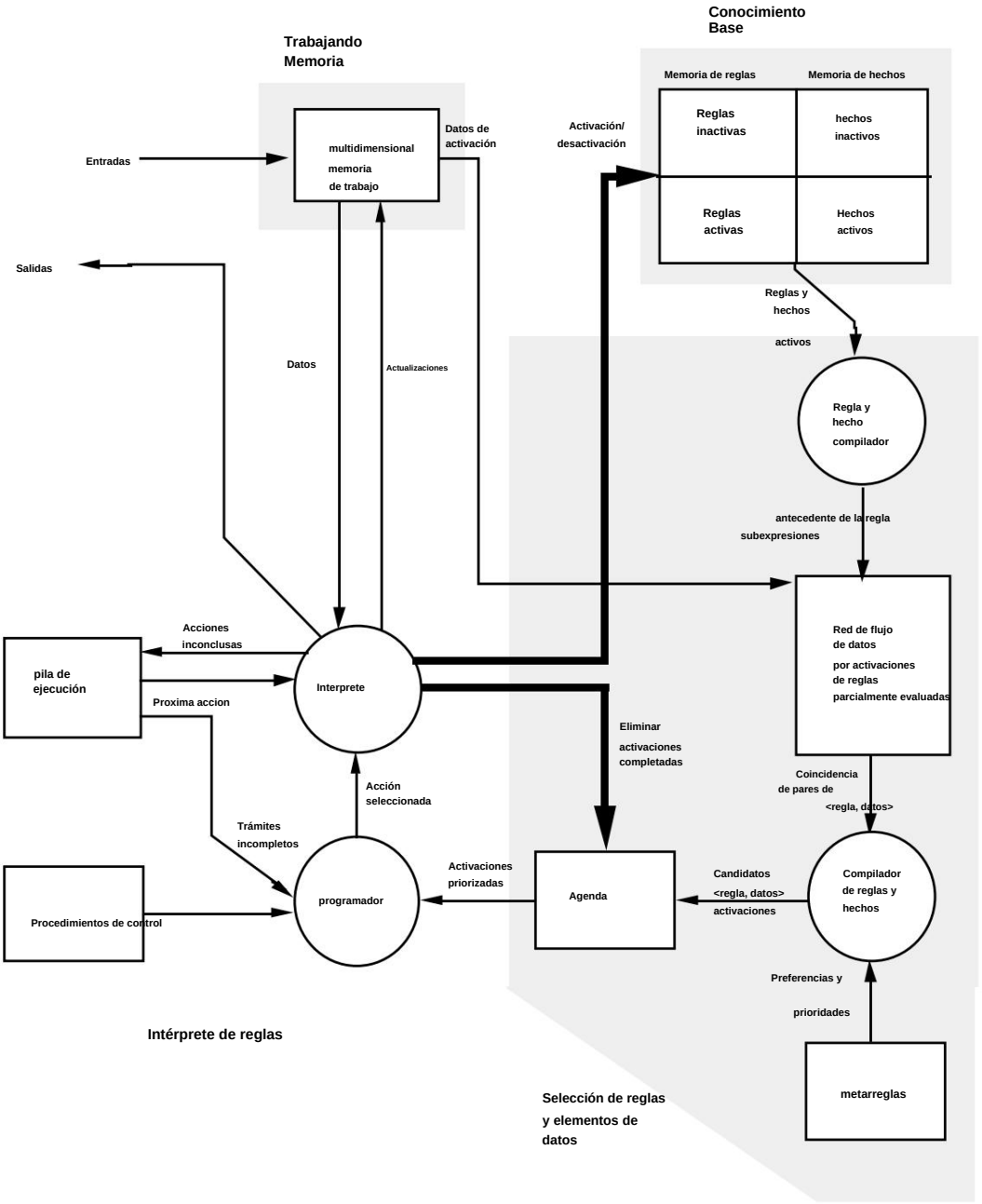

Figura 23: Sistema basado en reglas simplificado y sofisticado

Tenga en cuenta que el modelo basado en reglas es en sí mismo una estructura de diseño: requiere un conjunto de reglas cuyas relaciones de control están determinadas durante la ejecución por el estado del cálculo. Un sistema basado en reglas proporciona una máquina virtual, un ejecutor de reglas, para admitir este modelo.

4.6. Caso 6: Blackboard globalmente refundido como intérprete

El modelo de pizarra de resolución de problemas es un caso especial altamente estructurado de resolución de problemas oportunistas. En este modelo, el espacio de soluciones está organizado en varias jerarquías dependientes de la aplicación y el conocimiento del dominio es

dividido en módulos independientes de conocimiento que operan sobre el conocimiento dentro y entre niveles [34]. La Figura 4 mostró la arquitectura básica de un sistema de pizarra y describió sus tres partes principales: fuentes de conocimiento, la estructura de datos de la pizarra y el control.

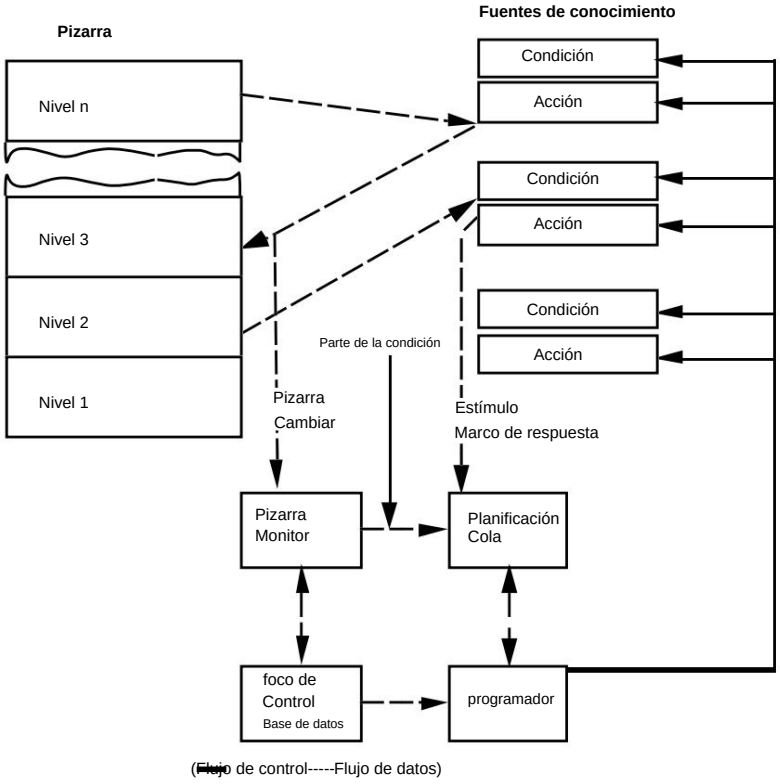

Figura 24: Oído-II

El primer gran sistema de pizarra fue el sistema de reconocimiento de voz HEARSAY-II. El esquema de Nii de la arquitectura HEARSAY-II aparece en la Figura 24.
La estructura de la pizarra es una jerarquía de seis a ocho niveles en la que cada nivel abstrae información de su nivel inferior adyacente y los elementos de la pizarra representan hipótesis sobre la interpretación de un enunciado. Las fuentes de conocimiento corresponden a tareas como la segmentación de la señal en bruto, la identificación de fonemas, la generación de palabras candidatas, la hipótesis de segmentos sintácticos y la propuesta de interpretaciones semánticas. Cada fuente de conocimiento está organizada como una parte de condición que especifica cuándo es aplicable y una parte de acción que procesa elementos de pizarra relevantes y genera otros nuevos. El componente de control se realiza como un monitor de pizarra y un programador; el planificador supervisa la pizarra y calcula las prioridades para aplicar las fuentes de conocimiento a varios elementos de la pizarra.

HEARSAY-II se implementó entre 1971 y 1976 en DEC PDP-10; estas máquinas no eran directamente capaces de control desencadenado por condiciones, por lo que no debería sorprender encontrar que una implementación proporciona los mecanismos de una máquina virtual que realiza la semántica de invocación implícita requerida por el modelo de pizarra.

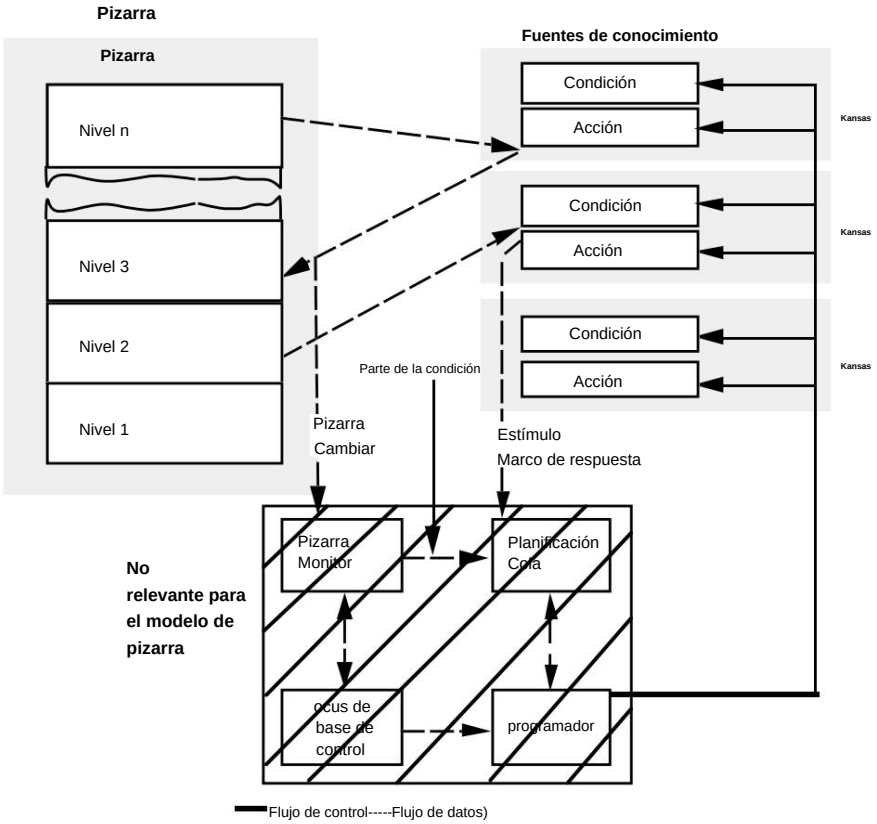

Figura 25: Vista de pizarra de Hearsay-II

La Figura 24 no solo elabora los componentes individuales de la Figura 4; también agrega componentes para el componente de control previamente implícito. En el proceso, la figura se vuelve bastante compleja. Esta complejidad surge porque ahora está ilustrando dos conceptos: el modelo de pizarra y la realización de ese modelo por una máquina virtual. El modelo de pizarra se puede recuperar como en la Figura 25 suprimiendo el mecanismo de control y reagrupando las condiciones y acciones en fuentes de conocimiento.

Se puede ver que la máquina virtual es realizada por un intérprete usando la asignación de función en la Figura 26. Aquí la pizarra corresponde limpiamente al estado actual de la tarea de reconocimiento. La colección de fuentes de conocimiento proporciona aproximadamente el pseudocódigo del intérprete; sin embargo, las acciones también contribuyen al motor de interpretación. El motor de interpretación incluye varios componentes que aparecen explícitamente en la Figura 24: el monitor de pizarra, la base de datos del foco de control y el planificador, pero también las acciones de las fuentes de conocimiento. La cola de programación corresponde aproximadamente al estado de control. En la medida en que la ejecución de condiciones determina las prioridades, las condiciones contribuyen a la selección de reglas, así como a la formación de pseudocódig

Aquí vemos un sistema diseñado inicialmente con un modelo (pizarra, una forma especial de depósito), luego realizado a través de un modelo diferente (intérprete). La realización no es una expansión componente por componente como

en los dos ejemplos anteriores; la vista como intérprete es una agregación diferente de componentes de la vista como pizarra.

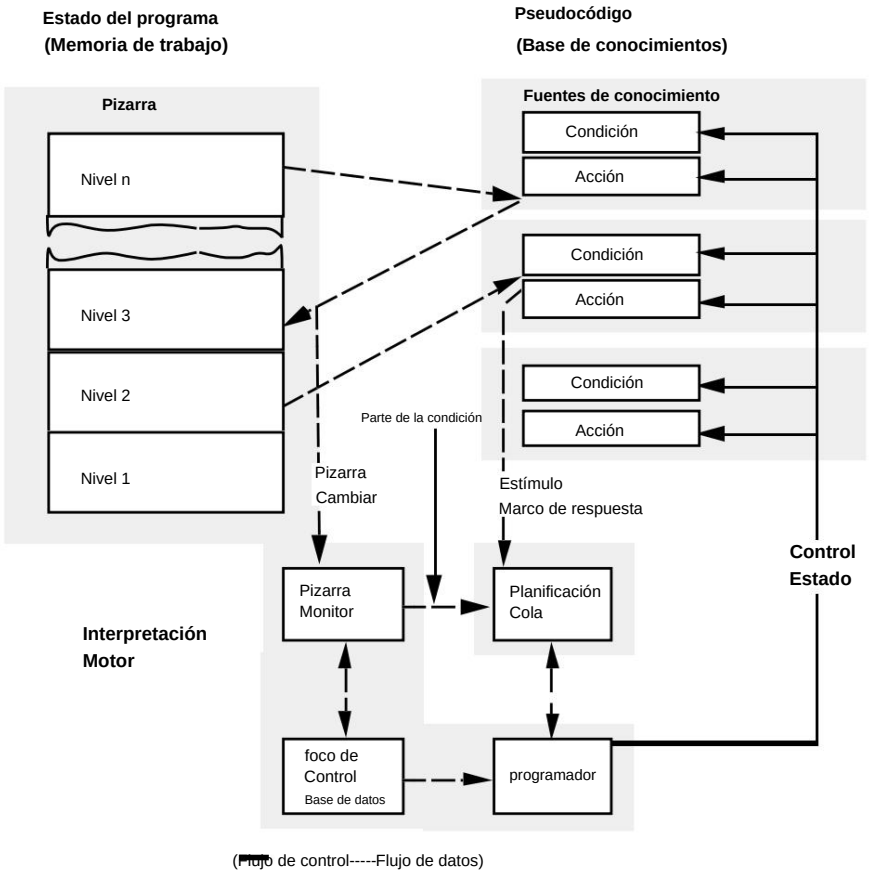

Figura 26: Vista del intérprete de Hearsay-II

5. Pasado, presente y futuro

Hemos esbozado una serie de estilos arquitectónicos y mostrado cómo se pueden aplicar y adaptar a sistemas de software específicos. Esperamos que esto haya convencido al lector de que el análisis y diseño de sistemas en términos de arquitectura de software es viable y vale la pena hacerlo. Además, esperamos haber dejado claro que la comprensión del vocabulario emergente de los estilos arquitectónicos es una herramienta intelectual significativa, si no necesaria, para el ingeniero de software.

Hay, por supuesto, mucho más en la arquitectura de software de lo que hemos tenido espacio para cubrir. En particular, hemos dicho muy poco sobre los resultados existentes en las áreas de análisis, especificación formal, arquitecturas específicas de dominio, lenguajes de interconexión de módulos y herramientas de arquitectura especial.

Esto no quiere decir que no se necesite más trabajo. De hecho, podemos esperar ver avances significativos en una serie de áreas que incluyen:

- Mejores taxonomías de arquitecturas y estilos arquitectónicos.

- Modelos formales para caracterizar y analizar arquitecturas.
- Mejor comprensión de las entidades semánticas primitivas a partir de las cuales se componen estos estilos.
- Notaciones para describir diseños arquitectónicos.
- Herramientas y entornos para el desarrollo de diseños arquitectónicos.
- Técnicas para extraer información arquitectónica del código existente.
- Mejor comprensión del papel de las arquitecturas en el proceso del ciclo de vida.

Todas estas son áreas de investigación activa tanto en la industria como en el mundo académico. Dado el creciente interés en este campo emergente, podemos esperar que nuestra comprensión de los principios y la práctica de la arquitectura de software mejore considerablemente con el tiempo. Sin embargo, como hemos ilustrado, incluso con los conceptos básicos que ahora tenemos a mano, el diseño a nivel de arquitectura de software puede brindar un beneficio directo y sustancial a la práctica de la ingeniería de software.

Agradecimientos

Agradecemos a nuestros muchos colegas que han contribuido a las ideas presentadas en este documento. En particular, nos gustaría agradecer a Chris Okasaki, Curtis Scott y Roy Swonger por su ayuda en el desarrollo del curso del que se extrajo gran parte de este material. Agradecemos a David Notkin, Kevin Sullivan y Gail Kaiser por su contribución a la comprensión de los sistemas basados en eventos. Rob Allen ayudó a desarrollar una comprensión rigurosa del estilo de tubería y filtro. Nos gustaría agradecer al equipo de desarrollo de osciloscopios de Tektronix, y especialmente a Norm Delisle, por su parte en la demostración del valor de los estilos arquitectónicos específicos del dominio en un contexto industrial.
Finalmente, nos gustaría agradecer a Barry Boehm, Larry Druffel y Dilip Soni por sus comentarios constructivos sobre los primeros borradores del documento.

Bibliografía [1] D.

Garlan, M. Shaw, C. Okasaki, C. Scott y R. Swonger, "Experiencia con un curso sobre arquitecturas para sistemas de software", en *Actas de la Sexta Conferencia SEI sobre Educación en Ingeniería de Software,* Springer -Verlag, LNCS 376, octubre de 1992.

[2] M. Shaw, "Hacia abstracciones de alto nivel para sistemas de software", en *Data & Knowledge Engineering,* vol. 5, págs. 119-128, Holanda Septentrional: Elsevier Science Publishers BV, 1990.

[3] M. Shaw, "Heterogeneous design idioms for software architecture", en *Proceedings of the Sixth International Workshop on Software Specification and Design, IEEE Computer Society, Software Engineering Notes* (Como, Italia), págs. 158-165, octubre 25-26, 1991.

[4] M. Shaw, "Arquitecturas de software para sistemas de información compartidos", en *Mind Matters: Contributions to Cognitive and Computer Science in Honor of Allen Newell,* Erlbaum, 1993.

[5] R. Allen y D. Garlan, "Un enfoque formal de las arquitecturas de software", en *Actas de IFIP'92* (J. van Leeuwen, ed.), Elsevier Science Publishers BV, septiembre de 1992.

[6] D. Garlan y D. Notkin, "Formalización de espacios de diseño: mecanismos de invocación implícita", en *VDM'91: Métodos formales de desarrollo de software.* (Noordwijkerhout, Países Bajos), págs. 31-44, Springer-Verlag, LNCS 551, octubre de 1991.

[7] D. Garlan, GE Kaiser y D. Notkin, "Uso de la abstracción de herramientas para componer sistemas", *IEEE Computer,* vol. 25 de junio de 1992.

[8] AZ Spector *et al.,* "Camelot: una instalación de transacciones distribuidas para Mach e Internet: un informe provisional", Tech. Rep. CMU-CS-87-129, Universidad Carnegie Mellon, junio de 1987.

[9] M. Fridrich y W. Older, "Helix: La arquitectura del sistema de archivos distribuido XMS" *Software IEEE,* vol. 2, págs. 21-29, mayo de 1985.

[10] MA Linton, "Gestión distribuida de una base de datos de software", *IEEE Software,* vol. 4, págs. 70-76, noviembre de 1987.

[11] V. Seshadri et al., "Análisis semático en un compilador concurrente", en *Actas de la Conferencia ACM SIGPLAN '88 sobre diseño e implementación de lenguajes de programación,* Avisos ACM SIGPLAN, 1988.

[12] MC Paulk, "La red ARC: un estudio de caso", *IEEE Software,* vol. 2, págs. 61-69, mayo de 1985.

[13] M. Chen y RJ Norman, "Un marco para el caso integrado", IEEE Software, vol. 9, págs. 18-22 de marzo de 1992.

[14] Modelo de referencia NIST/ECMA para marcos de entornos de ingeniería de software". Publicación especial NIST 500-201, diciembre de 1991.

[15] RW Scheifler y J. Gettys, "El sistema de ventanas X", *AACM Transactions on Graphics,* vol. 5, págs. 79-109, abril de 1986.

[16] MJ Bach, *El diseño del sistema operativo UNIX,* cap. 5.12, págs. 111-119. Serie de software, Prentice-Hall, 1986.

[17] N. Delisle y D. Garlan, "Aplicación de especificación formal a problemas industriales: A especificación de un osciloscopio", *IEEE Software,* septiembre de 1990.

[18] G. Kahn, "La semántica de un lenguaje simple para programación paralela", *Información Procesamiento,* 1974.

[19] MR Barbacci, CB Weinstock y JM Wing, "Programación en el nivel de interruptor de memoria del procesador", en *Actas de la 10ª Conferencia Internacional sobre Ingeniería de Software,* (Singapur), págs. 19-28, IEEE Computer Society Press, abril de 1988.

[20] GE Kaiser y D. Garlan, "Sintetizar entornos de programación a partir de funciones reutilizables", en Software Reusability (TJ Biggerstaff y AJ Perlis, eds.), vol. 2, Prensa ACM, 1989.

[21] W. Harrison, "RPDE: un marco para integrar fragmentos de herramientas", IEEE Software, vol. 4, noviembre de 1987.

[22] C. Hewitt, "Planificador: un lenguaje para probar teoremas en robots", en *Actas de la Primera Conferencia Internacional Conjunta sobre Inteligencia Artificial,* 1969.

[23] SP Reiss, "Conexión de herramientas mediante el paso de mensajes en el entorno de desarrollo de programas de campo", *IEEE Software,* julio de 1990.

[24] C. Gerety, "HP Softbench: una nueva generación de herramientas de desarrollo de software", Tech. Reps. SESD-89-25, División de Sistemas de Ingeniería de Software de Hewlett-Packard, Fort Collins, Colorado, noviembre de 1989.

[25] RM Balzer, "Vivir con el sistema operativo de próxima generación", en *Actas de la 4ª Conferencia Mundial de Computación,* septiembre de 1986.

[26] G. Krasner y S. Pope, "Un libro de cocina para usar el paradigma de interfaz de usuario modelo-vista-controlador en Smalltalk-80", Revista de programación orientada a objetos, vol. 1, págs. 26-49, agosto/septiembre de 1988.

[27] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols y R. Pausch, "Descartes: Un enfoque de lenguaje de programación para las interfaces de pantalla interactiva", *Actas de SIGPLAN '83: Simposio sobre Problemas del lenguaje de programación en los sistemas de software, ACM SIGPLAN Notices,* vol. 18, págs. 100-111, junio de 1983.

[28] AN Habermann y DS Notkin, "Gandalf: entornos de desarrollo de software", *IEEE Transactions on Software Engineering,* vol. SE-12, págs. 1117-1127, diciembre de 1986.

[29] AN Habermann, D. Garlan y D. Notkin, "Generación de entornos de software integrados para tareas específicas", en *CMU Computer Science: A 25th Conmemorative* (RF Rashid, ed.), Anthology Series, págs. 69-98 , Prensa ACM, 1991.

[30] K. Sullivan y D. Notkin, "Conciliación de la integración del entorno y la independencia de los componentes", en *Actas de ACM SIGSOFT90: Cuarto simposio sobre entornos de desarrollo de software,* págs. 22 y 23, diciembre de 1990.

[31] GR McClain, ed., *Manual de interconexión de sistemas abiertos.* Nueva York, NY: Intertexto Publicaciones McGraw-Hill Book Company, 1991.

[32] D. Batory y S. O'Malley, "El diseño y la implementación de sistemas de software jerárquicos utilizando componentes reutilizables", Tech. Rep. TR-91-22, Departamento de Ciencias de la Computación, Universidad de Texas, Austin, junio de 1991.

[33] HC Lauer y EH Satterthwaite, “Impacto de MESA en el diseño de sistemas”, en *Actas de la Tercera Conferencia Internacional sobre Ingeniería de Software,* (Atlanta, GA), págs. 174-175, IEEE Computer Society Press, mayo de 1979.

[34] HP Nii, "Blackboard system Parts 1 & 2", AI Magazine, vol. 7 nos 3 (pp. 38-53) y 4 (pp. 62-69), 1986.

[35] V. Ambriola, P. Ciancarini y C. Montangero, “Software process enactment in oikos”, en *Actas del Cuarto Simposio ACM SIGSOFT sobre Entornos de Desarrollo de Software,* SIGSOFT Software Engineering Notes, (Irvine, CA), págs. 183-192, diciembre de 1990.

[36] DR Barstow, HE Shrobe y E. Sandewall, eds., *Entornos de programación interactivos.* McGraw-Hill Book Co., 1984.

[37] GR Andrews, "Paradigmas para la interacción de procesos en programas distribuidos", *ACM Computing Surveys,* vol. 23, págs. 49-90, marzo de 1991.

[38] A. Berson, *Arquitectura Cliente/ Servidor.* McGraw-Hill, 1992.

[39] E. Mettala y MH Graham, eds., *El programa de arquitectura de software de dominio específico.* No. CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, junio de 1992.

[40] D. Harel, "Statecharts: un formalismo visual para sistemas complejos", *Science of Computer Programming,* vol. 8, págs. 231-274, 1987.

[41] KJ Åström y B. Wittenmark, *Diseño de sistemas controlados por computadora.* Prentice Hall, segunda ed., 1990.

[42] DL Parnas, “Sobre los criterios a utilizar en la descomposición de sistemas en módulos”, *Comunicaciones de la ACM,* vol. 15, págs. 1053-1058, diciembre de 1972.

[43] “Sistema de instrumentación PROVOX plus: descripción general del sistema”, 1989.

[44] F. Hayes-Roth, 'Sistemas basados en reglas', Comunicaciones de la ACM, vol. 28, págs. 921-932, septiembre de 1985.