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 Carnegie Mellon
Pittsburgh, PA 15213-3890

También publicado como "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering, Volume I*, editado por V.Ambriola y G.Tortora, World Scientific Publishing Company, Nueva Jersey, 1993.

También aparece como Informe Técnico del Instituto de Ingeniería de Software CMU
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 los Estados Unidos). 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 Estados Unidos, el Departamento de Defensa, la Fundación Nacional de Ciencias, Siemens Corporation o la Universidad Carnegie Mellon.

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 del cálculo 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 documento 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 estudios de caso para ilustrar cómo las representaciones arquitectónicas pueden mejorar nuestra comprensión de los sistemas de software complejos. Finalmente, examinamos algunos de los problemas pendientes 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 de diseño va más allá de los algoritmos y las estructuras de datos de la computación: diseñar y especificar la estructura general del sistema surge como un nuevo tipo de problema. Las cuestiones estructurales incluyen la organización bruta y la estructura de control global;

protocolos de comunicación, sincronización y acceso a datos; asignación de funcionalidad a los elementos de diseño; distribución física; composición de los elementos de diseño; escalado y rendimiento; y selección entre alternativas de diseño.

Este es el nivel de diseño de la arquitectura de software. Existe un considerable cuerpo de trabajo sobre este tema, incluidos 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 implícito de trabajo en forma de términos descriptivos utilizados 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 uso común de los principios arquitectónicos al diseñar software complejo. Muchos de los principios representan reglas generales o patrones idiomáticos que han surgido informalmente con el tiempo. Otros están más cuidadosamente documentados como estándares científicos y de la industria.

Cada vez está más claro que la ingeniería de software efectiva requiere facilidad en el diseño de software arquitectónico. En primer lugar, es importante ser capaz de reconocer paradigmas comunes para que las relaciones de alto nivel entre los sistemas puedan ser entendidas y para que los nuevos sistemas puedan ser construidos como variaciones de los sistemas antiguos. En segundo lugar, obtener la arquitectura correcta es a menudo crucial para el éxito del diseño de un sistema de software; El equivocado puede llevar 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, una representación del sistema arquitectónico es a menudo 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 selecciona de un curso semestral, Arquitecturas para Sistemas de Software, impartido en CMU por los autores [1]. Naturalmente, un breve documento como este solo puede resaltar brevemente las características principales del terreno. Esta selección enfatiza las 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 la importancia 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 los estilos heterogéneos se pueden combinar en un solo diseño. A continuación, utilizamos seis estudios de caso 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 pendientes 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 estilos arquitectónicos y los estudios de caso han incorporado partes de varios artículos publicados [1, 2, 3, 4]. En menor medida, el material ha sido 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 los lenguajes y herramientas de programación han sido los aumentos regulares en el nivel de abstracción, o el tamaño conceptual de los bloques de construcción de los diseñadores de software. Para colocar el campo de la Arquitectura de Software en perspectiva, comencemos por observar el desarrollo histórico de las técnicas de abstracción en ciencias de la computación.

2.1.Lenguajes de programación de alto nivel

Cuando surgieron las computadoras digitales en la década de 1950, el software estaba escrito en lenguaje de máquina; Los programadores colocaron instrucciones y datos individual y explícitamente en la memoria de la computadora. La inserción de una nueva instrucción en un programa puede requerir la comprobación manual de todo el programa para actualizar las referencias a los datos e instrucciones que se movieron como resultado de la inserción. Finalmente, 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 los códigos de operación y las direcciones de memoria. El resultado fueron ensambladores simbólicos. Pronto fueron seguidos por procesadores macro, que permitían que un solo símbolo representara una secuencia de instrucciones de uso común. La sustitución de símbolos simples por códigos de operación de máquinas, direcciones de máquina aún por definir y secuencias de instrucciones fue quizás la primera forma de abstracción en software.

En la última parte 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 primeros lenguajes de alto nivel, de los cuales Fortran fue el principal sobreviviente.

Los lenguajes de nivel superior permitieron desarrollar programas más sofisticados y surgieron patrones en el uso de datos. Mientras que en Fortran los tipos de datos servían principalmente como señales para seleccionar las instrucciones de máquina adecuadas, los tipos de datos en Algol y sus sucesores sirven para establecer las intenciones del programador sobre cómo se deben usar los datos. Los compiladores para 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 tipo.

El progreso en el diseño del lenguaje continuó con la introducción de módulos para proporcionar protección para 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 finales de la década de 1960, los buenos programadores compartían una intuición sobre el desarrollo de software: si obtienes las estructuras de datos correctas, el esfuerzo hará que el desarrollo del resto del programa sea mucho más fácil. El trabajo de tipo 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 implicaba la comprensión

- *la estructura del software* (que incluía una representación empaquetada con sus operadores primitivos),
- *especificaciones* (expresadas matemáticamente como modelos abstractos o axiomas algebraicos),
- *problemas lingüísticos* (módulos, alcance, tipos definidos por el usuario),
- *integridad del resultado* (invariantes de las estructuras de datos y protección contra otras manipulaciones),
- *normas para la combinación de 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 una comprensión de una buena organización para un módulo completo que sirve a un propósito particular. Esto implicó combinar representaciones, algoritmos, especificaciones e interfaces funcionales de manera uniforme. Se requería cierto apoyo del lenguaje de programación, por supuesto, pero el paradigma del tipo de datos abstracto 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 de lenguajes de programación.

2.3.Softw son Arquitectura

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 tipos de datos abstractos. Pero esta no es la única forma de organizar un sistema de software.

Muchas otras organizaciones se han desarrollado informalmente con el tiempo, y ahora son parte del vocabulario de los diseñadores de sistemas de software. Por ejemplo, las descripciones típicas de arquitecturas de software incluyen sinopsis como (cursiva 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]
- "*Las capas de abstracción* y la descomposición del sistema proporcionan la apariencia de uniformidad del sistema a los clientes, pero permiten a Helix acomodar una diversidad de dispositivos autónomos. La arquitectura fomenta un *modelo cliente-servidor* para la estructuración de aplicaciones". [9]
- "Hemos elegido un *enfoque distribuido y orientado a objetos* para gestionar 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 actualmente a través de las diversas fases de compilación [por múltiples procesos de compilación].] antes de una pasada, de fusión final se recombina 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 genérica de entorno de ingeniería de software basada en sustratos de comunicación en capas) [13, 14] y el Sistema X Window (una arquitectura de interfaz de usuario de ventana distribuida basada en la activación de eventos y devoluciones de llamada) [15].

Todavía estamos lejos de tener una taxonomía bien aceptada de tales paradigmas arquitectónicos, y mucho menos una teoría completamente desarrollada de la arquitectura de software. Pero ahora podemos identificar claramente una serie de patrones arquitectónicos, o estilos, 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 las opciones arquitectónicas e indicar cuáles son algunas de las compensaciones al 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 componentes computacionales, o simplemente componentes, junto con una descripción de las interacciones entre estos componentes, los *conectores*. Gráficamente hablando, esto lleva 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 llamada a procedimiento, difusión de eventos, consultas de base de datos y canalizaciones.

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, las que tienen que ver con la semántica de ejecución, también podrían ser parte de la definición de estilo.

Dado este marco, podemos entender qué es un estilo respondiendo a las siguientes preguntas: ¿Cuál es el patrón estructural: los componentes, conectores y restricciones? ¿Cuál es el modelo computacional subyacente? ¿Cuáles son los 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 calculando de forma 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 salidas de un filtro a entradas de otro. Por lo tanto, los conectores se denominan "tuberías".

Entre los invariantes importantes del estilo, los filtros deben ser entidades independientes: en particular, no deben compartir estado con otros filtros. Otro invariante importante es que los filtros no conocen la identidad de sus filtros ascendentes y descendentes. Sus especificaciones pueden restringir lo que aparece en las tuberías de entrada o hacer garantías sobre lo que aparece en las tuberías de salida, pero pueden no identificar 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 *tuberías*, que restringen las topologías a secuencias lineales de filtros; tuberías acotadas, que restringen la cantidad de datos que pueden residir en una tubería; y tuberías con tipo, que requieren que los datos pasados entre dos filtros tengan un tipo bien definido.

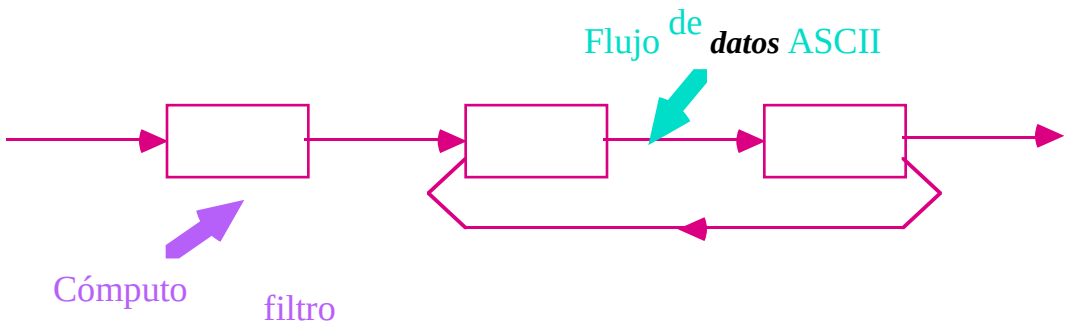

Figura 1: Tuberías y filtros

Un caso degenerado de una arquitectura de canalización se produce 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 programas escritos en el shell de Unix [16]. Unix soporta este estilo proporcionando una notación para conectar componentes (representados como procesos Unix) y proporcionando mecanismos de tiempo de ejecución para implementar tuberías. Como otro ejemplo bien conocido, tradicionalmente los compiladores han sido vistos como sistemas de tuberías (aunque las fases a menudo no son incrementales). Las etapas en la tubería incluyen análisis léxico, análisis, análisis semántico, generación de código. (Volvemos a este ejemplo en los estudios de caso). Otros ejemplos de tuberías 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. Primero, permiten al diseñador comprender el comportamiento general de entrada / salida de un sistema como una composición simple de los comportamientos de los filtros individuales. En segundo lugar, apoyan la reutilización: dos filtros cualesquiera se pueden conectar juntos, siempre que estén de acuerdo con 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, naturalmente admiten la ejecución concurrente. Cada filtro puede implementarse como una tarea separada y potencialmente ejecutarse en paralelo con otros filtros.

[1] En general, encontramos que los límites de los estilos pueden superponerse. Esto no debe disuadirnos de identificar las características principales de un estilo con sus ejemplos centrales de uso.

Pero estos sistemas también tienen sus desventajas. [2] En primer lugar, los sistemas de tuberías y filtros a menudo conducen a una organización por lotes de procesamiento. Aunque los filtros pueden procesar datos de forma incremental, ya 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 a los datos de salida. En particular, debido a su carácter transformador, los sistemas de tuberías y filtros generalmente no son buenos para manejar aplicaciones interactivas. Este problema es más grave cuando se requieren actualizaciones de pantalla 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 corrientes separadas, pero relacionadas. En tercer lugar, dependiendo de la aplicación, pueden forzar una

mon denominador en la transmisión de datos, lo que resulta en un trabajo adicional para que cada filtro analice y desanalice sus datos. Esto, a su vez, puede conducir tanto a la pérdida de rendimiento como a una mayor complejidad en la escritura de los propios 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 tipo de datos u objeto abstracto. Los componentes de este estilo son los objetos o, si se quiere, instancias de los tipos de datos abstractos. Los objetos son ejemplos de un tipo 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 algún invariante sobre él), y (b) que la representación está oculta de otros objetos. La figura 2 ilustra este estilo.[3]

[2] Esto es cierto a pesar del hecho de que las tuberías y filtros, como todos los estilos, hanEs un conjunto de seguidores religiosos devotos, personas que creen que todos los problemas que vale la pena resolver pueden resolverse mejor usando ese estilo en particular.

[3] No hemos mencionado la herencia en esta descripción. Si bien la herencia es un principio organizador importante para defEn los tipos de objetos de un sistema, no tiene una función arquitectónica directa. En particular, en nuestra opinión, una relación de herencia no es un conector, ya que no define la interacción entre los componentes de un sistema. Además, en un arcoLa herencia de propiedades de configuración hitectural no se limita a los tipos de objetos, sino que puede incluir conectores e incluso estilos arquitectónicos.

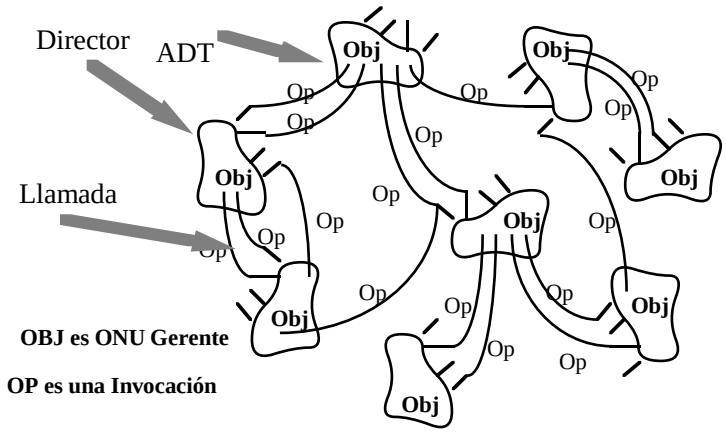

Figura 2: Tipos de datos y objetos abstractos

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 simultáneas; otros permiten que los objetos tengan múltiples interfaces [20, 21].

Los sistemas orientados a objetos tienen muchas propiedades agradables, 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. Lo 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 la identidad de un objeto cambia, 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 utiliza 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 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 radiodifusión selectiva. Este estilo tiene raíces históricas en 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 difundir) 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. Por lo tanto, 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 al sistema invocar 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 supervisadas. En este esquema, el depurador simplemente anuncia un evento, pero no sabe qué otras herramientas (si las hay) están relacionadas con ese evento, o qué harán cuando se anuncie ese evento.

Arquitectónicamente hablando, los componentes en 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 procedimientos pueden llamarse de la manera 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.

El principal invariante de este estilo es que los anunciadores 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 implícitos también incluyen la invocación explícita (es decir, la llamada a 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 administran los datos [26, 27], y por editores dirigidos por 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 puede introducirse 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 del sistema.

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

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, el razonamiento sobre la corrección puede ser problemático, ya que el significado de un procedimiento que anuncia eventos dependerá 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 estratificados***

Un sistema en capas se organiza jerárquicamente, cada capa proporciona servicio a la capa superior y sirve como cliente a la capa inferior. En algunos sistemas, las capas internas están ocultas de todas excepto de la capa externa adyacente, excepto para ciertas funciones cuidadosamente seleccionadas para la exportación. 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 limitar las interacciones a capas adyacentes. La figura 3 ilustra este estilo.

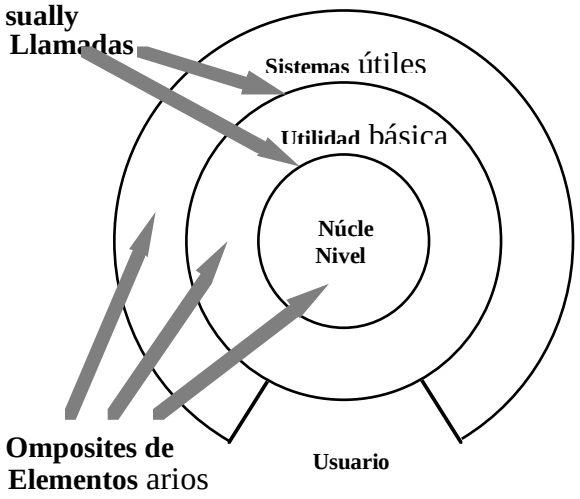

Figura 3: Sistemas en capas

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, los más bajos suelen definirse mediante 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. En primer lugar, apoyan el diseño basado en niveles crecientes de abstracción. Esto permite a los implementadores particionar un problema complejo en una secuencia de pasos incrementales. En segundo lugar, apoyan la mejora. Al igual que las canalizaciones, debido a que cada capa interactúa con un máximo de las capas inferiores y superiores, los cambios en la función de una capa afectan como máximo a otras dos capas. En tercer lugar, apoyan la reutilización. Al igual que los tipos de datos abstractos, diferentes implementaciones de la misma capa se pueden usar indistintamente, siempre que admitan las mismas interfaces para sus capas adyacentes. Esto conduce a la posibilidad de definir interfaces de capa estándar a las que diferentes implementadores pueden construir. (Un buen ejemplo es el modelo ISO OSI 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 caso). 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 estandarizados en capas. Uno señala 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 la ocultación de 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 tuberías, en que los componentes se comunican a lo sumo 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 mucho más ricas de interacción. Esto dificulta la definición de capas independientes del sistema (como con los filtros), ya que una capa debe admitir los protocolos específicos en sus límites superior e inferior. Pero también permite una interacción mucho más estrecha entre capas y permite la transmisión bidireccional de información.

3.5. Repositorios

En un estilo de repositorio hay dos tipos muy distintos de componentes: una estructura de datos central representa el estado actual y una colección de componentes independientes 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 a ejecutar, el repositorio puede ser una base de datos tradicional. Si el estado actual de la

estructura central de datos es el principal desencadenante de la selección de procesos a 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 las fuentes de conocimiento tiene lugar únicamente a través de la pizarra.

La estructura de datos de la 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: impulsado íntegramente por el estado de la pizarra. Las fuentes de conocimiento responden de manera oportunista cuando los cambios en la pizarra los hacen aplicables.

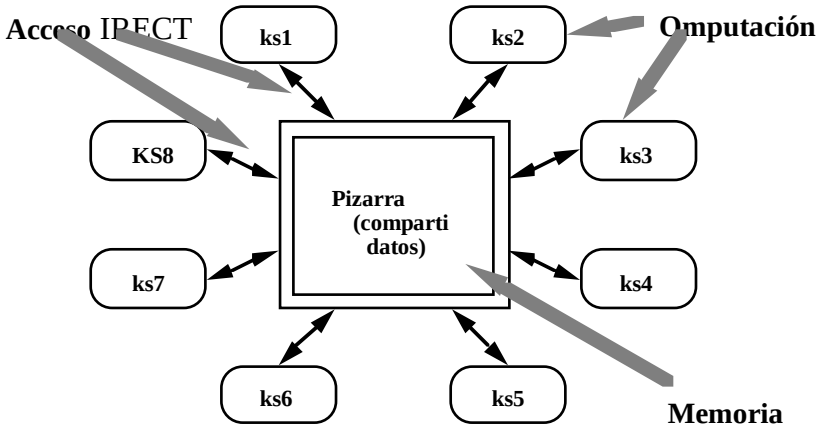

Figura 4: La pizarra

En el diagrama no hay representación explícita del componente de control. La invocación de una fuente de conocimiento se desencadena por el estado de la pizarra. El locus 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 Blackboard se han utilizado tradicionalmente para aplicaciones que requieren interpretaciones complejas del procesamiento de señales, como el reconocimiento de voz y patrones. Varios de estos son encuestados por Nii [34]. También han aparecido en otros tipos de sistemas que implican el acceso compartido a datos con agentes débilmente acoplados [35].

Hay, por supuesto, muchos otros ejemplos de sistemas de repositorio. 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 una arquitectura de compilador se ha presentado tradicionalmente como una tubería, 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 basados en tablas

En una organización de intérpretes, una máquina virtual se produce en software. Un intérprete incluye el pseudo-programa 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. Por lo tanto, un intérprete generalmente tiene cuatro componentes: un motor de interpretación para hacer el trabajo, una memoria que contiene el pseudo-código 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.)

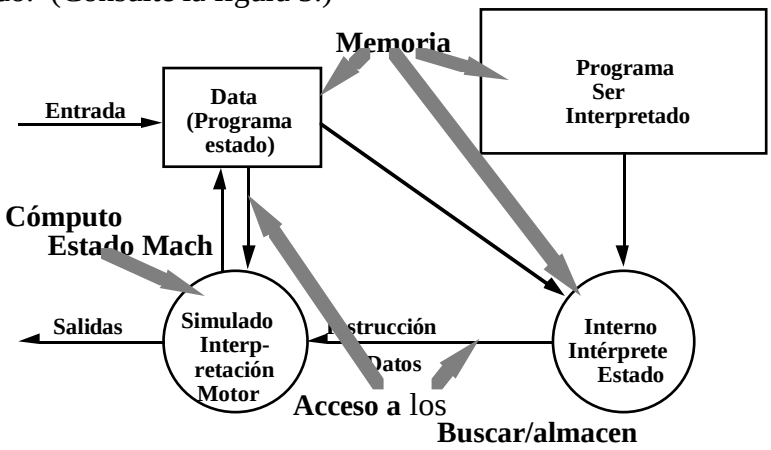

Figura 5: Intérprete

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

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

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 pueden caracterizarse principalmente por sus características topológicas, como las organizaciones de anillos y estrellas. Otros se caracterizan mejor en términos de los tipos de protocolos entre procesos que se utilizan para la comunicación (por ejemplo, algoritmos de latidos del corazón).

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

- **Organizaciones principales de programas/**subrutinas: La organización primaria de muchos sistemas refleja el lenguaje de programación en el que se encuentra el sistema.

  está escrito. Para los lenguajes sin soporte para la modularización, esto a menudo resulta en un sistema organizado alrededor de un programa principal y un conjunto de subrutinas. El programa principal actúa como el controlador de las subrutinas, proporcionando típicamente un bucle de control para la secuenciación a través de las subrutinas en algún orden.

- **Arquitecturas de software específicas de dominio:** Recientemente ha habido un interés considerable en desarrollar arquitecturas de "referencia" para dominios específicos [39]. Estas arquitecturas proporcionan una estructura organizativa adaptada a una familia de aplicaciones, como aviónica, comando 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 que un sistema ejecutable pueda generarse automática o semiautomáticamente a partir de la propia descripción arquitectónica.

- **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 bucle de retroalimentación en el que las entradas de los sensores son utilizadas por el sistema de control de procesos para determinar un conjunto de salidas que producirán un nuevo estado del medio ambiente.

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 suelen implicar alguna combinación de varios estilos.

Hay diferentes maneras en que los estilos arquitectónicos se pueden combinar. Una forma es a través de la jerarquía. Un componente de un sistema organizado en un estilo arquitectónico puede tener una estructura interna que se desarrolla 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 tubería y filtro.

Lo que quizás sea más sorprendente es que los conectores también pueden descomponerse jerárquicamente. Por ejemplo, un conector de tubería se puede implementar 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 mezcla de conectores arquitectónicos. Por ejemplo, un componente puede acceder a un repositorio a través de parte de su interfaz, pero interactuar a través de tuberías con otros componentes de un sistema y aceptar información de control a través de otra parte de su interfaz. (De hecho, los sistemas de tuberías y filtros de Unix hacen esto, el sistema de archivos desempeña el papel del 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 la invocación implícita. En esta organización, los componentes externos registran interés en partes de la base de datos. La base de datos invoca automáticamente las herramientas adecuadas basadas en esta asociación. (Las pizarras a menudo se construyen de esta manera; las fuentes de conocimiento están asociadas 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 estudios de caso.

4.Casos prácticos

Ahora presentamos seis ejemplos para ilustrar cómo se pueden utilizar los principios arquitectónicos para aumentar nuestra comprensión de los sistemas de software. El primer ejemplo muestra cómo diferentes soluciones arquitectónicas para el mismo problema proporcionan diferentes beneficios. El segundo estudio de caso resume la experiencia en el desarrollo de un estilo arquitectónico específico de dominio para una familia de productos industriales. El tercer estudio de caso 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 [Key Word in Context] 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 puede ser "desplazada circularmente" eliminando repetidamente la primera palabra y añadiéndola al final de la línea. El sistema de índice KWIC genera una lista de todos los desplazamientos 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 representaciones de datos, y una segunda basada en una descomposición que oculta las decisiones de diseño. Desde su introducción, el problema se ha hecho bien conocido y es ampliamente utilizado como un dispositivo de enseñanza en ingeniería de software. Garlan, Kaiser y Notkin también usan 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 los científicos informáticos. 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 de software, el problema deriva su atractivo del hecho de que puede usarse para ilustrar el efecto de los cambios en el diseño de software. Parnas muestra que las diferentes descomposiciones problemáticas 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 desplazamiento de línea se puede realizar en cada línea a medida que se lee desde el dispositivo de entrada, en todas las líneas después de que se leen, o bajo demanda cuando la alfabetización requiere un nuevo conjunto de líneas desplazadas.
- Cambios en la representación de datos: Por ejemplo, las líneas se pueden almacenar de varias maneras. Del mismo modo, los desplazamientos 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 de ruido (como "a", "an", "y", etc.). Cambie el sistema para que sea interactivo y permita al usuario eliminar líneas de las listas originales (o, alternativamente, de las desplazadas circularmente).
- Rendimiento: Tanto en el espacio como en el tiempo.

- Reutilización: Hasta qué punto los componentes pueden servir como entidades reutilizables.

Ahora esbozamos 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. El cuarto es una solución de canalización inspirada en la utilidad de índice 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 detalladas 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 secuencia a través de ellos 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 un protocolo de escritura de lectura sin restricciones. Esto es posible gracias al hecho de que el programa de coordinación garantiza el acceso secuencial a los datos. (Consulte la figura 6).

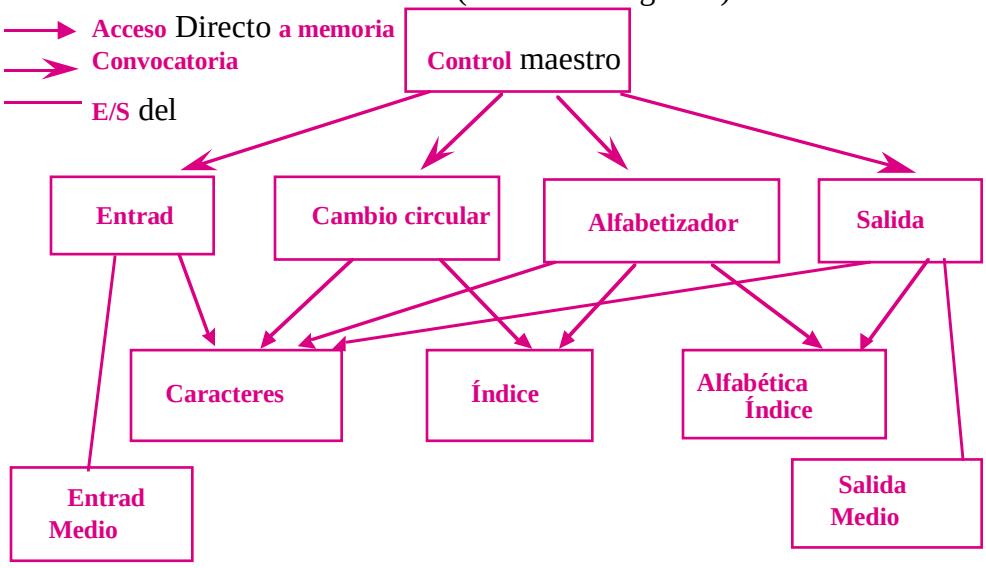

Figura 6: KWIC – Solución de datos compartidos

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

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

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 su lugar, cada módulo proporciona una interfaz que permite que otros componentes accedan a los datos solo invocando procedimientos en esa interfaz. (Consulte la figura 7, que ilustra cómo cada uno de los componentes tiene ahora un conjunto de procedimientos que determinan la forma de acceso de otros componentes del 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 los 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 otros con los que interactúan.

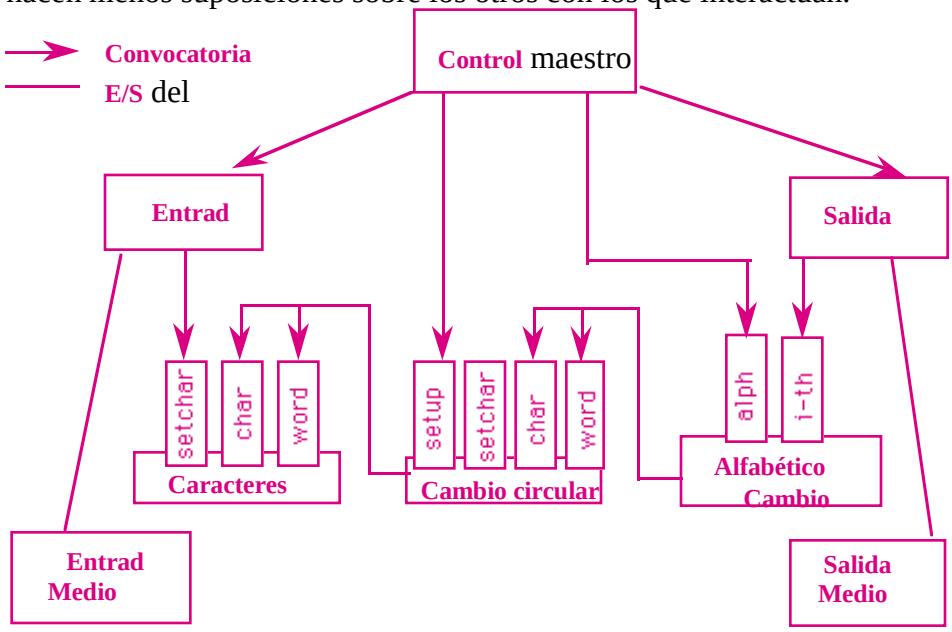

Figura 7: KWIC – Solución de tipo de datos abstracto

Por otro lado, como lo discutieron 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 conduzcan a penalizaciones de 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. En primer lugar, 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 shift. Esto le permite producir desplazamientos circulares (en un almacén de datos compartido abstracto independiente). Esto a su vez hace que el alfabetizador sea invocado implícitamente para que pueda alfabetizar las líneas.

Esta solución admite fácilmente mejoras funcionales en el sistema: se pueden conectar módulos adicionales al sistema registrándolos para ser invocados 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 datos. También se admite la reutilización, ya que los módulos invocados implícitamente solo dependen de la existencia de ciertos eventos activados externamente.

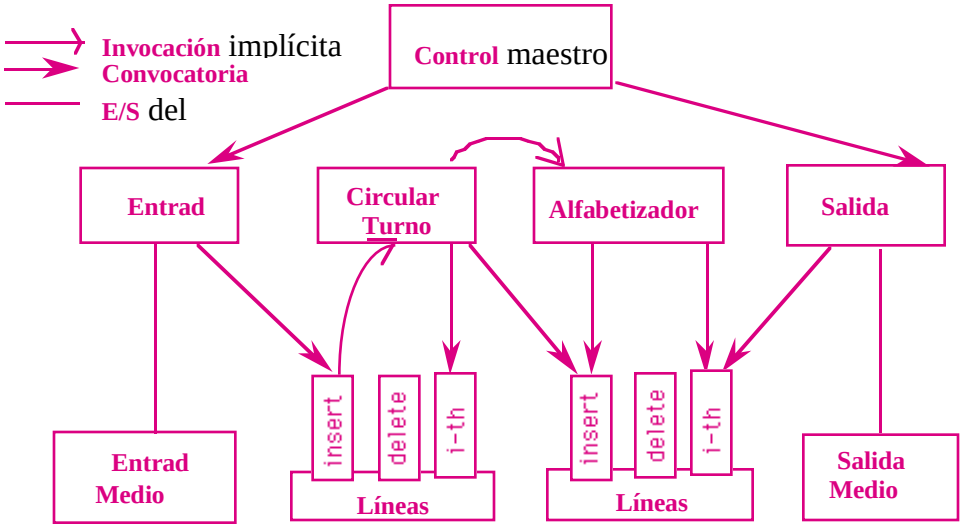

Figura 8: KWIC – Solución de invocación implícita

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, alfabetización 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 que calcular. El intercambio de datos entre filtros se limita estrictamente a los transmitidos en 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 que espera). Las nuevas funciones se agregan fácilmente al sistema insertando filtros en el punto apropiado de la secuencia de procesamiento. En tercer lugar, admite la facilidad de modificación, ya que los filtros son lógicamente independientes de otros filtros.

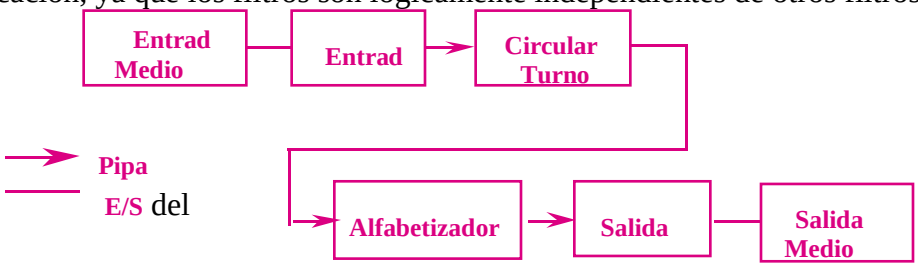

Figura 9: KWIC – Solución de tuberías y filtros

Por otro lado, tiene una serie de inconvenientes. En primer lugar, es prácticamente 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 su uso del 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 incluir la consideración de una serie de factores relacionados con el uso previsto del sistema: por ejemplo, si es por lotes o interactivo, requiere muchas actualizaciones o requiere muchas consultas, etc.

La Figura 10 proporciona una aproximación a dicho análisis, basada en la discusión de los estilos arquitectónicos introducidos anteriormente. Como señaló Parnas, la solución de datos compartidos es particularmente débil en su soporte para cambios en el algoritmo de procesamiento general, representaciones de datos y 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 tipo 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.

Compartido

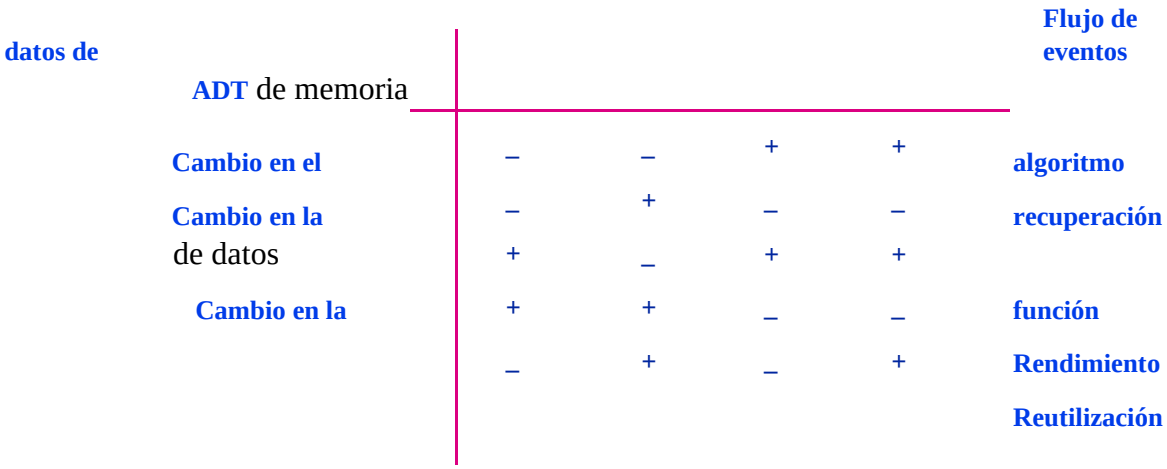

Figura 10: KWIC – Comparación de soluciones

La solución de invocación implícita es particularmente buena para agregar nuevas funcionalidades. Sin embargo, adolece de algunos de los problemas del enfoque de datos compartidos: escaso apoyo 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 reutilización. Por otro lado, las decisiones sobre la representación de datos se conectarán a los supuestos sobre el tipo de datos que se transmiten a lo largo de las tuberías. Además, dependiendo del formato de intercambio, puede haber una sobrecarga adicional involucrada en el análisis y la desconexión de los datos en tuberías.

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 Laboratorio de Investigación Informática 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 toma muestras de señales eléctricas y muestra imágenes (llamadas rastros) de ellas en una pantalla. Además, los osciloscopios realizan mediciones en las señales y también las muestran en la pantalla. Mientras que los osciloscopios fueron una vez simples dispositivos analógicos que involucraban poco software, los osciloscopios modernos se basan principalmente en la tecnología digital y tienen un software bastante complejo. No es raro que un osciloscopio moderno realice docenas de mediciones, suministre megabytes de almacenamiento interno, interactúe con una red de estaciones de trabajo y otros instrumentos, y proporcione una interfaz de usuario sofisticada que incluye una pantalla táctil con menús, instalaciones de ayuda incorporadas 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 osciloscopios fueron construidos por diferentes divisiones de productos, 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 nuevo osciloscopio generalmente requería un rediseño desde cero para acomodar 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 los de interfaz cambiaban cada vez más rápidamente. Además, se percibió la necesidad de abordar los "mercados especializados". Para ello tendría que ser posible adaptar un instrumento de uso general a un conjunto específico de usos.

En segundo lugar, hubo problemas de rendimiento crecientes 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 provocaba retrasos entre la solicitud de un nuevo modo por parte del usuario 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 específica de dominio que formó la base de la próxima generación de osciloscopios Tektronix. Desde entonces, el marco se ha ampliado y adaptado para dar cabida a una clase más amplia de sistema, mientras que al mismo tiempo se adapta 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 llevó a una aclaración de los tipos de datos utilizados en los osciloscopios: formas de onda, señales, mediciones, modos de disparo, etc. (Consulte la figura 11.)

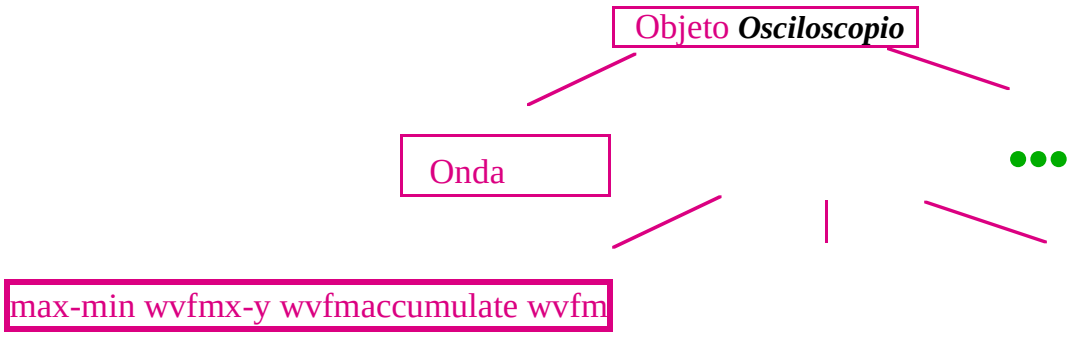

Figura 11: Osciloscopios – Un modelo orientado a objetos

Si bien este fue un ejercicio útil, estuvo muy lejos de producir los resultados esperados. Aunque se identificaron muchos tipos de datos, no hubo un modelo general que explicara cómo encajaban los tipos. Esto llevó a confusión sobre la partición de la funcionalidad. Por ejemplo, ¿deberían las mediciones asociarse con los tipos de datos que se miden o representarse externamente? ¿Con qué objetos debe 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 entran en el osciloscopio. Estas funciones se implementan normalmente 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 la forma 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 digitalizadas y mediciones 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 debían mostrarse en pantalla. (Consulte la figura 12.)

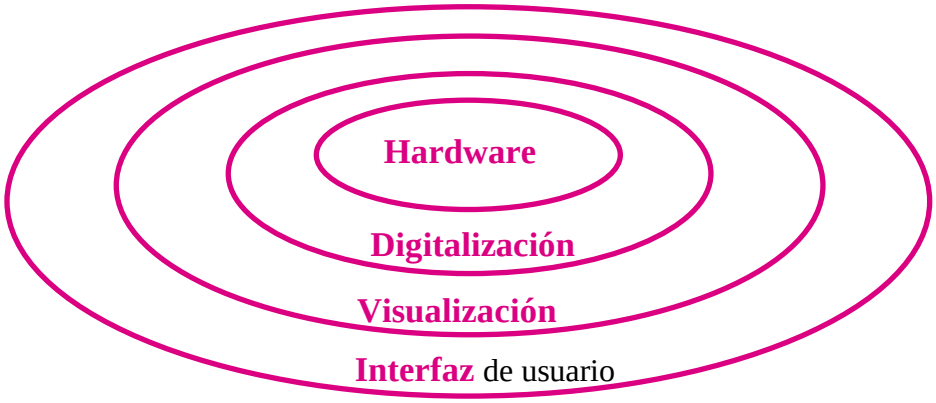

Figura 12: Osciloscopios: un modelo en capas

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 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 las representaciones visuales. Pero en la práctica, los usuarios reales de osciloscopios necesitan afectar directamente las funciones en todas las capas, como configurar 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 condicionar señales externas. Los transformadores de adquisición derivan formas de onda digitalizadas de estas señales. Los transformadores de pantalla convierten estas formas de onda en datos visuales. (Consulte la figura 13.)

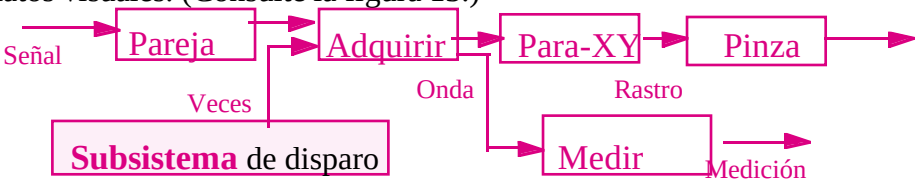

Figura 13: Osciloscopios: un modelo de tubería y filtro

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

Un modelo modificado de tubería y filtro

La cuarta solución tenía en cuenta las entradas del usuario asociando con cada filtro una interfaz de control que permite a una entidad externa establecer parámetros de operación para el filtro. Por ejemplo, el filtro de adquisición puede tener parámetros que determinan 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.

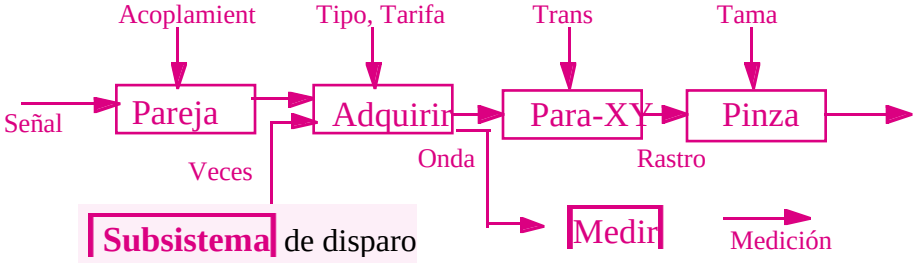

Figura 14: Osciloscopios: un modelo modificado de tubería y filtro

La introducción de una interfaz de control resuelve una gran parte del problema de la interfaz de usuario. En primer lugar, proporciona una colección de ajustes que determinan qué aspectos del osciloscopio pueden ser modificados dinámicamente por el usuario. También explica cómo se pueden realizar 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 comunica realmente 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 los parámetros de control. Esto permitió a los diseñadores cambiar la implementación del software y hardware de procesamiento de señales sin afectar una 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 tuvo algunos problemas. El problema más importante fue que el modelo computacional de tuberías y filtros 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 formas de onda cada vez que las procesan. Además, diferentes filtros pueden funcionar a velocidades radicalmente diferentes: es inaceptable ralentizar 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 permitieron que los datos se procesaran sin copiar. Otros permitieron que los datos fueran ignorados por filtros lentos si estaban en medio del procesamiento de otros datos. Estas tuberías adicionales aumentaron el vocabulario estilístico y permitieron que los cálculos de tubería / filtro se adaptaran más directamente a las necesidades de rendimiento del producto.

Resumen

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

4.3. Caso 3: Una nueva visión de los compiladores

La arquitectura de un sistema puede cambiar en respuesta a mejoras en la 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 organización de un compilador se dibujaba típicamente como en la Figura 15. El texto entra en el extremo izquierdo y se transforma de varias maneras: en flujo de tokens léxicos, árbol de análisis, código intermedio, antes de emerger como código máquina a la derecha. A menudo nos referimos a este modelo de compilación como una canalización, a pesar de que estaba (al menos originalmente) más cerca de una arquitectura secuencial por lotes en la que cada transformación ("pass") se completaba antes de que comenzara la siguiente.

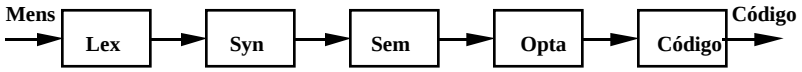

Figura 15: Modelo de compilador tradicional

De hecho, incluso la versión secuencial por lotes de este modelo no era completamente precisa. La mayoría de los compiladores crearon una tabla de símbolos separada durante el análisis léxico y la usaron o actualizaron durante las pasadas posteriores. No era parte de los datos que fluían de un paso a otro, sino que existía fuera de todos los pases. Por lo tanto, la estructura del sistema se dibujó más correctamente como en la Figura 16.

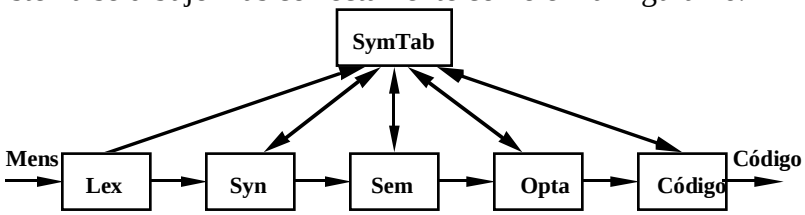

Figura 16: Modelo de compilador tradicional con tabla de símbolos compartidos

Con el paso del tiempo, la tecnología del compilador se hizo más sofisticada. Los algoritmos y representaciones de compilación se hicieron más complejos, y cada vez más atención se dirigió a la representación intermedia del programa durante la compilación. La mejora de la comprensión teórica, como los grammers 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 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 siguió siendo sustancialmente una estructura creciente en todo momento. Sin embargo, continuamos (a veces hasta el presente) modelando el compilador con flujo de datos secuenciales como en la Figura 17.

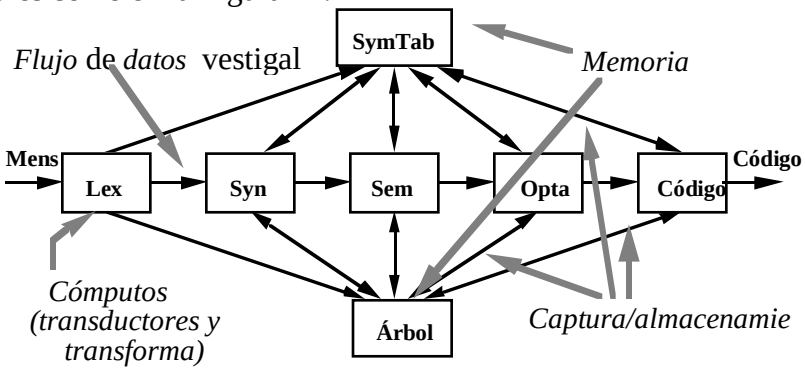

Figura 17: Compilador canónico moderno

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

en un repositorio, y esa es de hecho una forma más apropiada de pensar en un compilador de esta clase.

Afortunadamente, esta nueva visión 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.

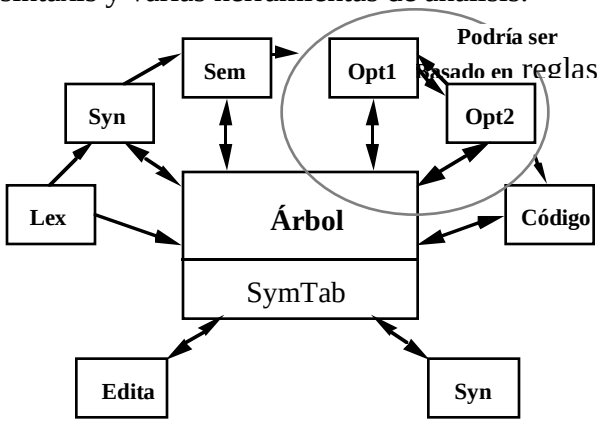

Figura 18: Compilador canónico, revisado

Tenga en cuenta que este repositorio se asemeja a una pizarra en algunos aspectos y difiere en otros. Al igual que una pizarra, la información del cálculo se encuentra centralmente y es operada por cálculos 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 entrante, 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 procesos distribuido para procesos de producción química [43]. Las capacidades de control de procesos van desde simples bucles de control que controlan la presión, el flujo o los niveles hasta estrategias complejas que involucran bucles de control interrelacionados. Se prevén disposiciones para la integración con la gestión de la planta y los sistemas de información en apoyo de la fabricación integrada por ordenador. La arquitectura del sistema integra el control de procesos con la gestión de la planta y los sistemas de información en una jerarquía de capas de 5 niveles. La figura 19 muestra esta jerarquía: el lado derecho es la vista de software y el lado izquierdo es la vista de hardware.

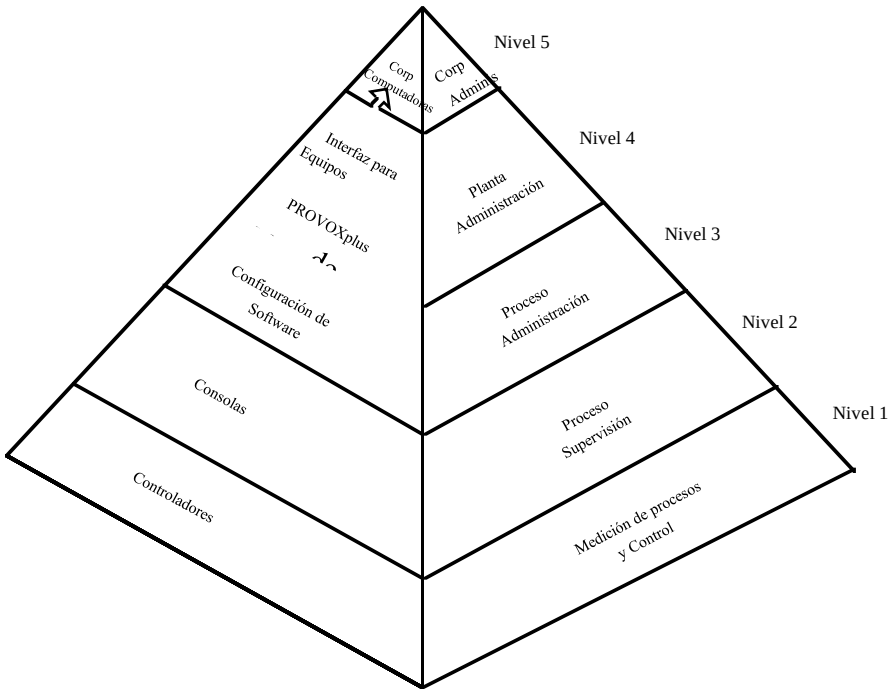

Figura 19: PROVOX – Nivel superior jerárquico

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

- Nivel 1: Medición y control del proceso: ajuste directo de los elementos de control final.
- Nivel 2: Supervisión de procesos: consola de operaciones para la monitorización y control Nivel 1.
- Nivel 3: Gestión de procesos: automatización de plantas basada en computadora, incluidos informes de gestión, estrategias de optimización y orientación para la consola de operaciones.
- Niveles 4 y 5: Gestión de planta y corporativa: funciones de nivel superior como contabilidad de costes, control de inventario y procesamiento / programación de pedidos.

Se requieren diferentes tipos de computación y tiempos de 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 convencionales de repositorio de procesamiento de datos. Para los propósitos presentes basta con examinar el modelo orientado a objetos del Nivel 2 y los repositorios de los Niveles 4 y 5.

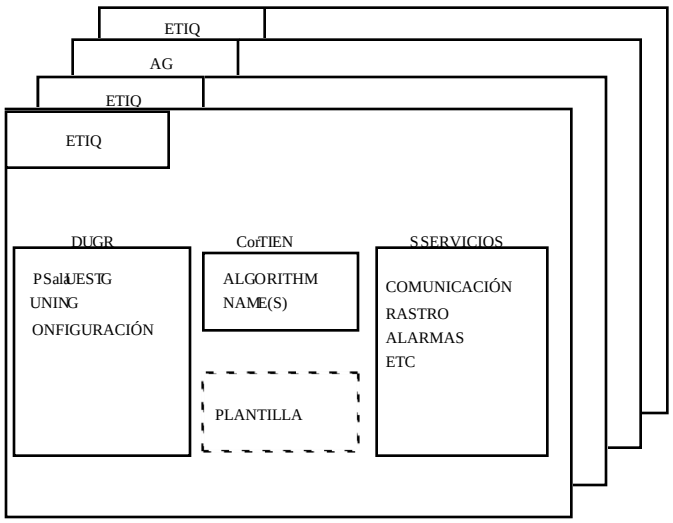

Figura 20: PROVOX – Elaboración orientada a objetos

Para las funciones de control y monitoreo del Nivel 2, PROVOX utiliza un conjunto de puntos u loci de control de procesos. La figura 20 muestra la forma canónica de una definición de punto; Siete formularios especializados admiten los tipos más comunes de control. 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 de funcionamiento, incluido el valor actual del proceso, el punto de consigna (valor objetivo), la salida de la válvula y el modo (automático o manual). Ajuste de parámetros, como puntos de disparo de ganancia, reinicio, derivados y alarma. Parámetros de configuración, incluyendo etiqueta (nombre) y 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 procedimiento como algoritmos de control, conexiones de comunicación, servicios de informes e instalaciones de seguimiento. Una colección de puntos implementa la estrategia de control de proceso 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 en un tanque, el valor actual de un punto que detecta el nivel del tanque reflejará este cambio). Aunque la comunicación a través del estado de proceso se desvía del control habitual de los objetos basado en procedimientos o mensajes, los puntos son conceptualmente muy parecidos a los objetos en su encapsulación de información esencial de estado y acción.

Los informes de los puntos aparecen como transacciones de entrada a los procesos de recopilación y análisis de datos en niveles de diseño superiores. La organización de los puntos en procesos de control puede ser definida por el diseñador para que coincida con la estrategia de control de procesos. Estos pueden agregarse 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 serían controlados por operadores individuales).

PROVOX establece disposiciones para la integración con la gestión de la planta y los sistemas comerciales en los niveles 4 y 5. La selección de esos sistemas es a menudo independiente del diseño del control del proceso; PROVOX no proporciona sistemas MIS directamente, pero sí proporciona la integración de un equipo host convencional con la gestión de bases de datos convencional. Las instalaciones de recopilación de datos del Nivel 3, las instalaciones de presentación 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. Dichas bases de datos se diseñan comúnmente como repositorios, con funciones de procesamiento de transacciones que admiten 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 son a menudo demasiado intrincadas para permitir una estratificación estricta.

4.5.Caso 5: Un intérprete que utiliza 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 capturar las técnicas de resolución de problemas como conjuntos de reglas de acción-situación cuya ejecución o activación se secuencia en respuesta a las condiciones del cálculo en lugar de por un esquema predeterminado. Dado que estas reglas no son directamente ejecutables por los ordenadores disponibles, deben proporcionarse sistemas para interpretarlas. Hayes-Roth examinó 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 controlado por 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 el selector de elementos de datos
- El *estado actual del programa* que se ejecuta en la máquina virtual, en este caso la memoria de trabajo.

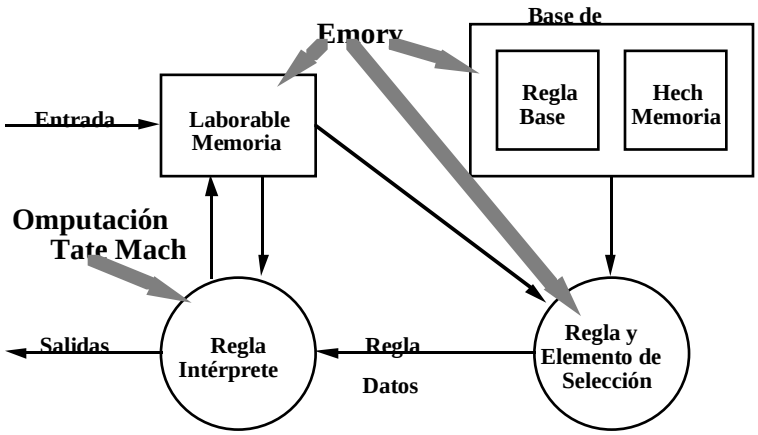

Figura 21: Sistema básico basado en reglas

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 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 entender. 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 expande con el idioma intérprete (es decir, el motor de interpretación del sistema basado en reglas se implementa como un intérprete controlado por tablas), con procedimientos de control que desempeñan el papel del pseudocódigo que se ejecutará y la 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 las reglas y los hechos activos en activaciones priorizadas; En esta tubería, el tercer filtro ("nominadores") también utiliza una base de datos fija de metareglas.
- La memoria de trabajo no se elabora más.

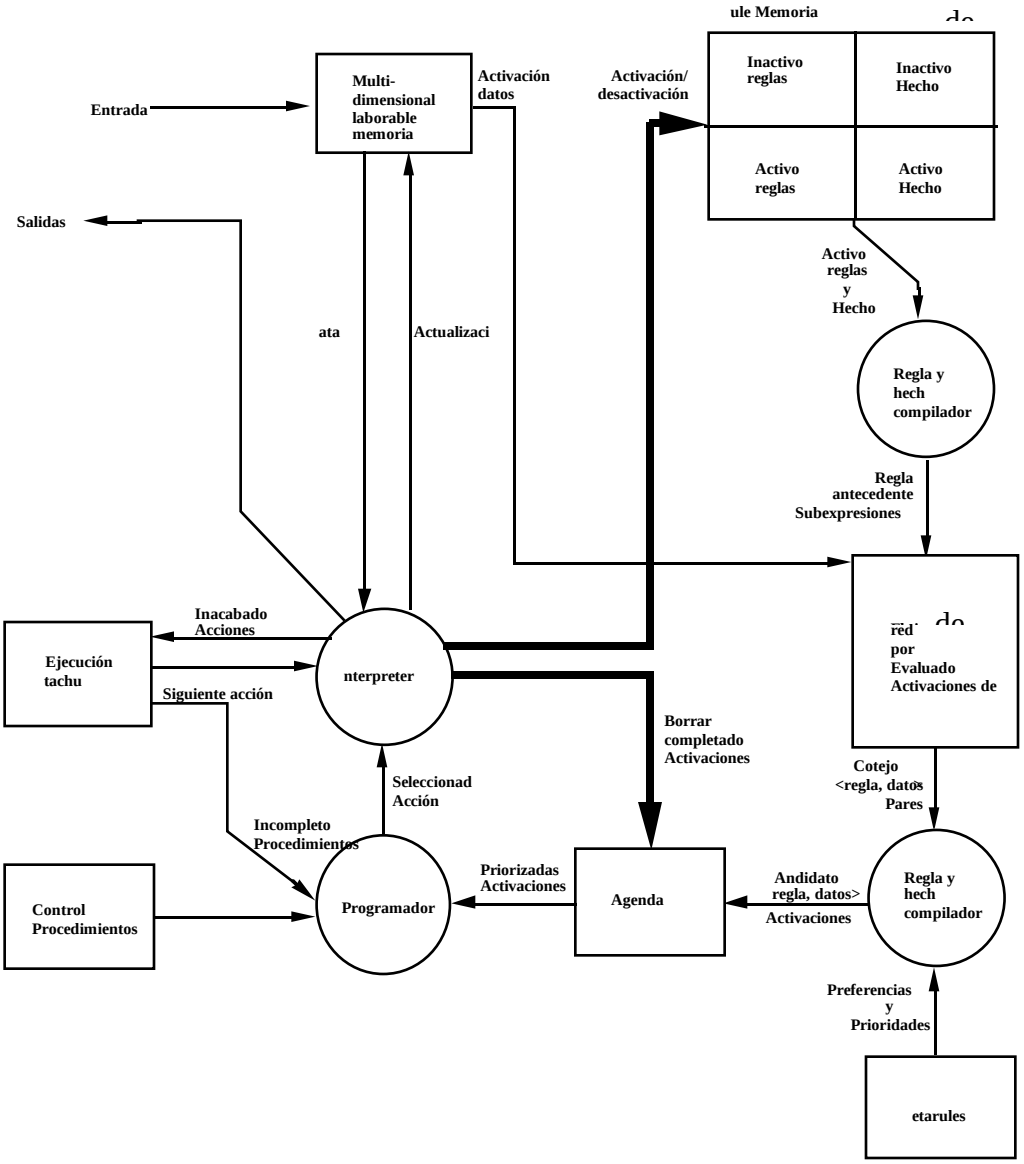

Figura 23: Sistema sofisticado basado en reglas

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

Este ejemplo ilustra dos puntos. En primer lugar, 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 idiomas que se interpretan. Si el diseño se presenta de esta manera, el concepto original se conserva 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 modismos.

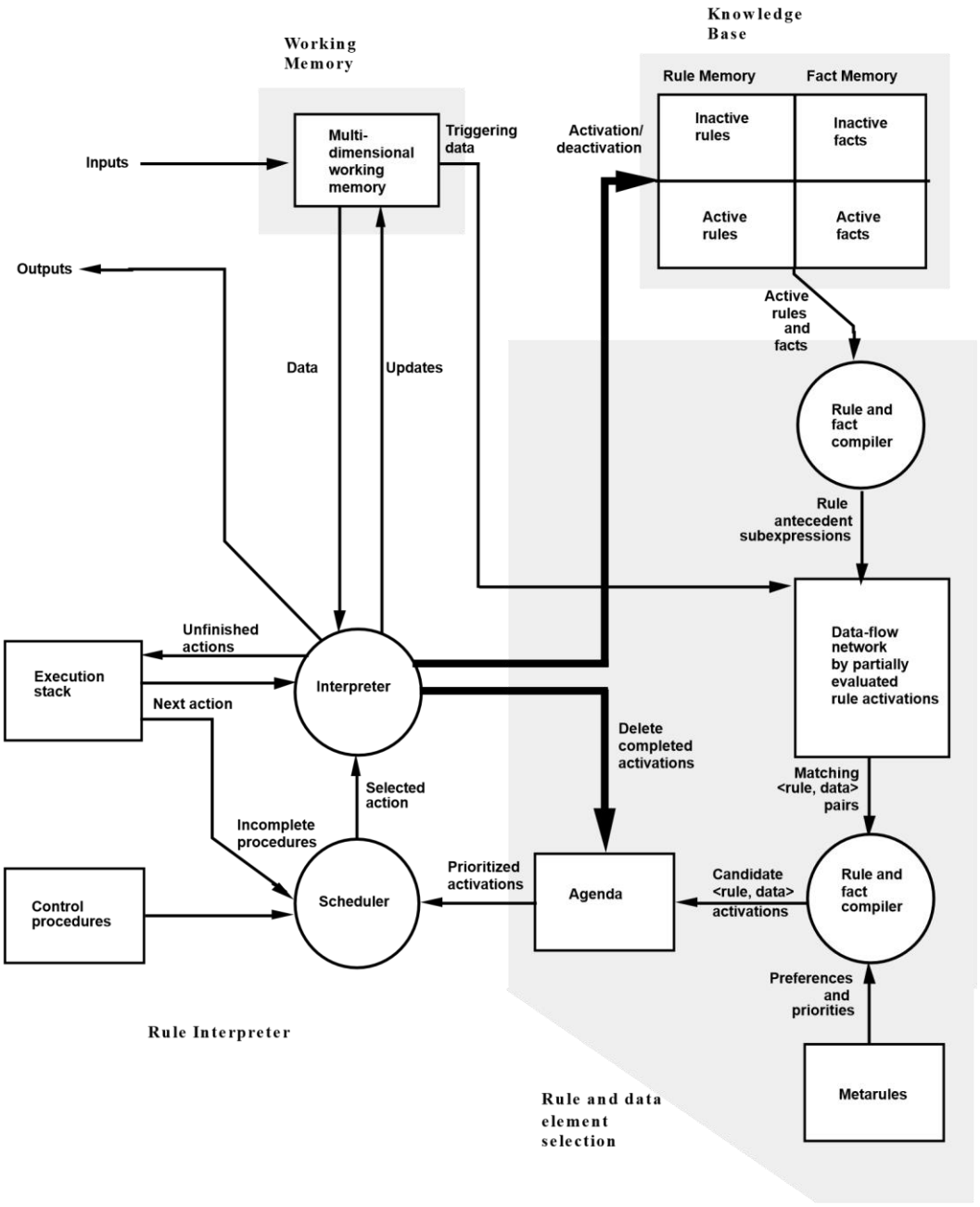

Figura 23: Sistema simplificado y sofisticado basado en reglas

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 se determinan 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: Una pizarra reformulada globalmente como intérprete

El modelo de pizarra para la resolución de problemas es un caso especial altamente estructurado de resolución oportunista de problemas. En este modelo, el espacio de

soluciones se organiza en varias jerarquías dependientes de la aplicación y el conocimiento del dominio se divide en módulos independientes de conocimiento que operan sobre el conocimiento dentro y entre los niveles [34]. La Figura 4 muestra la arquitectura básica de un sistema de pizarra y describe sus tres partes principales: fuentes de conocimiento, la estructura de datos de pizarra y control.

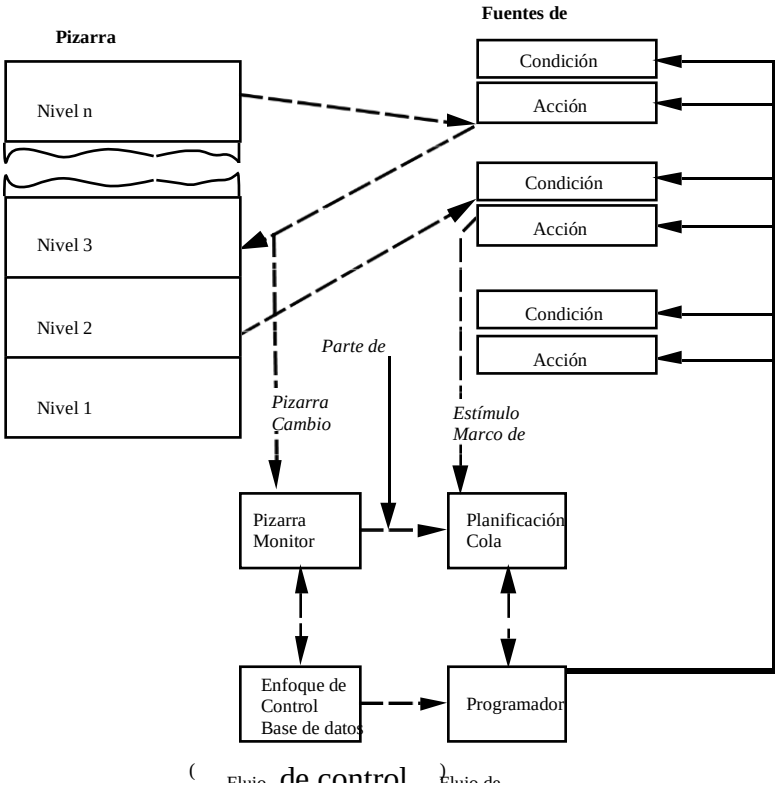

Figura 24: Rumores-II

El primer sistema de pizarra importante fue el sistema de reconocimiento de voz HEARSAY-II. El esquema de Nii de la arquitectura HEARSAY-II aparece como la Figura 24. La estructura de pizarra es una jerarquía de seis a ocho niveles en la que cada nivel abstrae información sobre su nivel inferior adyacente y los elementos de pizarra representan hipótesis sobre la interpretación de una expresión. Las fuentes de conocimiento corresponden a tareas tales como segmentar la señal en bruto, identificar fonemas, generar candidatos a palabras, hipotetizar segmentos sintácticos y proponer 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 relevantes de la pizarra y genera otros nuevos. El componente de control se realiza como un monitor de pizarra y un programador; El programador 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 controlar la condición activada, 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 Blackboard.

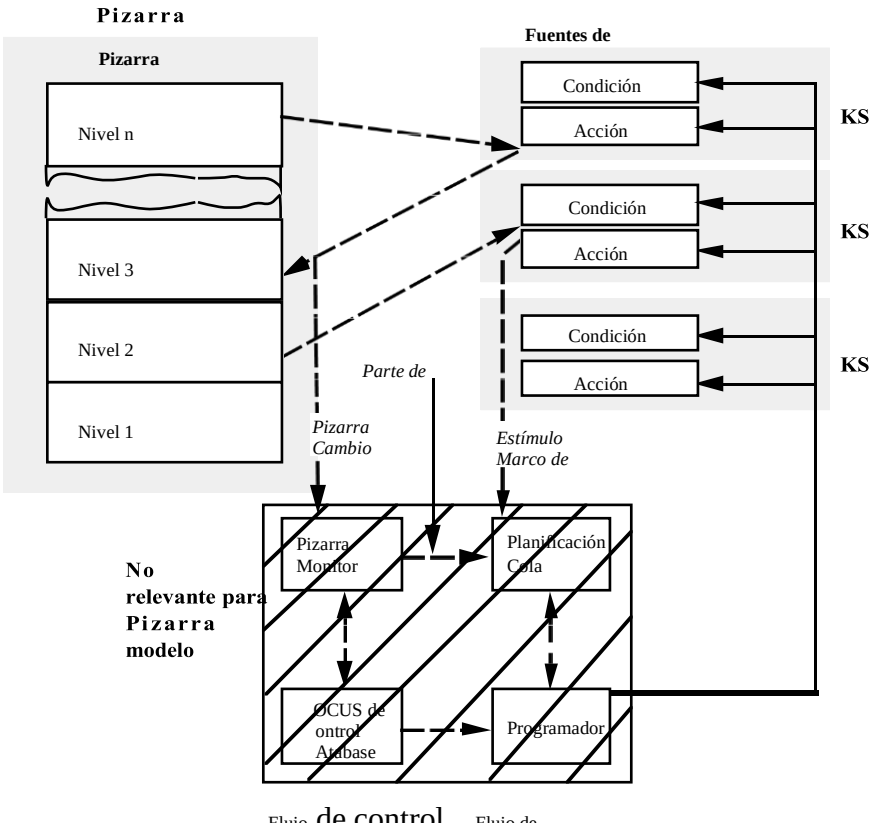

Figura 25: Vista de pizarra de Hearsay-II

La Figura 24 no sólo 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 ha sido realizada por un intérprete utilizando 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, el foco de la base de datos de control y el programador, 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 las condiciones determina las prioridades, las condiciones contribuyen a la selección de reglas, así como a la formación de pseudocódigo.

Aquí vemos un sistema diseñado inicialmente con un modelo (pizarra, una forma especial de repositorio), 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.

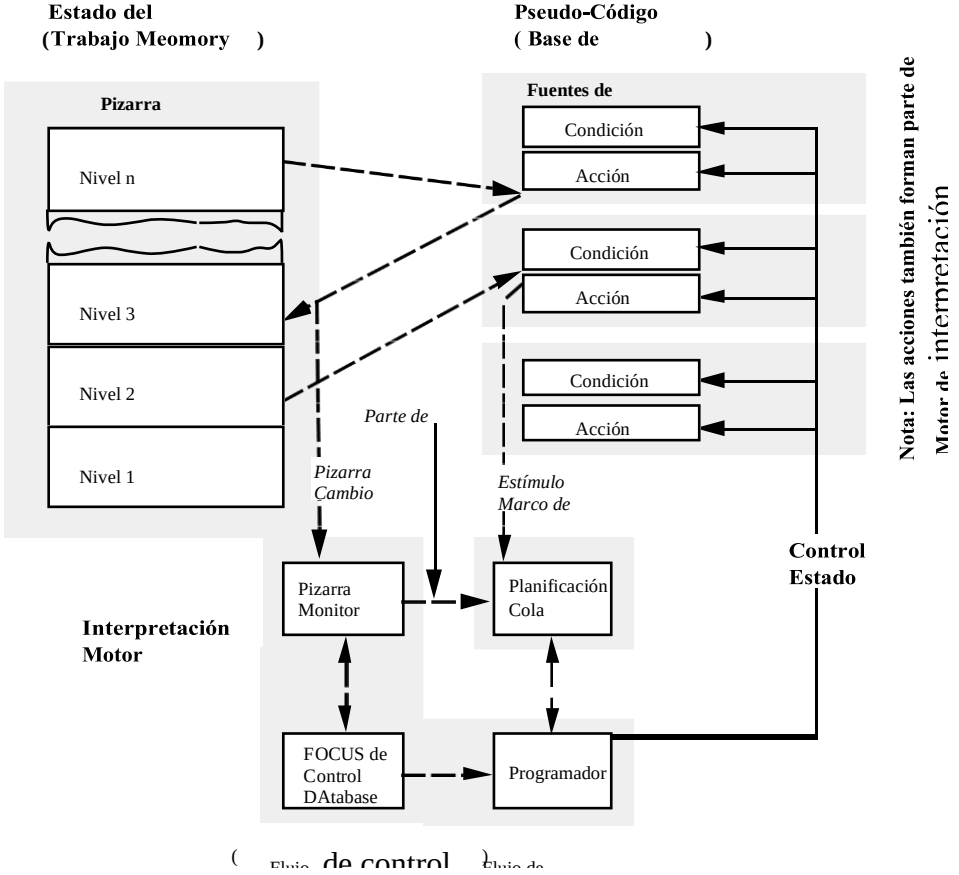

Figura 26: Vista del intérprete de Hearsay-II

5.Pasado, presente y futuro

Hemos esbozado una serie de estilos arquitectónicos y hemos mostrado cómo se pueden aplicar y adaptar a sistemas de software específicos. Esperamos que esto haya convencido al lector de que el análisis y el 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, incluyendo:

- 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 la academia. 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 en la mano, el diseño a nivel de arquitectura de software puede proporcionar un beneficio directo y sustancial a la práctica de la ingeniería de software.

Agradecimientos

Reconocemos con gratitud 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, "Experiencew ith a course on architectures for software systems", en *Proceedings of the Sixth SEI Conference on Software Engineering Education*, Springer-Verlag, LNCS 376, octubre de 1992.

[2] M. Shaw, "Towards higher-level abstractions for software systems", en *Data & Knowledge Engineering*, vol. 5, pp. 119-128, North Holland: Elsevier Science Publishers B.V., 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), pp. 158-165, 25-26 de octubre de 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, "A formal approach to software architectures", en *Proceedings of IFIP'92* (J. van Leeuwen, ed.), Elsevier Science Publishers B.V., septiembre de 1992.

[6] D. Garlan y D. Notkin, "Formalizing design spaces: Implicit invocation mechanisms", en*VDM'91: Formal Software Development Methods.* (Noordwijkerhout, Países Bajos), págs. 31 a 44, Springer-Verlag, LNCS 551, octubre de 1991.

[7] D. Garlan, G. E. Kaiser, y D. Notkin, "Using tool abstraction to compose systems," *IEEE Computer*, vol. 25, junio de 1992.

[8] A. Z. Spector *et al.*, "Camelot: A distributed transaction facility for Mach and the Internet an interim report," Tech. CMU-CS-87-129, Universidad Carnegie Mellon, junio de 1987.

[9] M. Fridrich y W. Older, "Helix: The architecture of the XMS distributed file system", *IEEE Software*, vol. 2, pp. 21-29, mayo de 1985.

[10] M. A. Linton, "Distributed management of a software database," *IEEE Software*, vol. 4, pp. 70-76, noviembre de 1987.

[11] V. Seshadri et al., "Sematic analysis in a concurrent compiler", en Proceedings of ACM SIGPLAN *'88 Conference on Programming Language Design and Implementation, ACM SIGPLAN* Notices, 1988.

[12] M. C. Paulk, "The ARC Network: A case study," *IEEE Software*, vol. 2, pp. 61-69, mayo de 1985.

[13] M. Chen y R. J. Norman, "A framework for integrated case," IEEE Software, vol. 9, pp. 1822, marzo de 1992.

[14] Modelo de referencia NIST/ECMA para frameworks de entornos de ingeniería de software." NISTSpecial Publication 500-201, diciembre de 1991.

[15] R. W. Scheifler y J. Gettys, "The X window system," *AACM Transactions on Graphics*, vol. 5, pp. 79-109, abril de 1986.

[16] M. J. Bach, *The Design of the UNIX Operating System*, cap. 5.12, pp. 111-119. Serie de software, Prentice-Hall, 1986.

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

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

[19] M. R. Barbacci, C. B. Weinstock, y J. M. Wing, "Programming at the processor-memoryswitch level", en *Proceedings of the 10th International Conference on Software Engineering*, (Singapur), pp. 19-28, IEEE Computer Society Press, abril de 1988.

[20] G. E. Kaiser y D. Garlan, "Synthesizing programming environments from reusablefeatures," en Software Reusability (T. J. Biggerstaff y A. J. Perlis, eds.), vol. 2, ACM Press, 1989.

[21] W. Harrison, "RPDE: A framework for integrating tool fragments," IEEE Software, vol. 4, noviembre de 1987.

[22] C. Hewitt, "Planner: A language for proving theorems in robots", en *Proceedings of the First International Joint Conference in Artificial Intelligence,* 1969.

[23] S. P. Reiss, "Connecting tools using message passing in the field program developmentenvironment," *IEEE Software,* julio de 1990.

[24] C. Gerety, "HP Softbench: A new generation of software development tools," Tech. Rep.SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, noviembre de 1989.

[25] R. M. Balzer, "Living with the next generation operating system", en *Proceedings of the 4th World Computer Conference, septiembre de* 1986.

[26] G. Krasner y S. Pope, "A cookbook for using the model-view-controller user interfaceparadigm in Smalltalk-80," Journal of Object Oriented Programming, vol. 1, pp. 26-49, agosto/septiembre de 1988.

[27] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols y R. Pausch, "Descartes: Aprogramming-language approach to interactive display interfaces," *Proceedings of SIGPLAN '83: Symposium on Programming Language Issues in Software Systems, ACM SIGPLAN Notices*, vol. 18, pp. 100-111, junio de 1983.

[28] A. N. Habermann y D. S. Notkin, "Gandalf: Software development environments," *IEEE Transactions on Software Engineering,* vol. SE-12, págs. 1117-11-27, diciembre de 1986.

[29] A. N. Habermann, D. Garlan, y D. Notkin, "Generation of integrated task-specificsoftware environments," in *CMU Computer Science: A 25th Commemorative* (R. F. Rashid, ed.), Anthology Series, pp. 69-98, ACM Press, 1991.

[30] K. Sullivan y D. Notkin, "Reconciling environment integration and componentindependence," in *Proceedings of ACM SIGSOFT90: Fourth Symposium on Software Development Environments*, pp. 22-23, diciembre de 1990.

[31] G. R. McClain, ed., *Open Systems Interconnection Handbook.* Nueva York, NY: Intertext Publications McGraw-Hill Book Company, 1991.

[32] D. Batory y S. O'Malley, "The design and implementation of hierarchical softwaresystems using reusable components," Tech. Rep. TR-91-22, Department of Computer Science, University of Texas, Austin, June 1991.

[33] H. C. Lauer y E. H. Satterthwaite, "Impact of MESA on system design", en *Proceedings of the Third International Conference on Software Engineering*, (Atlanta, GA), pp. 174-175, IEEE Computer Society Press, mayo de 1979.

[34] H. P. Nii, "Blackboard systems 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*Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments, SIGSOFT* Software Engineering Notes, (Irvine, CA), pp. 183-192, diciembre de 1990.

[36] D. R. Barstow, H. E. Shrobe, y E. Sandewall, eds., *Interactive Programming Environments.* McGraw-Hill Book Co., 1984.

[37] G. R. Andrews, "Paradigms for process interaction in distributed programs," *ACM Computing Surveys*, vol. 23, pp. 49-90, marzo de 1991.

[38] A. Berson, *Arquitectura cliente/servidor*. McGraw-Hill, 1992.

[39] E. Mettala y M. H. Graham, eds., *The Domain-Specific Software Architecture Program.* Nº CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, junio de 1992.

[40] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231-274, 1987.

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

[42] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, vol. 15, págs. 1053-1058, diciembre de 1972.

[43] "PROVOX plus Instrumentation System: System overview," 1989.

[44] F. Hayes-Roth, 'Rule-based systems," Communications of the ACM, vol. 28, pp. 921-932, septiembre de 1985.