UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

SIUX: UNA PROPUESTA MÁS PARA DESARROLLAR SISTEMAS DE INFORMACIÓN

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A:
DARIO BAHENA TAPIA

DIRECTOR DE TESIS:
L.EN C.C. MANUEL ALBERTO SUGAWARA MURO

CO-DIRECTORA DE TESIS:
M.EN. C. ELISA VISO GUROVICH

2004

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

"SIUX: Una propuesta más para desarrollar sistemas de información"

realizado por Dario Bahena Tapia

con número de cuenta 093106575 , quien cubrió los créditos de la carrera de:Licenciatura en Ciencias de la Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis
Propietario L. en C. C. Manuel Alberto Sugawara Muro
Co-Directora de Tesis
Propietario M. en C. Elisa Viso Gurovich
Propietario M. en C. Javier García García
Suplente M. en C. María Guadalupe Elena Ibarguengoitia González
Suplente L. en C. C. Francisco Lorenzo Solsona Cruz

Consejo Departamental de Matemáticas

Dr. Francisco Hernández Quiroz

SIUX: Una propuesta más, para desarrollar Sistemas de Información.

Dario Bahena Tapia, dario@ciencias.unam.mx

25 de Noviembre del 2004

Índice general

Capítulo 1

Introducción

De todos los posibles tipos de sistemas computacionales que uno podría construir, definitivamente los llamados *Sistemas de Información*, son los más comunes (hablando del ámbito comercial). De hecho, para las personas que estudiamos alguna carrera relacionada con la computación, el trabajar en el llamado "mundo real", se suele traducir como dedicarse a la construcción de este tipo de sistemas.

Durante el desarrollo de estos sistemas, uno esperaría poder echar mano de las herramientas aprendidas en el pasado; en especial, del conocimiento técnico no perecedero; es decir, aquél que está suficientemente divorciado de la tecnología, como para ser no volátil. De hecho, en las carreras de informática se suele decir a los alumnos (o por lo menos se les debería decir), que no dependan de ningún producto en especial. En cambio, se hace énfasis en aprender aquéllos aspectos trascendentes, en los que se basan dichos productos; en este rubro podrían entrar detalles que, si bien todo mundo da por hechos básicos, en la práctica pocos suelen aplicarlos: lógica de programación paramétrica y patrones de diseño modular, serían dos claros ejemplos. También importante, tenemos al conocimiento de los estándares (como *TCP/IP*, *SQL* o *XML*), que a pesar de poseer distinta naturaleza del par de ejemplos mencionados, definitivamente merecen entrar en el baúl de conocimiento trascendente (aunque aquí podríamos tener caducidad, pero ésta suele ser bastante holgada).

Otro aspecto a considerar, es el peso que asignamos a cada aspecto del desarrollo: si bien es cierto que las reglas del negocio son un punto muy importante, también es cierto que su comprensión requiere de habilidades que, o bien se ejercitaron en la carrera, o bien se desarrollaron con la experiencia. No ocurre así con el aspecto tecnológico, del cual existen tantas posibilidades en nuestros días, que es fácil perderse o emplear mucho tiempo en decidirse al respecto. Motivado en experiencias personales (un tanto dolorosas, debo admitir), este trabajo pretende ofrecer una opción tecnológica más, para desarrollar Sistemas de Información (que de aquí en adelante, trata-

remos de abreviar como *SI*). Obviamente, también intentaremos justificar la propuesta. Aunque el enfoque está centrado en la tecnología, podemos apreciar que su influencia se extiende hasta el campo del diseño.

Continuando la línea inicial, la presente propuesta no hará propaganda para ningún producto en particular, pero sí respecto a estándares actuales (los cuáles pueden considerarse inmersos en la categoría de los aspectos no volátiles de la computación). Obviamente, para probar dichos estándares, se escogieron productos específicos que los implementaran e incluso, puede que se hayan usado características específicas de los mismos; sin embargo, éstas fueron las menos, comparadas con el porcentaje usado que recae exclusivamente en los estándares empleados. Entonces, el uso de los productos fue sólo con la intención de probar la factibilidad de la propuesta. Desgraciadamente, rara vez el empleo de estándares se suele traducir directamente en portabilidad de código. Los empleados en esta propuesta no son la excepción, aunque su empleo por lo menos hace más factible una tarea de migración.

El paso natural ahora, sería decir cuáles estándares estamos proponiendo; pues bien, se trata de los siguientes: *SQL99*, Servicios Web (*Web Services*) y Formas *XML*(*XForms*). Para mencionar brevemente en qué consiste cada uno (y/o para qué se usó en este trabajo), usaremos como marco la arquitectura clásica de los sistemas de información, misma que está estructurada en 3 capas que listamos en seguida, junto con la tecnología asociada:

Capa 1: Modelo de datos. Si pensamos en que la tarea de los Sistemas de Información consiste esencialmente en manipular datos, y que dichos datos poseen cierta estructura, en esta capa es donde se define esta estructura. La herramienta para hacerlo suele ser alguna implementación del estándar *SQL*, aunque a un nivel muy básico, donde sólo se usa al manejador de la base de datos como un simple contenedor (una excepción común de este hábito es el uso de las llaves primarias y secundarias).

Capa 2: Lógica del sistema. Las reglas del negocio, que permiten manipular los datos de la capa anterior, usualmente están implementadas en algún lenguaje de tercera generación como C, Cobol o Java; y son ofrecidos hacia el exterior a través de algún servidor de aplicaciones como Tuxedo, CICS, EJB, etc. Dichos servidores de aplicaciones ofrecen infraestructura común a los servicios que manipulan los datos, entre los cuales podemos encontrar: accesibilidad, balanceo de carga, seguridad, acceso homogéneo a recursos comunes (base de datos), etc.

Capa 3: Interfaz del Usuario. Mientras que las capas anteriores conforman lo que comúnmente consideramos el traspatio (*backend*), esta capa haría lo propio para el vestíbulo (*frontend*).

Aquí es donde el usuario final interactúa con el sistema en su conjunto, a través de interfaces gráficas (el modo texto no es muy favorecido), que piden servicios al *backend*, a través de la capa 2 del mismo. A diferencia de la capa anterior, aquí es más común el empleo de herramientas RAD (*Rapid Application Development*), que facilitan la generación de dichas interfaces (en lenguajes/ambientes como Visual Basic, Java, Uniface, etc.).

Esta es la arquitectura que comúnmente vemos o, incluso, que tratamos de implementar para los Sistemas de Información. Por muy sencilla u obvia que pueda parecer, en realidad no es tan fácil de materializar en forma estricta, y sin mucho temor a equivocarnos, podemos declarar que gran cantidad de sistemas de la vida real fallan en implementarla. Probablemente, el fracaso se deba a un mal diseño; y aquí es donde, en forma indirecta la tecnología juega un papel importante: la falta de dominio de ella puede implicar que dediquemos mucho tiempo a resolver problemas de bajo nivel (configuraciones, integraciones, etc.), y que descuidemos el corazón del sistema (análisis y diseño). Entre los vicios más comunes merece su mención la falta de fronteras claras entre las capas; y dentro de éstos, la inclusión de reglas del negocio dentro de la capa 3 (*frontend*) podría ser la más famosa.

Por ello es tan importante que, antes de embarcarnos en alguna empresa de construcción de sistemas, contemos con la infraestructura necesaria para su implementación. Y dicha infraestructura es precisamente nuestra apuesta, donde sus componentes clave no son productos sino los estándares previamente mencionados (aunque podría haber preferencias en los productos, gracias a su madurez en las implementaciones). La incrustación de la infraestructura en las capas de esta arquitectura es bastante natural:

Capa 1 (Modelo de Datos). *SQL99-DDL*: Lenguaje para Definición de Datos (*Data Definition Language*).

Capa 2 (Reglas del negocio). *SQL99-PSM*: Módulos almacenados permanentemente (conocidos como procesos almacenados, aunque el término en inglés es *Permanent Stored Modules*) + Web Services (para la y publicación de la funcionalidad hacia el *frontend* u otros sistemas).

Capa 3 (Interfaz de usuario). *XForms* (junto con algunos otros componentes necesarios para hacer accesibles las interfaces, probablemente en intranets o en internet).

Para mayor modularidad de nuestra propuesta, usaremos una arquitectura extendida, donde distinguiremos la publicación de la funcionalidad del *backend* hacia el exterior como otra capa,

quedando el esquema como sigue (anexamos a cada capa un brevario del estándar asociado, sus ventajas e implementación seleccionada):

Capa 1 (Modelo de datos). Estándar propuesto: *SQL99* (*DDL+PSM*). Ventajas: posibilidad de garantizar parte de la integridad de los datos, usando un paradigma declarativo; se disminuye así la tediosa tarea del la verificación imperativa de errores. Aunque el componente principal de esta capa sea la porción del estándar *SQL* denominada *DDL* , también usamos otra sección llamada *PSM* (mencionada abajo). El estándar define la interacción entre estas partes, y la misma se aprovecha para validar todas aquellas cosas que caen fuera del alcance de las opciones declarativas.

Implementación seleccionada del estándar: Oracle 9i DBMS.

Capa 2 (Reglas del negocio). Estándar propuesto: *SQL99-PSM.* Ventajas: liberar al programador del aburrido menester de mover datos desde y hacia la base de datos; uniformidad entre el lenguaje usado para implementar la lógica y el usado para definir los datos (Capa 1); acceso a un lenguaje de cuarta generación, especializado en procesamiento de datos; optimización de los tiempos de ejecución y desarrollo.

Implementación seleccionada del estándar: Oracle 9i DBMS.

Capa 3 (Interfaz del backend). Estándar propuesto: Web Services. Ventajas: Abrir las puertas hacia el exterior, poniendo disponible la funcionalidad a prácticamente cualquier plataforma, sin discriminar sistema operativo o lenguaje alguno (que cuenten con implementaciones del estándar, es suficiente); ocultamiento eficiente de los menesteres de implementación del *backend*, expresividad en los tipos de datos que pueden recibir y arrojar los servicios del *backend* (*XML*).

Implementación seleccionada del estándar: Axis (subproyecto de la *Apache Software Foundation*, localizable en http://ws.apache.org/axis).

Capa 4 (Interfaz de usuario). Estándar propuesto: *XForms* (+ infraestructura web). Ventajas: Programación declarativa de las interfaces de usuario, lo cual a su vez habilita la posibilidad de generarlas automáticamente, o por lo menos de forma semiautomática; separación real entre los datos, el control y la presentación de la interfaz; varias opciones tecnológicas para ofrecer la interfaz de usuario (web,ventanas tradicionales,consola sólo texto,etc.).

Implementación seleccionada del estándar: Chiba (disponible en el repositorio http://SourceForge.net).

Esta arquitectura es el componente central de la propuesta, y en torno a ella gira la infraestructura que se desarrolló. En su mayor parte, ésta consiste en programas y/o configuraciones, que permiten ensamblar las implementaciones de los estándares involucrados, y en menor medida también sugiere ciertos patrones de diseño (donde la homogeneidad suele ser la prioridad).

Y ahora es tiempo para un poco de proselitismo. El autor de estas líneas considera importante mencionar su interés y admiración por gran parte del software libre (o de código abierto), y ello se refleja en las implementaciones seleccionadas para las capas 3 y 4, no así para las dos primeras. La razón de ello radica en que se consideró más importante ser fiel a los estándares que a los productos (siguiendo la premisa mencionada al principio); y en ese sentido, tenemos que aceptar que el lenguaje PL/*SQL* de Oracle está más cerca del estándar *SQL99-PSM*, que su análogo en *PostgreSQL* (llamado por cierto, *PGPLSQL*). También podríamos mencionar que Oracle tiene mejor manejo de errores (particularmente de las excepciones), característica que se valoró mucho para el presente trabajo; y que en general el lenguaje de Oracle ofrece mayor madurez.

De haber contado Postgre*SQL* con las características requeridas por esta propuesta, definitivamente hubiera sido la opción seleccionada. Siguiendo con la propaganda, un excelente y muy interesante proyecto de software libre podría consistir en llenar estos huecos en Postgre*SQL*, y permitir a esta propuesta ofrecer una posibilidad de implementación totalmente libre (haciendo referencia al código de sus componentes).

Terminada la nota proselitista, continuemos con esta introducción. La infraestructura que intentamos presentar se irá explicando en las siguientes secciones, dividida de tal manera que las partes correspondan con cada una de las capas de la arquitectura esbozada. Este proceso de explicación se hará en función de un ejemplo muy sencillo, algo así como el "hola mundo" de la propuesta: un mantenimiento de datos, y que consiste del mantenimiento de una tabla, desde su definición hasta la presentación de los servicios al usuario.

Se optó por este problema ejemplo, debido a su sencillez y relativa fama; prácticamente todo *SI* tiene mantenimientos incluidos, así que es altamente probable que el lector ya se haya visto en la tediosa necesidad de programar alguno (por lo menos una vez en su vida). Estas características, nos permitirán concentrarnos en los aspectos tecnológicos de la propuesta; digamos que este problema es algo parecido al "hola mundo" del lenguaje de programación C (tradición imitada por otros lenguajes imperativos). De hecho, hay más que decir acerca del mantenimiento de tablas: los patrones que presenta se pueden encontrar en gran parte de los *SI*. De hecho su solución nos lleva de paseo por las cuatro capas de la arquitectura mostrada, un recorrido mínimo que puede ser fácilmente extendido con datos y las reglas de negocio, para darnos lugar a *SI* de

la vida real (aunque en este trabajo, fue sufiente con incluir el ejemplo de mantenimiento de tabla).

Mientras se busca un lugar decente para albergar el código resultante de este trabajo (como *sourceforge.net*), el lector interesado puede contactar al autor para pedir una copia (a la dirección de correo electrónico: *dario@ciencias.unam.mx*). Todos los programas, ejemplos, utilerías y demás, son de código abierto.

Capítulo 2

Capa 1: Modelo de datos (SQL99-ddl)

En esta sección platicaremos de nuestra propuesta para el diseño y tecnología, que deberían usarse en la capa 1 de nuestra arquitectura, que es donde radica el modelo de datos.

2.1. Paradigmas de programación

Tradicionalmente, los modelos de datos de los Sistemas de Información, se implementan en manejadores de bases de datos relacionales; desde hace algunos años, los más famosos de estos sistemas también soportan algunas características orientadas a objetos e incluso han comenzado las interacciones con el nuevo vecino *XML*. Pero de momento nos basta con el modelo relacional; antes de pensar en la incorporación de nuevas tecnologías y paradigmas, ¿por qué no darle una última oportunidad al modelo relacional y su viejo compañero, el paradigma imperativo?, y tratar de exprimirlos al máximo, de ver hasta dónde nos pueden llevar por sí solos.

No es raro caer en la tentación de pensar que el simple hecho de usar la tecnología o metodología de moda, automáticamente se traducirá en mejores sistemas. Para un vendedor, este tipo de argumentación es el pan nuestro de cada día; pero como desarrolladores o clientes, antes de pensar en cambiar de tecnología conviene meditar si exploramos las opciones que nos ofrece, o simplemente, si el problema más que tecnológico es de índole humana (malos análisis, diseños o implementaciones). El último caso seguramente es el más probable, pero también conviene explorar las herramientas que tenemos a la mano: entre mejor conozcamos a nuestras herramientas, mayor probabilidad tendremos de seleccionar la correcta para una situación determinada. Inclusive, conocer nuestra plataforma actual, nos permitió visualizar mejor una integración con la tecnología emergente.

Entonces, en esta propuesta nos conformaremos con el Modelo Relacional de Datos: a lo largo de toda la infraestructura y las instancias de *SI* que pudiera parir, sólo contaremos con los siguientes tipos de datos: simples (alfanuméricos, numéricos y fechas), registros de estos tipos simples y sucesiones de estos registros. Con eso nos basta. En las primeras dos capas de nuestra arquitectura, podría ser más obvia esta situación que en las dos últimas, mismas que incluyen la aparición de estándares *XML*. Pero esta inclusión sólo se debe de pensar como una nueva piel para nuestros viejos datos relacionales, que permite su integración en un escenario más actual; pero nada más. No confundir con el siguiente paso, que sería no conformarse con el paradigma relacional imperativo y usar desde la base de datos Orientación a Objetos o estructuras de datos arbitrarias (con el estándar *SQLX* por ejemplo, que permite una integración de *SQL* con *XML*).

Aquí conviene hacer un breve paréntesis, porque le hemos hecho trampa al lector (esperando que se haya dado cuenta). Hemos mencionado afinidad o alabanzas hacia dos paradigmas computacionales, que podrían pensarse en conflicto: el imperativo y el declarativo. En la introducción mencionamos que parte de las ventajas de aprovechar ciertas partes de los estándares *SQL* o *XForms*, consiste en usar un paradigma más declarativo; y hace unos momentos, dijimos que nos remitiremos a datos que pertenecen al modelo relacional, junto con el paradigma imperativo como medio para su manipulación. Entonces, ¿estamos a favor o en contra del modelo imperativo como forma de programación?

Intentaremos aclarar esta aparente contradicción. Comencemos por enunciar lo que entendemos por los términos empleados: paradigma, paradigma imperativo y paradigma declarativo. Por paradigma entendemos una forma particular de pensar en la solución de los problemas computacionales, es decir, el estilo de nuestros algoritmos. Cada paradigma presupone una caja de herramientas asociada (variables,funciones,tipos,etc). Los lenguajes de programación, son el espacio donde se distingue más claramente el paradigma empleado; sin embargo, también se manifiesta en otros componentes que interactúan con los mismos (como sistemas operativos, manejadores de bases de datos, servidores de aplicaciones, etc). El imperativo es uno de los paradigmas más antiguos que se conocen en computación, y supone una estrategia donde se distinguen dos entes principales: datos y operaciones que los transforman, cambiándolos de estado en estado, hasta obtener la solución a un problema particular. Si pensamos en esta definición, los *SI* actuales, que usan bases de datos y lenguajes como C o Cobol, podrían ser clasificados bajo este rubro.

A pesar de su fama y gran extensión, el paradigma imperativo tiene una facilidad de la que suele abusarse: las variables. La posibilidad de asignarlas y de cambiar su estado en cualquier punto de un programa suele ser un arma de dos filos, y los programadores podrían adquirir el vicio de usar variables innecesariamente o bien, de reusarlas peligrosamente (cualquiera que haya intentado rastrear la ejecución irregular de un programa convencional, se habrá percatado que la misión se transforma en una búsqueda de los valores que adquieren las variables a lo largo de

la ejecución del programa). Aparte de las variables, los lenguajes imperativos (por ser los más antiguos), suelen integrarse de manera no muy armoniosa con los elementos modernos de la computación; lo que suele traducirse en la necesidad de lidiar con aspectos de bajo nivel, que no están presentes en lenguajes más modernos y especializados. A pesar de algunos detractores de esta idea, el paradigma Orientado a Objetos puede pensarse como una evolución del imperativo (especialmente, en cuanto a los datos).

En contraparte con el paradigma imperativo, tenemos al declarativo que opta por evitar los detalles de bajo nivel, y en especial, del uso de variables. Si con expresiones se puede resolver el problema, ¿para qué usar estados intermedios? La preferencia por expresiones, en lugar de asignaciones intermedias, suele traducirse en algoritmos más elegantes y menos propensos a errores (aunque podrían parecer un tanto extraños a los programadores de "la vieja escuela"). Entonces, en el paradigma declarativo, no nos salvamos de indicar cómo deseamos transformar los datos para obtener cierto resultado; sin embargo, lo hacemos olvidándonos de ciertos detalles tediosos, de manera que nos podemos concentrar en lo fundamental del algoritmo. Podemos resumir diciendo que la declaratividad consiste en una mayor especialización en ciertos problemas, misma que permite delegar ciertos detalles del cómo se harán las cosas al entorno de ejecución. El paradigma funcional suele clasificarse dentro de los declarativos.

Aclarado el asunto de los paradigmas, al menos en cuanto a sus definiciones (someras, pero suficientes, esperamos) podemos percibir que la balanza se inclina hacia la declaratividad (más aún, si recordamos que la pereza suele ser la principal motivación para usar determinadas herramientas computacionales). Continuaremos diciendo que la tendencia de esta propuesta consiste en aprovechar la declaratividad presente en los estándares propuestos, delegando la menor parte posible al uso del paradigma imperativo. Y cuando no quede de otra, la imperatividad se tratará de usar en cierta modalidad, que disminuye sus peligros: Asignación Estática Única (*Static Single- Assigment*, por sus siglas en inglés *SSA*). Dicha forma, consiste en usar variables de tal manera que, sólo una vez son asignadas dentro del programa - estamos hablando del número de asignaciones en el flujo real y no de la cantidad sintáctica de instrucciones de asignación, por ejemplo, en una condicional podrían aparecer tantas asignaciones como bifurcaciones, pero sabemos que sólo se ejecutará una; sin embargo, la excepción sería una iteración donde podría ejecutarse varias veces su única asignación sintáctica, pero ello será sólo para lograr un efecto acumulativo. Los programas con esta característica, de hecho, pueden ser traducidos a una forma puramente funcional, debido a que las variables sólo se usan como depósitos de resultados intermedios. Así mismo, tener un único punto de salida en los programas, suele ser buena práctica (aquí no entran las excepciones, por considerar que las mismas tienen un flujo de ejecución independiente u ortogonal del principal).

2.2. El estándar SQL

Pero sigamos con nuestra capa 1. Los Sistemas Manejadores de Bases de Datos (SMBD) implementan el estándar *SQL*, o por lo menos intentan hacerlo, ya que también deben guardar compatibilidad con las versiones anteriores de sus productos; y es dicho lenguaje el que usamos los programadores para comunicarles nuestros deseos, referentes a la manipulación de la información que administran por nosotros. El estándar *SQL* consta de varias versiones, donde la última de ellas apenas fue publicada y consolidada internacionalmente a finales del 2003 y principios del 2004. Sin embargo, este trabajo se basó en la penúltima versión (*SQL3* (1999)) también conocida como *SQL99*. El motivo de esta decisión fue circunstancial, ya que la concepción de las ideas de *SIUX* estuvo basada en experiencias con manejadores que a lo más, soportaban *SQL99*. Aunque la versión 2003 de *SQL*, también incorpora elementos presentes hoy en día en los *DBMS*, todavía pasará algún tiempo antes de que éstos implementen el resto de las innovaciones recién publicadas.

Por otro lado, una breve revisión al estándar *SQL:2003* nos revela que no hay choque con la propuesta aquí presentada. Al contrario, algunas características podrían usarse más cómodamente al ser soportadas directamente por la implementación de *SQL* (por ejemplo: las columnas derivadas). De hecho, en general las actualizaciones de los estándares tratan de ser inclusivas con las versiones anteriores, para minimizar el impacto sobre los productos existentes. Hecha esta aclaración sobre las versiones de *SQL*, continuemos con nuestra exposición de la versión empleada para este trabajo. *SQL99* está dividido en las siguientes secciones:

Parte 1: Marco de referencia (SQL / Framework). Define la nomenclatura general del estándar, y el ambiente de ejecución *SQL*.

Parte 2: Fundamentos (SQL / Foundation). Esto es, lo que comúnmente llamamos *SQL*, por ser la parte con la que más familiarizados estamos. Aquí están definidas la sintaxis y semántica del lenguaje *SQL*, tanto la parte *DDL*, y la parte para manipular los datos abreviada como DML (*Data Manipulation Language*). Están incluidos los tipos datos, las operaciones permitidas sobre ellos, las expresiones, las funciones que deben estar disponibles, etc.

Parte 3: Interfaz en el nivel de llamada (Call-Level Interface o SQL / CLI). Define un mecanismo de comunicación entre los clientes y servidores *SQL*, a través de una biblioteca de funciones. ODBC (*Open DataBase Connectivity*) es la implementación más famosa de esta parte.

Parte 4: Módulos Persistentes Almacenados (Persistent Stored Modules o SQL / PSM). Extiende el lenguaje *SQL* de la parte 2, agregando lo necesario para hacerlo computacionalmen-

te completo, en el sentido imperativo (variables, asignación, estructuras de control, manejadores de errores, etc.). Comúnmente conocidos simplemente como *procedimientos almacenados.*

Parte 5: Ligados con el lenguaje anfitrión (Host Language Bindings o SQL / Bindings). Define otro mecanismo alterno al mostrado en la parte 2 de comunicación entre clientes y servidores *SQL*. Una de las diferencias consiste en que en lugar de rutinas, se extienden algunos lenguajes estándar de programación, para incorporar directamente instrucciones *SQL*, tanto en su modalidad estática como dinámica.

A pesar de que nos enfocaremos a la versión *SQL99* del estándar, consideramos importante mencionar que la última versión (*SQL:2003*) fusiona la parte 5 que acabamos de listar dentro de la parte 2 (fundamentos). Otra mención relevante es que fueron agregadas nuevas secciones, dentro de las cuales está presente la interacción *SQL-XML* (parte 14).

Ya mencionamos que los manejadores actuales modernos (Oracle,DB2,Sysbase,Postgre*SQL*,etc.), definen no sólo secciones de la lista anterior, sino nuevos apartados que están planeando ser integrados: los tópicos en boga son las facilidades orientadas a objetos e integración con estándares *XML* (ver estándar *SQLX*). Pero nuevamente reiteramos que ignoraremos dichas facilidades y nos concentraremos en el paradigma relacional+imperativo (y declarativo, cuando se pueda). De hecho, de estas cinco partes, sólo la 2 y la 3 se enmarcan en el alcance del presente trabajo, y, para el caso concreto de esta sección, nos avocaremos a la segunda.

La práctica común, en cuanto a la capa 1 de nuestra arquitectura extendida, es emplear *SQL* únicamente para definir los datos (las tablas, principalmente) y poner una que otra restricción; no nulidad, llaves primarias y, si tenemos suerte, llaves foráneas. Entonces, muchas de las facilidades ofrecidas por el estándar *SQL*, a través del manejador en cuestión, se desperdician.

Muchas validaciones básicas, podrían definirse a este nivel, usando la parte *DDL* de *SQL*; aparte de las llaves foráneas, las restricciones (*constraints*), que permiten la verificación de condiciones arbitrarias sobre los campos de las tablas, son una característica comúnmente ignorada del estándar, y por lo tanto, estas validaciones son delegadas a la siguiente capa, para desgracia del programador (porque es muy tedioso programarlas una y otra vez).

Pero el lector podría preguntarse, ¿y qué tiene de malo verificarlas en la segunda capa y no en la primera, si a final de cuentas, alguien las tiene que verificar? La diferencia podría radicar en dos cosas: primera, si hacemos la verificación en la capa 1, aprovechando lo que *SQL* nos ofrece, estaríamos optando por un paradigma más declarativo (sólo decimos que deseamos validar, pero no nos preocupamos de cuándo, ni de los detalles de bajo nivel); mientras que en la capa 2

tendríamos que seguir el enfoque imperativo tradicional, lo cual definitivamente es más lento y tedioso.

Las validaciones suelen estar inmersas en las reglas del negocio, implícita o explícitamente. Para simplificar nuestra labor, podríamos optar por una clasificación muy sencilla de estas validaciones, asumiendo que ya tenemos nuestros datos estructurados en tablas: por un lado tenemos a las validaciones que se restringen a los campos de un registro determinado de una cierta tabla, y por el otro tenemos aquellas que involucran más de un registro, en una o varias tablas. A las primeras les llamaremos estáticas y a las segundas dinámicas. La razón de esta nomenclatura radica en la dependencia del *factor temporal* de cada tipo de validación. Mientras que las primeras validaciones pueden ser vistas como condiciones que deben cumplirse todo el tiempo, son una propiedad inherente a la estructura de datos que representa nuestra información (tablas de la base de datos). Por otro lado, el segundo tipo de validaciones, no necesariamente son tales que deba cumplirse todo el tiempo, sino generalmente se manejan como precondiciones para ejecutar alguna operación.

Hicimos esta breve distinción entre las validaciones que tendremos que implementar en el sistema, porque desgraciadamente no es posible, de momento y con los medios ofrecidos actualmente por el estándar *SQL*, expresarlas todas de manera declarativa. Las merecedoras de tal distinción, sólo serán las estáticas, con la importante excepción de las restricciones de unicidad dentro de una tabla (llaves primarias) y de integridad referencial (llaves foráneas), dado que éstas ya cuentan con implementaciones directas en *SQL*, y por tanto en los manejadores. Independientemente de que ello entre en coherencia con la idea de atributos instrínsecos a nuestra estructura de datos (tablas), también por cuestiones de eficiencia conviene impedir que las validaciones que llamamos dinámicas, se implementen imperativamente.

Condensando lo dicho en los párrafos anteriores, las restricciones o validaciones que nos permitiremos implementar en la capa 1, aprovechando la naturaleza declarativa de *SQL*, son las siguientes:

No nulidad. *SQL* provee una cláusula explícita para ello, que se puede asociar a cada columna de las tablas. En estricta teoría, también las deberíamos meter dentro de las validaciones estáticas, pero como son un caso especial para *SQL* (clásulas aparte), también nosotros las distinguimos.

Unicidad dentro de una tabla. Se puede implementar con las cláusulas de tipo llave primario (*primary key*), o simplemente, con las de unicidad (*unique*) que ofrece *SQL*.

Integridad referencial. Es posible gracias a las llaves foráneas que ofrece *SQL* (*foreign key*).

Validaciones estáticas. Son expresiones booleanas arbitrarias, que pueden involucrar sólo los registros de un renglón dado de las tablas; son de las que hablamos párrafos más arriba. Se pueden implementar con la cláusula *check* de *SQL*.

2.3. Migrando las validaciones

Pongamos un ejemplo para tratar de aclarar el panorama descrito. Comenzaremos con el extremo opuesto a nuestra propuesta, es decir, aquel que aprovecha al mínimo las facilidades ofrecidas por las implementaciones de *SQL*; con ello no queremos decir que este ejemplo hipotético no considere las validaciones, sólo supondremos que las delega en la siguiente capa (2). Nuestro objetivo será ir migrando esa implementación de la capa 2 a la capa 1, usando *SQL* estándar. Como ejemplo, pondremos un problema muy sencillo: mantenimiento de una tabla.

Supongamos que deseamos crear una aplicación que dé mantenimiento a una tabla, misma que representa una bitácora de actividades. Evidentemente, por el hecho de estar partiendo del requerimiento de *una tabla*, damos a entender que ya pasamos por las etapas tempranas del desarrollo de esta miniaplicación (de cualquier forma, para estos problemas tan pequeños, efectivamente solemos brincarnos las fases de análisis y diseño y saltamos directamente a definir la tabla, dada la sencillez del problema). La información que deseamos ya se encuentra plasmada en la tabla, y traducida a *SQL*:

```
create table mtto_bitacora
 (
  numero        numeric (12),
  tipo          numeric (12),
  fecha         date,
  duracion_hrs  numeric (3,2),
  descp         varchar (2048)
);

create table mtto_tipo_actividad
 (
  numero  numeric (12),
  descp   varchar (80)
);
```

La segunda tabla corresponde al catálogo de tipos de actividad. Ahora, supongamos que necesitamos implementar las siguientes validaciones sobre nuestros datos:

Ninguno de los campos puede ser nulo, todos deben contener información.

Las actividades se distinguirán de manera única por su número.

Los tipos de actividad deberán existir en el catálogo *tipo_actividad.*

Los números de actividad y sus duraciones deberán ser mayores a cero.

La fecha de la actividad, deberá pertenecer a un año posterior al 2000, pero no mayor que 2006-12-08.

Sólo interesan actividades realizadas de Lunes a Viernes. No aceptamos actividades en la segunda semana del mes, ni dentro de los últimos cinco días del mismo.

El horario para ingresar actividades, será de 9am a 2:30pm.

Ahora supongamos que implementamos nuestro sistema ignorando todo lo que ofrece *SQL*, y que nuestra capa 1 sólo corresponde con la definición de las tablas. El resto de las restricciones las implementamos en la capa 2. Supongamos que estamos usando lenguaje *C* con *SQL* incrustado para tales fines; las validaciones podrían estar implementadas en una función que comienza como sigue:

```
#define TRUE  1
#define FALSE 0

int valida_restr (int     in_numero,
                  int     in_tipo,
                  char  * in_fecha,
                  float   in_duracion_hrs,
                  char  * in_descp)
{
  int out_son_validas = TRUE;

...

  return out_son_validas;
```

El objetivo de esta función, es validar todas las restricciones que mencionamos y regresar un boleano que nos indique si dicha validación fue exitosa o no; inicialmente dicho boleano vale "verdadero", y cambiará a "falso", si la validación de alguna restricción falla. Aun no comenzamos a ver las restricciones, y ya se vislumbra el primer inconveniente: tenemos que manejar variables especiales, para emplear *SQL*; es decir, que las variables nativas del lenguaje no tienen tal virtud, y por lo tanto no las podemos emplear directamente en consultas y demás instrucciones; tenemos que copiarlas primero a estas variables especiales. Entonces, modificamos nuestra función para incluirlas dentro de una sección especial que marca el estándar de *SQL* incrustado (parte de *SQL99*):

```
#define TRUE  1
#define FALSE 0

int valida_restr (int     in_numero,
                  int     in_tipo,
                  char  * in_fecha,
                  float   in_duracion_hrs,
                  char  * in_descp)
{
  exec sql begin declarare section;
    int   sql_numero;
    int   sql_tipo;
    char  sql_fecha[32];
    float sql_duracion_hrs;
    char  sql_descp[2049];
  exec sql end declare section;
  int out_son_validas = TRUE;


  sql_numero = in_numero;
  sql_tipo   = in_tipo;
  strcpy (sql_fecha,in_fecha);
  sql_duracion_hrs = in_duracion_hrs;
  strcpy (sql_descp,in_descp);
...

  return out_son_validas;
```

Y ahora procederemos con la primera validación ..., pero esperen, aquí tenemos nuestra segunda dificultad: el hecho de manejar el lenguaje C para representar los datos, nos provoca perder la posibilidad de preguntar por los valores nulos de los mismos. Es cierto que en *C* existen apuntadores y estos sí pueden tener valores nulos, inclusive, tal vez podrían ser usados para simular los nulos de *SQL*; pero evidentemente se trata de un artificio que no resulta del todo natural. De hecho, tal cual está nuestra función, que venía luciendo de lo más natural, sólo podríamos verificar la primera condición para las variables de tipo cadena (fecha y descp). Podríamos decir que ciertos valores (-1 por ejemplo, para los numéricos), representan los valores nulos, pero ello oscurecería

aún más la lógica de la función (en nuestra visión). Otra opción, que no nos aleja tanto de *SQL* y que permitiría a todas nuestras variables el tener nulos, sería manejarlas todas como variables apuntador (del tipo en cuestión, claro está). Mostremos pues, la modificación mencionada a nuestra función:

```
#define TRUE  1
#define FALSE 0

int valida_restr (int   * in_numero,
                  int   * in_tipo,
                  char  * in_fecha,
                  float * in_duracion_hrs,
                  char  * in_descp)
{
  exec sql begin declarare section;
    int   sql_numero;
    int   sql_tipo;
    char  sql_fecha[32];
    float sql_duracion_hrs;
    char  sql_descp[2049];
  exec sql end declare section;

  int out_son_validas = TRUE;

...

  return out_son_validas;
```

Aquí tenemos que agregar otro supuesto, el cual ya no depende directamente del lenguaje que utilicemos para estas validaciones de la capa 1. *SQL* puede representar valores nulos para todos sus tipos, pero seguramente los datos vendrán de alguna interfaz gráfica de usuario, y éstas tampoco tienen (por lo menos no de manera estándar) una manera de representar este tipo de valores. A final de cuentas tenemos que elaborar alguna convención para representar cuando el usuario nos proporcionó un valor nulo (aunque ello suene paradójico). La convención que emplearemos aquí es que si el usuario no interactuó con el control asociado al campo, el *frontend* le pasa al *backend* un valor que representa el nulo, y, para nuestro ejemplo en lenguaje C, sería en valor "NULL" que pueden contener los apuntadores.

Ahora sí, pasemos a la posible implementación de la primera restricción. A propósito delegamos hasta este momento la copia de las variables de entrada en las variables para uso con *SQL*. La razón tiene que ver con detalles de bajo nivel del lenguaje (si intentamos redireccionar un apuntador nulo, obtendremos un error grave): nuevamente las características del lenguaje alejan nuestra lógica del problema original. Para evitar repetir la función una y otra vez, por cada res-

trícción sólo iremos mostrando el pedazo del código correspondiente a la restricción que se discute:

```
if  (
     in_numero       == NULL ||
     in_tipo         == NULL ||
     in_fecha        == NULL ||
     in_duracion_hrs == NULL ||
     in_descp        == NULL
   )
   out_son_validas = FALSE;
else
{
  sql_numero       = *in_numero;
  sql_tipo         = *in_tipo;
  strcpy (sql_fecha,in_fecha);
  sql_duracion_hrs = *in_duracion_hrs;
  strcpy (sql_descp,in_descp);
}
```

Todo este ritual podría haber sido evitado (menos la convención de la interfaz de usuario sobre valores nulos), si hubiéramos usado *SQL*. Veamos el equivalente para la primera restricción:

```
alter table mtto_bitacora modify numero        not null;
alter table mtto_bitacora modify tipo          not null;
alter table mtto_bitacora modify fecha         not null;
alter table mtto_bitacora modify duracion_hrs  not null;
alter table mtto_bitacora modify descp         not null;
```

Lamentablemente la sintaxis mostrada para alterar la propiedad de no nulidad es particular de Oracle (en este sentido los manejadores divergen). Sin embargo, la sintaxis para cuando esta propiedad se indica junto con la definición de la tabla, sí es respetada del estándar, así que optemos por esa (que de hecho, es mucho más compacta):

```
create table mtto_bitacora
 (
  numero        numeric (12)   not null,
  tipo          numeric (12)   not null,
  fecha         date           not null,
  duracion_hrs  numeric (3,2)  not null,
  descp         varchar (2048) not null
```

```
);

create table mtto_tipo_actividad
 (
  numero  numeric (12) not null,
  descp   varchar (80)
);
```

Y eso es todo; el manejador cuidará que, independientemente de quién haga la inserción o actualización en la tabla, nunca se pasen nulos. Dependiendo de las necesidades específicas de cada proyecto, se podría optar por la primera o segunda opción (declaración de la restricción junto con la creación de la tabla, o separado de ésta): la primera es más versátil, pero menos portátil. Nótese que si nos fuéramos por la opción que no emplea *SQL*, tendríamos que garantizar la no nulidad de la información para cada programa que pretendiera manipular la tabla. Sabemos que la no nulidad es un ejemplo trivial y en la práctica, esta situación extrema de no aprovechar nada de *SQL* es rara. Sin embargo, evidencia bien lo que sucede con otras partes del estándar que no son tan famosas.

2.4. Manejo de errores

Pasemos a la siguiente restricción: las actividades tendrán un número único, que las distinguirá. Las voces del *SQL* nos susurran *llave primaria*, pero fingiremos que no las oímos y presentaremos nuestra validación manual en C. Supondremos que agregamos a nuestra lista inicial de variables para uso con *SQL*, un contador auxiliar:

```
if  ( out_son_validas )
{
  exec sql
    select
      count (*)
    into
      :sql_count
    from
      mtto_bitacora
    where
      numero = sql_numero;

  if  ( sql_count > 0 )
    out_son_validas = FALSE;
}
```

En la versión alterna que aprovecha *SQL*, bastaría agregar la siguiente instrucción:

```
alter table mtto_bitacora add constraint mtto_bitacora_cpk
  primary key (numero);
```

Al igual que con las restricciones de no nulidad, las llaves primarias también cuentan con una notación que se puede incluir en la misma definición de la tabla; sin embargo, aquí la elección es menos arbitraria. El usar esta notación, aparte de permitirnos mayor flexibilidad (podemos modificar características aisladas de la tabla, sin desactualizar la instrucción que la crea); también tenemos la posibilidad de bautizar la restricción y ello abre las puertas a la inserción de algún esquema para el manejo de errores y sus mensajes hacia el usuario. Y es que no sólo nos interesa enterarnos de que un número de actividad se repitió, sino que necesitamos convertir dicho evento en un aviso al usuario de tal situación. Normalmente, la capa 2 se encargaría de esta tediosa labor. En la infraestructura propuesta, el manejo de errores, incluyendo sus mensajes, se incrustó al nivel 1, es decir, junto con la defínción de los datos (y es que, el lugar natural para asociar un error, ¿no debería ser acaso el mismo donde definimos la restricción que lo valida?).

Entonces, el programador de *SIUX* (acrónimo de *Sistemas de Información Usando XML y SQL*) podría anexar a la definición de su restricción la siguiente invocación a una función de la infraestructura:

```
call
  siux1_error.add_errcat
   (
    'mtto_bitacora_cpk',
    'El número de actividad de bitácora, ya está en uso.'
  );
```

Desde la butaca del programador, el escenario luce bastante cómodo: una invocación a esta rutina y ya está. Si acaso, vale la pena notar la asociación con la definición de la restricción, a través de su nombre, que precisamente es el primer argumento de la función. Aunque esta rutina en particular no exige que haya cierto estándar en los nombres de las restricciones, es una buena práctica definir alguno, y no sólo para este caso, sino para todos los objetos que vayamos a crear en la base de datos. Las aplicaciones reales requieren a menudo de una gran cantidad y tipos de entes, y es muy valioso contar con algún mecanismo que nos permita identificar quiénes son y para

qué sirven, con sólo ver su nombre. Además otra sección de nuestra propuesta de hecho requiere cierta nomenclatura para funcionar bien, así que adelantemos esto: todos los objetos asociados a una tabla deberán estar prefijados con el nombre de la misma (el sufijo "cpk" es para significar *Constraint of type Primary Key*).

Pero una vez observado el amable escenario, es nuestro deber bajar y asomarnos detrás de bambalinas para saber cómo se están haciendo las cosas; así que platiquemos más de la implementación sugerida para el manejo de errores. Antes que nada, diremos que todo descansa en la existencia de las excepciones, mecanismo relativamente moderno para permitir un manejo menos tedioso de los errores. Para entenderlas imaginemos un mundo sin ellas, para lo cual no tenemos que viajar mucho, ya que unos cuantos párrafos arriba bastarán para encontrarnos con un viejo conocido: lenguajes como C. En ellos, el manejo de errores se tiene que incorporar a los algoritmos, y en cuanto se presenta alguna situación desagradable (un dato inválido, por ejemplo) hay que evitar seguir adelante, interrumpir el flujo y salirnos de manera brusca del proceso. En un caso ideal, esta salida se hace dignamente por la puerta delantera, a través de la lógica estructurada; en muchos casos reales, se hace por la puerta trasera con una salida alterna (los programas con más de una puerta de salida no son del todo bien vistos por los académicos, por justas razones: dificultan su depuración y mantenimiento).

Entonces, remitiéndonos a estos lenguajes, nos podemos encontrar en la frustrante situación de tener gran parte de nuestra lógica infestada con validaciones y manejo de errores. Lo tenemos que hacer, lo sabemos, pero es interesante ver lo claros que pueden resultar los programas si les suprimimos todo el manejo de errores: nos quedamos con la esencia del algoritmo, donde las bifurcaciones no son por errores, sino por los dictados de las reglas del negocio. Y de esta observación, vienen las excepciones a nuestra salvación: ¿por qué tener un solo flujo para los programas, si nos estamos dando cuenta que las validaciones por sí mismas generan toda una red alterna de flujo, independiente en gran medida de nuestro flujo *ideal*? Pues entonces organicemos las validaciones en un flujo adicional alternativo.

Las excepciones son un mecanismo que permite al programador olvidarse de la verificación manual de los errores en todos los puntos del programa; y sólo nos preocupamos de los mismos en ciertos niveles superiores de los algoritmos. Estos niveles se especifican a través de bloques, donde deseamos aislar todos los posibles errores que ocurran y en estos bloques proporcionar lógica para manejarlos. Dentro del bloque todo mundo podría ignorar validaciones, puesto que el entorno de ejecución se encargará de suspender el flujo normal ante la presencia maligna de un error; y mandarlo por esa autopista alterna hasta nuestro código para manejarlo. En algunos lenguajes, el mecanismo de excepciones inclusive permite la posibilidad de repropagar la excepciones, una vez "atrapadas".

Podemos añadir que las excepciones también pueden ser clasificadas dentro de lo que conocemos como "programación orientada a eventos". El sistema de ejecución subyacente tiene predefinidas ciertas acciones ante la presencia de algunos eventos (como los errores). Las excepciones entran en este esquema, donde uno puede aumentar ese conjunto predefinido de eventos, que activan el flujo alterno de ejecución, por medio del cual podemos atrapar los errores. Aunque *SQL* podría no exigirlo, las implementaciones del *SQL* que tenemos en nuestros días suelen recurrir al mecanismo de excepciones para reportar errores; mismo que se hace más evidente cuando nos movemos de *SQL* a *PSM* (extensión imperativa del primero). Adicionalmente, la naturaleza misma del estándar permite asociar la generación de excepciones a los eventos deseados (errores), de una manera declarativa (*DDL*). Oracle, la implementación seleccionada para esta propuesta, no es la excepción en esta tendencia y ello se aprovechó para implementar el manejo de errores (tratando de disminuir el tedio al usarlo).

Internamente, SIUX define los siguiententes catálogos para la administración de errores:

```
create table siux1_errcat1
(
  appname        varchar (64)   not null,
  errcode        numeric (12)   not null,
  excpname       varchar (64)   not null,
  primary key  (appname,errcode),
  unique  (appname,excpname)
);

create table siux1_errcat2
(
  appname        varchar (64)   not null,
  errcode        numeric (12)   not null,
  lang           varchar (8)    not null,
  descp_errcode  varchar (2048) not null,
  primary key  (appname,errcode,lang),
  foreign key  (appname,errcode)
    references siux1_errcat1 (appname,errcode)
);
```

En general, toda la información de los catálogos de SIUX se encuentra clasificada por aplicaciones, así que todas las tablas tendrán ese campo como parte de la llave primaria (con lo cual permitimos que varias aplicaciones guarden su metainformación en una misma base de datos). El primer catálogo relaciona los nombres de la excepciones (que son más fáciles de recordar), con unos códigos o números de error que se generan automáticamente. Esta representación dual es necesaria porque, mientras que dentro de los programas el empleo de los nombres puede ser cómodo, a la hora de salir de entorno de ejecución *SQL* y transmitir los datos hacia otro ambiente (como el lenguaje de programación C o Java), sólo podemos regresar dos datos acerca de algún

error: un código numérico y un mensaje asociado. Tradicionalmente, este par lo usa únicamente el manejador de la base de datos, y la capa 2 se encarga de traducirlos a mensajes más amables para el usuario. En esta propuesta se proporciona la posibilidad de extender esos códigos y mensajes del SBMD con el catálogo de errores propio del Sistema de Información en construcción. Así, tenemos aún más centralizada la administración de los errores: definimos sus mensajes asociados en el mismo lugar donde realmente los detectamos.

En la segunda tabla podemos apreciar información variable del error, como lo es su mensaje asociado, mismo que suele estar en función de información local (de momento, sólo se nos ocurrió el lenguaje usado para los datos de la aplicación). Entonces, lo que pedimos para usar el manejo de errores son tres cosas: primero, definir la restricción que le pedirá al manejador validar el error por nosotros y notificarnos; segundo, asociar el nombre de dicha restricción con una excepción y un mensaje de error (a través de la rutina siux1_error.add_errcat); por último, se pide que cuando sea deseado "atrapar" cualquier error (o excepción), definido de esta forma, se haga uso de la instrucción *PSM* para tal fin (*exception* en Oracle) y se invoque al procedimiento siux1_error.handle_error (que se encargará de propagar la excepción adecuada, junto con el mensaje asociado al nivel de invocación superior inmediato). Este último paso lo veremos hasta llegar a la capa 2.

Las rutinas de SIUX, se encargan de convertir los errores genéricos de manejador, a los errores definidos por el usuario a través de los catálogos. Pero un momento, ¿cómo se asocia el mensaje adecuado para los errores de nulidad, cuando no definimos mensaje alguno ahí? La respuesta radica en que dichos errores usan un mensaje genérico (que debe ser ingresado una sola vez para toda la aplicación), sin embargo, este caso particular también usa otro tipo de metainformación: los datos de las columnas de las tablas. Habría que mencionar que para que funcione correctamente el manejo de los errores de no nulidad, en nuestro ejemplo hipotético de mantenimiento tendríamos que hacer las siguientes invocaciones después de la definición de la tabla:

```
call
  siux1_info.add_colinfo
   (
    'mtto_bitacora',
    'numero',
    'Número',
    'Número de actividad'
  );

call
  siux1_info.add_colinfo
   (
    'mtto_bitacora',
    'tipo',
    'Tipo',
    'Tipo de actividad'
```

```
  );

call
  siux1_info.add_colinfo
   (
    'mtto_bitacora',
    'fecha',
    'Fecha',
    'Fecha de realización de actividad'
  );

call
  siux1_info.add_colinfo
   (
    'mtto_bitacora',
    'duracion_hrs',
    'Duración (hrs)',
    'Duración en horas de actividad'
  );

call
  siux1_info.add_colinfo
   (
    'mtto_bitacora',
    'descp',
    'Descripción',
    'Descripción de actividad'
  );
```

Las invocaciones anteriores nutren el siguiente catálogo de metainformación sobre nuestras tablas:

```
create table siux1_colinfo
(
  appname  varchar (64)    not null,
  lang     varchar (2)     not null,
  tabname  varchar (32)    not null,
  colname  varchar (32)    not null,
  label    varchar (128)   not null,
  descp    varchar (2048)  not null,
  primary key  (appname,lang,tabname,colname)
);
```

Las primeras dos columnas se toman de valores predefinidos, mismos que pueden sobreescribirse con una invocación previa a toda definición, donde inicializamos la infraestructura e indicamos el nombre de aplicación y el idioma. El resto de las columnas de este catálogo corresponde respectivamente a los argumentos de la rutina empleada y son: nombre de la tabla, nombre de la columna, etiqueta y descripción de la misma. Esta información es más que un mero lujo extravagante, es la

que permite personalizar más los mensajes de error genéricos, como los de nulidad (el nombre de la tabla-columna se reemplaza por la etiqueta); pero también es la pauta para permitir a capas superiores informar sobre los servicios que otorga el *backend*, y a su vez, esto puede habilitar la posibilidad de generar automáticamente las interfaces de usuario (en algún lado se tienen que tomar esas leyendas, así que adelantamos y centralizamos su ubicación en esta primera capa).

2.5. Más sobre implementación de validaciones

Retomando la comparación entre los dos estilos de programar las validaciones básicas, continuamos con la siguiente restricción: los tipos de actividad deberán existir en el catálogo *tipo_actividad*. Toca el primer turno al más antiguo:

```
if  ( out_son_validas )
{
  exec sql
    select
      count (*)
    into
      sql_count
    from
      mtto_tipo_actividad
    where
      numero = sql_tipo;

  if  ( sql_count = 0 )
  {
    out_son_validas = FALSE;
    ...
  }
}
```

Con *SQL* claramente tenemos la opción de las llaves foráneas. Seguimos las convenciones que acabamos de presentar hace algunas líneas:

```
call
  siux1_error.add_errcat
   (
    'mtto_bitacora_cfk1',
    'El tipo de actividad ingresado, no existe en catálogo.'
  );
```

```
alter table mtto_bitacora add constraint mtto_bitacora_cfk1
  foreign key  (tipo) references mtto_tipo_actividad (numero);
```

Nótese que no sólo estamos definiendo de manera central las restricciones (independiente de los programas que acceden a las tablas), sino que de paso estamos nutriendo el catálogo de errores. Sigamos con la siguiente validación, referente al número y duración de las actividades.

```
if  ( out_son_validas )
{
  if  ( !  (sql_numero > 0 && sql_duracion_hrs > 0) )
  {
    out_son_validas = FALSE;
    ...
  }
}
...
```

En *SQL* indicamos la condición que deseamos se mantenga en todo momento (la invariante), aunque en la práctica suponemos que el SBMD activa los mecanismos de validación ante la presencia amenazadora de instrucciones que puedan modificar el contenido de las tablas (*SQL*-DML). Presentamos aquí a la cláusula *check* de *SQL*, no tan conocida como las facilidades mostradas anteriormente, pero bastante útil ya que podemos incluir expresiones que representen predicados arbitrarios de validación, sea sobre una sola columna, o sobre todas las incluidas en el registro de la tabla en cuestión:

```
call
  siux1_error.add_errcat
   (
    'mtto_bitacora_cch_a001',
    'Valor inválido para el número o la duración de actividad.'
  );

alter table mtto_bitacora add constraint mtto_bitacora_cch_a001
  check  (numero > 0 and duracion_hrs > 0);
```

Veamos el siguiente elemento de nuestra lista de reglas por implementar, mismo que nos permitirá comenzar a valorar el uso de *SQL* para validar los datos: la fecha de la actividad deberá pertenecer a un año posterior al 2000, pero no mayor que 2006-12-08. Aquí ya nos topamos de frente

con el hecho de que estamos programando la lógica en un lenguaje externo al lugar que alberga y manipula directamente los datos. Para comenzar, C no tiene ningún tipo que represente directamente fechas, pero hemos tomado la convención de usar cadenas para ello. Luego, tenemos el "pequeño inconveniente" de no contar con una función que valide la fecha, aceptando un formato arbitrario (como sucede en *SQL*); y ello ya no es una tarea trivial de programación que pueda hacerse en 5 minutos. Pero no seamos tan duros; supongamos que ignoramos la validez en general de la fecha, y nos concentramos sólo en su formato (supongamos yyyy-mm-dd) y en la restricción particular que tenemos frente a nosotros (nuevamente, imaginemos que ya declaramos las variables correspondientes):

```
if  ( out_son_validas )
{
  if  (
        sscanf(fecha,"%04d-%02d-%02d",&ano,&mes,&dia) != 3 ||
        ! (ano > 2000  &&  (ano*10000 + mes*100 + dia) <= 20061208)
      )
  {
    out_son_validas = FALSE;
    ...
  }
  ...
}
```

Si nos quedamos del lado de *SQL*, no tenemos necesidad de convertir a un tipo de datos auxiliar, puesto que ya tenemos uno para representar las fechas directamente; es más, cuando dicho dato sea introducido a la tabla, el manejador aceptará cierto formato indicado por nosotros (yyyy-mm-dd, por ejemplo) y se valida independientemente de la sintaxis, que la combinación año, mes y día sea válida. Así que podemos estar seguros de tener una fecha correcta, y nos concentraremos en la condición. También podemos asumir que el manejador ha sido informado del formato deseado para la fecha y que, al encontrar una cadena, hará la conversión de tipo automáticamente:

```
call
  siux1_error.add_errcat
   (
    'mtto_bitacora_cch_a002',
    'La fecha de la actividad no pertenece al rango  (2001-01-01,2006-12-18]'
  );

alter table mtto_bitacora add constraint mtto_bitacora_cch_a002
  check  (extract (year from fecha) > 2000 and fecha <= '2006-12-08');
```

Para reforzar la idea que deseamos ilustrar, pasemos inmediatamente a la siguiente precondición: sólo interesan actividades realizadas de lunes a viernes. No aceptamos actividades en la segunda semana del mes, ni dentro de los últimos cinco días del mismo. La situación de querer usar un lenguaje como C comienza a cobrar su precio aquí; y es que ya no resulta del todo práctico. No sólo se trata de la ausencia del tipo de datos para representar fechas, sino de la falta de operadores o funciones para manipularlos. Tal vez podríamos usar una función como *strptime*, para obtener una representación interna de la fecha-hora, pero con ello, a lo más, ganaríamos saber en qué día de la semana estamos. Preguntas como el último día del mes, o el número de semana, quedan fuera del alcance de la biblioteca estándar de C (y probablemente, también de otros lenguajes similares). Necesitaríamos alguna biblioteca ya hecha, que nos permitiera obtener esta información; lo cual, por supuesto, no sería un mecanismo estándar.

Después de ni siquiera haber intentado implementar esta validación directamente en C (no es el tipo de cosas que un programador de *SI* (acrónimo de *Sistemas de Información*) querría hacer para un mantenimiento de tablas), pasemos a la opción que ofrece *SQL*:

```
call
  siux1_error.add_errcat
   (
    'mtto_bitacora_cch_a003',
    'Fecha inválida: sólo tareas de L-V, descontando segunda semana ' ||
     ' del mes y los últimos 5 días del mismo.'
  );

alter table mtto_bitacora add constraint mtto_bitacora_cch_a003
  check  (
           (to_char (fecha,'D') between 2 and 6) and -- 1 = domingo
           to_char (fecha,'W') <> 2              and
           (last_day (fecha) - fecha) > 5
         );
```

Nótese que estamos aprovechando el hecho de que los manejadores modernos hacen conversión automática de tipos (de *char* a entero, en este ejemplo); también usamos la posibilidad de hacer aritmética de días directamente sobre las fechas. Es justo decir que estamos haciendo trampa, dado que usamos la función *last_day*, que es propia de Oracle, y no precisamente parte del estándar. Sin embargo, se puede implementar con otras primitivas (por ejemplo, se puede calcular el primer día del siguiente mes, y restarle un día).

No queríamos ser aguafiestas y esperamos hasta este momento para mencionar una limitación: se refiere a la imposibilidad de usar expresiones arbitrarias en la cláusula *check*. Estas imposiciones pudieran variar de implementación en implementación de *SQL*; por ejemplo, en Oracle no

se pueden usar funciones que sean *no determinísticas*, en particular, funciones que proporcionen información del contexto, por ejemplo, cosas tan básicas como la fecha u hora actuales. Esa situación no ocurre en Postgre*SQL*, aunque en ambos coincide el criterio de impedir la presencia de subconsultas dentro de la expresión. Veamos como resuelve SIUX esta limitante, con la siguiente restricción: el horario para ingresar actividades será de 9am a 2:30pm. Lo natural sería pensar en una implementación como la siguiente:

```
alter table mtto_bitacora add constraint mtto_bitacora_cch_a004
  check ( current_time between '09:00:00' and '14:30:00' );
```

Pero recordando las limitantes de la claúsula *check*, por lo menos en Oracle, tendríamos que recurrir al siguiente mecanismo:

```
call
  siux1_error.add_errcat
   (
    'mtto_bitacora_cch_a004',
    'El horario de nuestro sistema marca las ''%1'', lo cual ' ||
      'lamentablemente, se encuentra fuera del horario ' ||
      'permitido (9-14:30).'
  );

create or replace procedure mtto_bitacora_cch_a004
(
  in_fecha in date
)
as
begin
  if  ( not  (current_time between '09:00:00' and '14:30:00') ) then
    siux1_error.put_errpar
     (
      'mtto_bitacora_cch_a004',
      1,
      current_time
    );
    siux1_error.raise_uexcp ('mtto_bitacora_cch_a004');
  end if;
end mtto_bitacora_cch_a004;
/
```

Lo admitimos, hemos perdido la declaratividad en pos de habilitar esta restricción. Sin embargo, esta implementación imperativa continúa siendo más natural que la posible equivalente en C, u

otro lenguaje similar, porque aquí contamos con las expresiones que tienen acceso directo a los datos (expresiones acompañadas por un rico juego de funciones, en su mayoria estándar). Nótese que aquí requerimos propagar la excepción manualmente con la invocación a *siux1_error.raise_uexcp* (cosa que el SBMD hacía por nosotros, en los ejemplos anteriores). A manera de consuelo, diremos que por este mecanismo podríamos incluir lógica arbitraria que podría incluir funciones definidas por el usuario, o consultas *SQL*. También nótese que, con el mismo espíritu mostramos la posibilidad de hacer los mensajes más detallados, más personalizados; y es que muchas veces, los mensajes son pensados como moldes que son rellenados con valores que se presentan en tiempo de ejecución. Lo que aquí vemos es la opción de SIUX para esa parametrización (el símbolo de porcentaje seguido de un 1 en el mensaje y la llamada a la rutina *siux1_error.put_errpar*).

2.6. Restricciones procedurales

Postergamos hasta este momento la explicación de la nomenclatura para una restricción (que no es arbitraria, sino todo lo contrario), y la aparición de estas rutinas (que se hizo necesaria por limitaciones del SMBD) nos indica que ya es tiempo. A diferencia de las instrucciones *SQL* para agregar restricciones a nuestra tabla, en estos procedimientos no existe un nexo directo con ningún evento; entonces, ¿cómo esperamos que sean equivalentes?; a final de cuentas, fueron presentados como una opción alterna para solventar las carencias de las cláusulas *check*. La respuesta es que SIUX se encarga de esta asociación, instalando detonadores (*triggers*) en las tablas e incluyendo en los mismos las invocaciones pertinentes; y aquí, los nombres de nuestros objetos cobran la importancia antes señalada. Por uniformidad, todas las restricciones usan la misma convención en el nombre, aunque solo las versiones imperativas se fían del mismo para funcionar correctamente. Por ejemplo, un nombre como *mtto_bitacora_cch_a004*, nos indica varias cosas, rompámoslo en bloques:

mtto_bitacora: Indica la tabla asociada con esta validación imperativa (rutina o procedimiento, en otras palabras). Muy necesario, pues requerimos saber en qué tabla instalar el detonador donde haremos las invocaciones pertinentes. Nuestras validaciones se presentaron como *implementaciones de las invariantes de nuestros datos*, sin embargo, al estar sin el manto protector del SBMD, no queda otra más que ser prácticos y admitir que sólo se atentará contra la integridad de nuestros datos ante la proximidad de instrucciones *SQL*-DML (*insert,delete o update*); y precisamente con los detonadores podemos atrapar dichos *eventos*.

cch: Indica el tipo de validación: en este caso, una restricción de tipo *CHeck*. Con las restricciones declarativas, la infraestructura SIUX no se mete (el manejador hace todo el trabajo sucio); pero adicionalmente a las validaciones, también podemos incluir procedimientos que calculen

automáticamente columnas *derivadas*, es decir, cuyos valores se pueden obtener directamente a través de las demás columnas del registro al que pertenecen. Entonces, si tenemos estos dos tipos de rutinas, a ser llamados desde los detonadores, conviene distinguirlos (ya que seguramente las validaciones irán primero). La etiqueta para dichos procedimientos, en esta sección del nombre sería *dcl* (*Derived CoLumn*).

Al final de esta sección del nombre, y antes del carácter de subrayado, podríamos poner un indicador de la operación que deseamos lance la ejecución de nuestra rutina. Usaríamos una *i* (*nsert*) o una *u* (*pdate*). Al no estar presente dicho indicador, se asume deseamos la invocación ante la presencia de ambos eventos.

a: Esta letra indica el tiempo en el cual deseamos se haga la invocación de la rutina con *a* para después (*after*) y *b*(*efore*) para indicar la contraparte. El antes y después al que nos referimos tienen que ver con la visibilidad de los cambios en la tabla durante la ejecución de una instrucción *SQL-DML*. Si marcamos el momento como *después*, entonces SIUX instala un detonador de ese tipo, y podemos estar seguros que dentro de la ejecución de nuestro procedimiento, ya podemos contar conque la operación ha sido intentada por parte del manejador, y que, si llegamos hasta ese lugar, es porque todo ha salido bien (particularmente importante es el hecho de que todas las validaciones declarativas se han cumplido). Con estas justificaciones, sonaría prudente pedir que toda validación imperativa se lance después del intento; pero hay casos que lo impiden. Podríamos necesitar comparar los valores previos del registro con los nuevos (sólo aplica para actualizaciones), o bien, hacer consultas de validación a la misma tabla afectada. Ambas cosas, son imposibles de lograr en un detonador *after*, así que no tenemos más remedio que pedirlas antes del intento de la operación DML.

Haremos otro paréntesis, dependiente de la implementación seleccionada de *SQL*. La restricción de las consultas en los detonadores *after*, podría ser particular de Oracle. Éste protesta diciendo que la tabla se encuentra en un estado *de mutación*, lo que suponemos se puede comparar con un estado intermedio entre el el intento de realizar la operación (que ya pasó), y la materialización de la misma (*commit*). En ese hueco, al invocarse nuestras rutinas que hacen consultas a la misma tabla, el manejador, supongo, intenta avisarnos que la tabla aún se encuentra en estado *sucio*, y que sus datos pudieran no ser confiables todavía.

004: Éste es simplemente el número de sequencia para este procedimiento. Dado que pudieran existir otros con las mismas características (hecho que se manifestaría con que ambas rutinas tendrían el mismo prefijo, con excepción de este número). Así, podemos indicar a SIUX el orden en la ejecución que deseamos para nuestras validaciones imperativas.

Demos unos ejemplos del uso de las rutinas para calcular las columnas derivadas, mismas que mencionamos hace apenas unos momentos. Supongamos que necesitamos guardar la descripción de la tarea sin espacios al principio o final, y en mayúsculas. En lugar de forzar una validación para ello, nosotros nos aseguramos de garantizar dicho resultado. Otra petición común podría ser el generar un número automático de la actividad, cuando éste no se informa (se presentaría como NULL ante nuestro procedimiento, y si lo dejáramos pasar se activaría la restricción declarativa de no nulidad). Ambas peticiones, pueden ser atendidas como sigue, dentro de SIUX:

```
create or replace procedure mtto_bitacora_dcl_001
(
  io_descp  in out varchar
)
as
begin
  io_descp  := upper (trim (io_descp));
end mtto_bitacora_dcl_001;
/

-- Foliador de actividades
create sequence mtto_bitacora_seq_numero start with 1;

create or replace procedure mtto_bitacora_dcli_002
(
  io_numero  in out numeric
)
as
begin
  -- Sólo ponemos folio, si el usuario no lo informó explícitamente
  if  ( io_numero is null ) then
    select
      mtto_bitacora_seq_numero.nextval
    into
      io_numero
    from
      dual;
  end if;
end mtto_bitacora_dcli_002;
/
```

Reusando la explicación previa, estamos indicando a la infraestructura que la corrección de la descripción debe ser lanzada tanto en inserciones, como en actualizaciones; mientras que la generación de folios únicamente en las inserciones. Una diferencia a notar con los procedimientos de validación es que los parámetros deben entrar en modo de *lectura/escritura*, puesto que pueden ser modificados (las validaciones los requieren en *sólo lectura).*

2.7. Auditoría en tablas

Un ejemplo muy famoso de las columnas derivadas son las llamadas *columnas de auditoría*, llamadas así porque pretenden llevar un registro de quién, cuándo y desde dónde modificó o creó un registro de cierta tabla (por mencionar sólo los campos más comunes). Probablemente no todas las tablas que hagamos merezcan tales precauciones, pero ¿si contáramos con un mecanismo automático de auditoría? sería difícil resistirse, "por si acaso", diríamos.

Sin embargo, interrumpiremos esta presentación diciendo que la definición de columnas derivadas no encaja directamente con el comportamiento esperado para estos campos. Y es que sus valores no son tomados de valores previos de los mismos, o de algún otro campo del registro, sino del contexto de ejecución. Pero podemos tranquilizar nuestras conciencias, diciendo que la definición de columna derivada podría extenderse para abarcar cualquier expresión; de hecho, con la posibilidad de hacer consultas *SQL*, ya ni siquiera tenemos por qué pedir la restricción a los campos del registro.

Muy bien, continuemos. SIUX ofrece un servicio automático de auditoría, a través de una invocación como la que sigue:

```
call siux1_audit_cols ('mtto_bitacora');
```

Con esta invocación, estamos agregando las siguientes columnas a nuestra tabla:

```
SIUX1_INS_USER          NOT NULL VARCHAR2 (128)
SIUX1_INS_IP            NOT NULL VARCHAR2 (64)
SIUX1_INS_TIMEST        NOT NULL DATE
SIUX1_UPD_USER          NOT NULL VARCHAR2 (128)
SIUX1_UPD_IP            NOT NULL VARCHAR2 (64)
SIUX1_UPD_TIMEST        NOT NULL DATE
```

No hay mucho misterio en cuanto al propósito de cada columna: se trata del nombre del usuario, la *IP* desde donde se conecta y el momento en que lo hizo -el tipo *date* de Oracle, en realidad es una marca de tiempo (*timestamp*)-; esto tanto para la inserción como para la actualización del registro. Mientras que el *momento* puede obtenerse con un simple llamado a la función *current_timestamp*, el usuario y la *IP* podrían no ser tan obvios. Prácticamente todos los SMBD

permitirán acceder de una u otra forma a estos datos, pero con respecto al cliente *SQL*, es decir, al programa que se conecta al manejador para hacer la inserción o la actualización. Y dicha información suele diferir de la que tenemos en mente cuando nos llenan la cabeza con la palabrita *auditoría.*

Lo común sería que el usuario y la *IP* que nos pueda ofrecer directamente el ambiente de ejecución del SBMD, sean aquéllos que pertenecen a otro componente de nuestro sistema en conjunto (por ejemplo, el servidor de aplicaciones se conecta al servidor de base de datos con algún usuario genérico). Y lo que nos interesa más bien es la clave de usuario empleado por el *humano* para identificarse con el sistema en su conjunto (que no necesariamente coincide con uno de la BD), así como la *IP* que usó desde su casa u oficina para el acceso. Pero estos datos no los conoce el manejador de los datos, sino los intermediarios entre el usuario final y éste. Así que la solución prudente sería: cada intermediario con el SMBD, al levantar o usar una conexión con el mismo, debería ser responsable de proporcionar esta información, para que la misma sea accesible desde las capas 1 y 2 de la infraestructura. Unos procedimientos de SIUX cumplen esta función.

Por último, diremos que la invocación de la rutina *siux1_audit_cols* no sólo habilita la creación y mantenimiento de las columnas de auditoría, sino que es la puerta de entrada para que ingrese la consideración de toda la lógica imperativa (rutinas tipo *cch* y *dcl*), que hayamos implementado por parte de SIUX. Todo ello se agrupa en los detonadores correspondientes para la tabla en cuestión.

Capítulo 3

Capa 2: Implementación de las reglas del negocio

3.1. Opciones para implementar las reglas del negocio

Esta parte de la propuesta probablemente sería la que podría generar algún tipo de controversia o extrañeza ante los lectores. Y es que, habitualmente, el manejador de la base de datos es visto como un mero receptáculo de información; mientras que algún lenguaje externo al mismo, es apreciado como el ambiente ideal para implementar las reglas del negocio (conectándose al SBMD con algún mecanismo o biblioteca ofrecido por el mismo: JDBC, ODBC, etc).

Entonces, Java, C o Cobol vendrían siendo las propuestas esperadas para esta capa. Sin embargo nosotros estamos proponiendo al lenguaje *PSM* (*Permanent Stored Modules*), que forma parte del estándar *SQL99*. Dicho lenguaje podría ser visto como el *SQL* que conocemos (*DDL*+DML), más una inyección de estructuras de control (que recuerdan a Pascal) y que lo hacen computacionalmente completo. ¿Y quién implementa dicho lenguaje? El mismo SMBD, dado que es parte del estándar.

Pero no tenemos la intención de ocultar las deficiencias de *SQL* como un estándar, ya que comparando su situación a ese respecto, con los lenguajes de programación mencionados, no podemos evitar pasar un trago amargo. Mientras que con los lenguajes de programación, la variabilidad la tenemos principalmente a nivel de bibliotecas adicionales, pormenores del ambiente de ejecución y otros agregados que suelen dar los fabricantes (tanto comerciales, como proyectos de código libre); en el caso de *SQL* la situación es todavía más crítica, y no estamos hablando aún de la parte correspondiente a *PSM*; ni siquiera es necesario llegar hasta ese lugar para detectar los problemas actuales.

Revisemos las situaciones familiares para los programadores de *SI*. Para dichos casos, sabemos que parte fundamental de nuestro diseño e implementación tendrá que ver con el modelo de datos y los accesos que definamos sobre el mismo. Y también sabemos (probablemente desde la escuela) que existe el estándar *SQL*, en el cual podremos comunicarnos con un sistema que administrará la información. Podemos incluso aprender sobre las bases teóricas que fundamentan dicho lenguaje (álgebra relacional) o comprar libros sobre el estándar. En esta etapa sentiremos lo agradable que es contar con el conocimiento neutral de la plataforma y detalles de implementación, porque se nos antoja poder comenzar la definición de nuestras tablas, o incluso las consultas de acceso, sin pensar todavía en algún producto en concreto.

Y podríamos incluso ceder ante la tentación, y construir las instrucciones *SQL-DDL* y *SQL-DML* para nuestro sistema. Pero llegado el momento, querríamos plasmar nuestras ideas en alguna implementación real; y aquí comienza la desilusión. Para comenzar, veremos que a pesar de las promesas de los fabricantes, el estándar no está completamente soportado en sus productos, y que algunas de las cosas que realizamos simplemente no funcionarán. Entonces, una decisión prudente podría ser el escoger aquel fabricante que ofrezca la mayor conformidad con el estándar; pero muy seguramente, dicha decisión no está en nuestras manos (sea por el lugar que ocupamos en el organigrama, sea por políticas de la empresa para la cual trabajamos, o bien, por el factor económico). Entonces, viene la resignación con no poca melancolía: desarrollaremos pensando en un manejador en particular.

El siguiente problema viene cuando tenemos que hacer interactuar a nuestro sistema con otros, y siendo más concretos, cuando tenemos que migrar lógica de negocios de los mismos. Lo frustrante no es sólo que el estándar sigue teniendo partes sin soportar, sino que cada producto parece abarcar un subconjunto distinto. Entonces, vemos que mientras unos soportan más tipos de datos, otros tienen más funciones enteras implementadas; pero aquéllos ofrecen mayor extensibilidad. Vamos, ni siquiera a nivel sintáctico, uno puede estar a salvo. Entonces, ¿dónde quedó la ventaja de tener un estándar, si nadie lo respeta? Tratemos de ser justos en este punto. Es verdad, no podemos teclear *SQL* genérico y esperar que sea reconocido por los implementaciones más comunes (como esperaríamos con un trozo de código en C o Java, por ejemplo). Empero, las ventajas son parecidas al aprender lógica de programación genérica, en lugar de algún lenguaje de programación concreto: soluciones más genéricas y parametrizables, aparte de que la facilidad para movernos de implementación en implementación; podemos pensar en las definiciones y accesos a los datos, en *SQL* genérico, y después aterrizarlos en la sintaxis e idiosincracia específica del SMBD (auxiliados por la documentación correspondiente). Además, el haber conocido antes el estándar, en lugar que los productos nos ayudará a no depender innecesariamente de características propias de los mismos: si alguna funcionalidad no es *SQL* estándar, no es seguro usarla (ya que podría desaparecer en versiones posteriores).

Pero algunos desarrollos sencillamente no pueden vivir admitiendo el matrimonio con una implementación particular de *SQL*, y deciden usar capas intermedias a los datos que oculten la presencia de manejador. Esto es especialmente común en desarrollos que utilicen el paradigma orientado a objetos. Un patrón muy común de diseño, en tales casos, es el representar a las tablas con objetos, y a las famosas operaciones de *insert*, *update* o *delete* como métodos de los mismos. Inclusive, algunos manejadores o herramientas adicionales son capaces de generar el código de dichas clases, usando la metainformación de la base de datos. A pesar de esta facilidad, nótese que no estaríamos evitando tener un manejador concreto; simple y sencillamente, la mayor parte de la lógica la estamos sacando del mismo y llevando a código en el lenguaje de programación seleccionado. Adicionalmente, se necesitaría de código específico para cada SMBD con los cuales quisiéramos trabajar; que sirviera a manera de engrudo con el resto de la aplicación. La ventaja es que dicho código sería mínimo (en comparación con el resto del desarrollo).

Aunque el esquema anterior es una solución viable, y nos ofrece la recompensa de la portabilidad de la solución, también es cierto que pagamos un precio por ello. El que dicho costo sea muy elevado o insignificante, podría depender más del paradigma preferido del programador. Estamos hablando de que la gracia de la declaratividad de *SQL* nos ha dejado de acompañar; la hemos canjeado por instrucciones de algún lenguaje de programación, y probablemente esto se traduzca en imperatividad (la OO no se salva de esto). Y si retomamos lo mencionado acerca de los paradigmas en el capítulo anterior, recordaremos que lo imperativo a menudo viene acompañado de los detalles de bajo nivel. Tal vez este punto no quede del todo claro, si nos quedamos con la idea de que acceder a los datos se reduce a instrucciones básicas de *insert*, *update* o *delete*; pero en cambio, se puede aclarar si metemos al juego a las expresiones.

De hecho, podríamos decir que una caracterización de los paradigmas de alto nivel es precisamente su empeño en maximizar el uso de expresiones. Todo mundo ha experimentado las ventajas de este "hábito" incluso en los lenguajes tradicionales; por ejemplo, las expresiones que evalúan hacia un valor boleano y que son empleadas en las estructuras de control condicionales; algunos patrones de estas instrucciones pueden ser fácilmente remplazados por un solo bloque, pero eso sí, con una expresión más compleja. Por ejemplo, patrones tan sencillos como los siguientes (que no son raros en programadores poco expuestos al paradigma funcional):

```
if  ( condición1 )
{
  if  ( condición2 )
  {
    if  ( condición3 )
    {
      ...
    }
  }
```

```
}

if  ( condición4 )
{ bloque1; }

if  ( condición5 )
{ bloque1; }
```

Pueden ser simplificados aprovechando las expresiones que ofrezca el lenguaje, C en este caso:

```
if  ( condición1 && condición2 && condición3 )
{
  ...
}

if  ( condición4 || condición5 )
{
  bloque1;
}
```

Acabamos de canjear la estructura en el algoritmo mismo (estructuras de control), por estructura en las expresiones (que en general son más compactas). La idea que tratamos de ejemplificar con un lenguaje de programación relativamente genérico, se puede trasladar al escenario que nos concierne: *SQL* usado desde otro lenguaje externo. Esto se logra poniendo atención en las expresiones que ofrece *SQL*, y que podrían ser subutilizadas al preferir el lenguaje convencional. Ya en el capítulo anterior observamos algunas de las inconveniencias de esta situación, que comienza desde el mismo hecho de que *SQL* y nuestro lenguaje *anfitrión* manejan tipos de datos distintos; lo que implica un proceso de aproximación (tipos *SQL* como las fechas o los números de precisión arbitraria, pueden ser un dolor de cabeza en este sentido). Luego, las expresiones escalares (resuelven a un valor de tipo simple) que podríamos usar en *SQL*, y que es poco probable que las encontremos en nuestro lenguaje de forma nativa, muy probablemente tengamos que traducirlas a una sucesión de instrucciones imperativas, alternando el uso de rutinas de alguna biblioteca auxiliar.

Aunque implique tedio, las validaciones y cálculos que podríamos hacer empleando expresiones *SQL*, una vez convertidas a sus implementaciones en el lenguaje anfitrión (como C o Java), podrían ser aisladas en subprogramas y ello disimularía, o por lo menos nos haría olvidar parcialmente, el precio que nos están cobrando por alojamiento (de *SQL*). De hecho, retomemos nuestra discusión anterior, donde presentábamos el caso de una aplicación que prefirió la portabilidad y decidió depender al mínimo del SMBD (lo cual se traduce en minimizar el uso de *SQL*). También recordemos que mencionamos al paradigma Orientado a Objetos (OO) como el lugar común donde

hemos visto esta situación. Entonces, podríamos pensar en que tenemos nuestras tablas representadas por clases, y dentro de las mismas métodos que implementan imperativamente las validaciones y que son invocadas, por ejemplo, en los constructores de dicha clase.También podríamos agregar a nuestras clases hipotéticas, métodos que representen las operaciones básicas de manipulación *SQL* (*insert,update* y *delete*); incluso podríamos pensar en una consulta básica, representada por otro método, y que podría regresar, en lugar de un cursor, alguna estructura de datos lineal y homogénea del lenguaje para representar el resultado de las búsquedas (una clase, de hecho).

Hasta este punto, el panorama no pinta tan mal; no parece que hayamos perdido nada importante al abandonar *SQL*. Es cierto que lo seguimos utilizando, pero únicamente para lo indispensable y lo refundimos en lo más recóndito de las capas de nuestra lógica. En esta discusión hicimos un paréntesis para recalcar la importancia de las expresiones en la claridad y expresividad que aportan a los algoritmos; y hablando de *SQL* y manipulación de datos, probablemente el tipo de expresión más importante que tenemos sea la consulta empleada para extraer información de la base de datos. Empleando una frase más que gastada "mucho dinero, tiempo de investigación y por supuesto esfuerzo" se ha invertido en hacer de las consultas de *SQL* un sublenguaje poderoso, eficiente y funcional para explotar la información. Probablemente sea de los rubros que más preocupan a los fabricantes de SBMD, evidenciado por la gran cantidad de literatura, tanto práctica como teórica, que existe en torno a esta parte de *SQL* (uno podría dedicarse, exclusivamente, al problema de la optimización de las consultas *SQL* a nivel investigación y tener bastante para entretenerse una gran cantidad de años).

Además, es muy probable que el diseño de nuestro modelo de datos esté expresado en términos de relaciones (que encarnarán en tablas). Incluso siguiendo un paradigma OO, con un modelado como UML, habrá una parte donde necesitemos de la representación estática de la información y en ese lugar será natural pensar en los accesos a la información usando *SQL* (o en su defecto, alguna versión del álgebra relacional, fundamento teórico de las consultas *SQL*). Pero si no queremos usar *SQL*, por no querer depender de los detalles del fabricante, ¿cómo supliremos esta parte del estándar? Las consultas pueden ser sofisticadas construcciones (expresiones), con agrupaciones, subconsultas, condicionales, operadores relacionales, etc. Entonces, nos encontramos ante dos posibles salidas: o buscamos una manera de traducir dichas consultas *SQL* en términos de invocaciones a servicios de nuestros objetos, o bien, reinventamos la rueda y creamos otro lenguaje para acceder a los datos.

Ambas soluciones suenan poco factibles, dado su carácter de "no triviales"; inclusive, al final acabarían dependiendo del manejador que estuviésemos usando (en mayor o menor medida) puesto que el implementar su propio SBMD definitivamente queda muy lejos del alcance de cualquier proyecto realista de *SI*. De las dos alternativas, probablemente la más prometedora sería la primera, pero aunque lográsemos implementar únicamente un subconjunto útil del lenguaje de consulta

SQL, a través de nuestras clases, no estaríamos exentos de perder la declaratividad: el símil de esta elección sería equivalente a programar en un lenguaje X, ingresando al compilador como entrada la estructura de datos que resultaría de análizar sintácticamente el código, en lugar de comunicarnos directamente a través de la sintaxis. Aunado a ello, como no estaríamos soportando toda la funcionalidad de *SQL*, tendríamos que suplirla con más lógica imperativa en nuestros programas lo cual seguramente haría más compleja la implementación de las reglas del negocio (sería como boxear usando únicamente golpes rectos, posible tal vez, pero difícil no admitir la incomodidad que implicaría no poder recurrir a todo el repertorio).

3.2. La opción de SIUX: SQL99-PSM

Suficiente por ahora de estas opciones que tenemos para hacer frente a los problemas presentados para desarrollar *SI*. Dijimos que es difícil (si no imposible actualmente), diseñar y programar en *SQL* estándar (por la falta de uniformidad en las implementaciones). También mencionamos nuestra falta de simpatía con la opción actual, de tirar a la basura gran parte de funcionalidad que ofrece el estándar, en aras de ser portátiles. Una solución ideal podría ser el contar con algún traductor de *SQL* estándar a los dialectos particulares de cada fabricante: pero tampoco se visualiza como una ruta de escape a corto plazo. Entonces, este trabajo se presenta como un camino alterno, tomando un camino pragmático. Si a final de cuentas, a pesar de los esfuerzos, es difícil no terminar dependiendo de tal o cual atributo de los SMBD particulares, ¿por qué no aceptar dicho matrimonio, y llevarlo hasta sus últimas consecuencias? Obviamente, tendrían máxima prioridad los aspectos que sean cien por ciento compatibles con el estándar, recurriendo a las otras opciones cuando no quede otra salida.

¿Qué ganamos con esta opción? Al aceptar que deseamos aprovechar todo lo que nos ofrezca *SQL*, y resignarnos a que nuestro código no será directamente portátil (aunque tampoco extremadamente difícil de migrar) llegamos a una conclusión interesante: ya no es necesario programar en un lenguaje externo. ¿Para qué querríamos usar un lenguaje como C, Java o Cobol, cuando lo único que harían es invocar todo el tiempo al SMBD? De hecho, sería una pérdida de tiempo sacar los datos del ambiente de ejecución de la base de datos, para regresarlos inmediatamente en la siguiente instrucción *SQL*. Y en este punto es donde surge de manera más o menos natural (esperamos), la justificación del módulo *PSM* del estándar: si ya aceptamos a *SQL*, con todos sus defectos y virtudes, entonces llevemos el paquete completo. Llevémonos también este lenguaje que tiene contacto directo con *SQL* (de hecho, es una extensión imperativa del mismo), e implementemos en él nuestra lógica del negocio. Hay que tener cuidado, ya que este lenguaje, al ser imperativo (y con variables y asignaciones), nos podría llevar a los viejos vicios y desperdiciar la naturaleza declarativa de otras partes de *SQL*; pero si resistimos estos malos hábitos, podremos

ver que *PSM* únicamente nos servirá para usar las estructuras de control y darle vida directamente a las reglas del negocio; cualquier validación, cálculo, expresión o extracción de datos que necesitemos, la podemos hacer aprovechando las expresiones *SQL*, que ahora son directamente accesibles y están integradas en el lenguaje.

Aparte, recordemos que una de las ventajas de este lenguaje (aparte de su contacto tan íntimo con *SQL*), es que si bien imperativo, contiene algunas armas relativamente modernas de los lenguajes de programación, como lo son las excepciones. En la capa 1 tratamos de implementar la mayor cantidad posible de validaciones (las invariantes); y a través de la infraestructura SIUX, estas se propagarían como excepciones ante los eventos que las intentaran violar (alguna operación que modifique las tablas, seguramente). Ello nos evitaría minar nuestro código con validaciones manuales de los errores, aunque como un efecto no tan agradable o natural, nos podría obligar a adelantar las afectaciones a las tablas lo más que sea posible, para activar en la misma medida las alarmas instaladas una capa abajo en nuestra arquitectura. Esto último no es tan difícil de lograr, ya que, despreocupados de gran parte del manejo de errores y armados con expresiones *SQL* (incluyendo claro, las consultas *SQL*), complacientemente observaríamos como la lógica de nuestros procedimientos almacenados reflejan de una manera más clara la regla del negocio que pretende modelar. Así que no habría muchas razones para retardar las afectaciones a la base de datos; hagámoslas sin temor, que una capa abajo nos aguarda para validar los datos.

Todos los párrafos que precedieron al presente, están unidos en el propósito de justificar el lenguaje propuesto: *SQL-PSM*. Pasemos ahora a continuar con nuestro ejemplo que nos lleva de paseo por todas las capas de la arquitectura: el mantenimiento de la hipotética tabla *mtto_bitacora*. Como se trata de un simple mantenimiento, digamos que fue posible delegar todas las validaciones en la capa 1, así que en la presente sólo nos resta invocar directamente las instrucciones *SQL* para el *insert*, *update* o *delete*. En este sentido, para los mantenimientos de catálogos simples (o tablas con usos similares), se convierten en meros puntos de acceso. Sin embargo, son necesarias, ya que una de las premisas de la presente propuesta es mantener una homogeneidad en todo el sistema. No es válido que de repente llegue un programa, y por considerar a los mantenimientos triviales, se olvide de las capas intermedias y trabaje directamente con *SQL*, en lugar de hacerlo a través de un *WebService*, o en su defecto (si vive dentro del *backend*), a través del correspondiente procedimiento *PSM*.

Entonces, el código para un alta de actividad, por ejemplo, se vería reducido al mínimo, como lo mostramos en el siguiente listado:

```
create or replace procedure mtto_bitacora_add
(
```

```
  in_numero          in numeric,
  in_tipo            in numeric,
  in_fecha           in date,
  in_duracion_hrs    in numeric,
  in_descp           in varchar
)
as
begin
  insert into mtto_bitacora
   (
    numero,
    tipo,
    fecha,
    duracion_hrs,
    descp
  )
  values
   (
    in_numero,
    in_tipo,
    in_fecha,
    in_duracion_hrs,
    in_descp
  );

  exception
    when others
      siux1_error.handle_error ();
end mtto_bitacora_add;
/
call
  siux2_par.add_parinfo
   (
    'mtto_bitacora_add',
    'in_numero',
    'mtto_bitacora',
    'numero'
  );

...

call
  siux2_par.add_parinfo
   (
    'mtto_bitacora_add',
    'in_descp',
    'bitacora',
    'descp'
  );
```

Es verdad, lo admitimos, un alta en la tabla no es precisamente el lugar donde saldrían a relucir las ventajas de usar *PSM* para implementar la lógica (ya que en este caso, ni siquiera hay tal lógica). Sin embargo, nuestra intención de momento es recorrer todas las capas de la arquitectura y hacer énfasis en la tecnología y convenciones, y no en alguna implementación de la vida real, cuya complejidad pudiera distraer de este objetivo. Sin embargo, hay algunos detalles a notar en

este listado, mismos que comentamos a continuación:

1. Independientemente de que el lenguaje (siguiendo al estándar), ofrece la posibilidad de indicar el *modo* del parámetro (entrada, salida o entrada-salida) nosotros, como parte de la propuesta, también usamos un prefijo en el nombre del parámetro, que debe estar acorde al modo que le sigue. Para los parámetros de entrada-salida (se usó uno en el ejemplo de la generación automática de folios en el capítulo anterior) *PSM* pide usar la claúsula "in out", mientras que nosotros usaremos el prefijo abreviado *io_*.

2. A diferencia de los enunciados *SQL-DDL* que sirven para definir la estructura de las tablas, mismos que contienen precisión para las variables numérica y alfanuméricas, en las declaraciones de los procedimientos nos abstenemos de indicar ese detalle. Ello inmediatamente prende alguna campana interna, y nos remonta a los días donde veíamos los métodos de paso de parámetros, concretamente al paso por referencia: si el compilador (incluído en el SMBD) de *PSM* no requiere de esa información, seguro pasa las variables por referencia. Aunque puede ser natural esta sospecha, también podría ser un tanto precipitada, toda vez que ese detalle no debería importarnos directamente, ya que es el modo de los parámetros (*in,out o in out*) el que indica si los mismos tendrán accesos de lectura y/o escritura dentro de la definición del procedimiento.

3. Las variables del lenguaje, en este caso en su variedad de parámetros de entrada, pueden ser usadas directamente en la sintaxis de las instrucciones *SQL*-DML. Inclusive, podríamos llamarlas de la misma forma que las columnas de la tabla. La implementación del lenguaje sabrá reconocer que los nombres en la primera sección del *insert* corresponden a nombres de columnas y que los segundos deberán buscarse en el espacio de variables del programa en ejecución. Cuando llegáramos a tener ambigüedad en los nombres, la prioridad la tendrán los objetos de la base de datos (en una instrucción *update*). A pesar de esta facilidad, para algunas personas podría resultar confuso el repetir los nombres; pero ello no justifica el uso de variables intermedias, si el contexto resuelve la duda nos abstendremos de esta práctica.

4. La última instrucción, corresponde al manejo de excepciones en la implementación *PSM* que seleccionamos: Oracle-PL/*SQL*. Con ella estamos diciendo que cualquier excepción que se llegara a producir, provocará la invocación de la rutina *siux1_error.handle_error*, misma que se encargará de atrapar la excepción nativa del manejador, de traducirla al par *(código,mensaje)* asociado en la capa 1 y de repropagar el evento como otra excepción.

5. No hay necesidad de regresar explícitamente un código y un mensaje de error; lo que regresamos, en caso de contingencia (con ayuda de la la infraestructura) es una excepción; y ella tiene asociada dicha información. Sin embargo, adelantándonos un poco, cuando salgamos del ambiente del ejecución de *SQL* que nos ofrece el SMBD, y exportemos esta funcionalidad en la capa 3, entonces sí se podrían incluir como parámetros de salida estos datos; sin embargo, en cuanto a carácteristica común de todos las rutinas, este agregado lo puede hacer automáticamente SIUX.

6. Los procedimientos que estamos invocando al final (y que sólo mostramos parcialmente), sirven para asociar metainformación con los parámetros del procedimiento; porque al igual que con la capa anterior, aquí también tenemos catálogos para tales fines, que posteriormente serán explotados para la publicación de los servicios del *backend.* Los parámetros que estamos pasando, son, en ese orden: nombre del procedimiento, nombre del parámetro, nombre de la tabla y nombre de la columna. Con ello estamos diciendo que deseamos reusar la información que ya proporcionamos en la capa 1, puesto que los parámetros de este procedimiento tienen una correspondencia directa con las columnas de la tabla *mtto_bitacora.* La rutina *siux2_par.add_parinfo* tiene más parámetros que permiten poner una etiqueta y descripción, en caso de que no se desee aprovechar información de los catálogos de la capa 1. A pesar de esta previsión, podemos asumir que la mayoría de los parámetros de los procedimientos no la necesitarán; aunque sí podría darse el caso de que un parámetro usara ambas cosas: la asociación con un par *(tabla,columna)* para efectos de validación, así como una etiqueta y/o descripción propia (por ejemplo, un parámetro que representa el extreno de un rango de valores para una columna será del mismo tipo que la misma, pero seguramente querremos que su nombre sea distinto).

3.3. Búsquedas en PSM bajo SIUX

Probablemente, la curiosidad natural en el lector llevaría a la pregunta esperada: ¿qué pasa con las búsquedas? ¿cómo se regresan los resultados? SIUX soporta, de momento, únicamente tipos de datos que correspondan directamente al modelo relacional, esto es, registros y listas de registros. Bajo este marco, podemos visualizar la firma de las funciones como la declaración de dos tipos de datos estructurados: un registro para la entrada y otro para la salida. En la entrada, únicamente damos permiso a datos simples (sin estructura), e incluso nos restringimos a tres únicamente: números (incluye los enteros y los de precisión arbitraria), fechas (incluye *timestamps*) y cadenas; representados respectivamente por los tipos Oracle-*SQL*: *numeric*, *date* y *varchar.* Concentrando nuestra atención en aquéllos parámetros que corresponderían al registro de salida, mencionaremos que SIUX sólo soporta el regreso de un parámetro de salida, con tipo "lista de registros", y que es modelado con una referencia de un cursor abierto. Por ejemplo, supongamos que también de-

seamos un servicio de consultas de las actividades, aceptando como criterio de entrada un rango de número, de fechas y un patrón para la descripción. El siguiente listado muestra un ejemplo de implementación para SIUX:

```
create or replace procedure mtto_bitacora_con
(
  in_numero_min in   numeric,
  in_numero_max in   numeric,
  in_fecha_min  in   date,
  in_fecha_max  in   date,
  in_descp      out  varchar,
  out_cur       out  sys_refcursor
)
as
begin
  open cursor out_cur for
    select
      *
    from
      mtto_bitacora
    where
       (numero between coalesce (in_numero_min,-.999999999999e12)
                   and  coalesce (in_numero_max,+.999999999999e12))  and
       (fecha  between coalesce (in_fecha_min,
                          to_date ('0001-01-01 00:00:00',
                                   'yyyy-mm-dd hh24:mi:ss'))
                   and coalesce (in_fecha_max,
                          to_date ('9999-12-31 23:59:59',
                                   'yyyy-mm-dd hh24:mi:ss'))) and
      upper (descp) like upper ('%'||in_descp||'%');

  exception
    when others
      siux1_error.handle_error ();
end mtto_bitacora_con;
/
```

La referencia al cursor que mencionamos se llama en este ejemplo *out_cur* (nombre genérico que usarán todos los procedimientos que pretendan regresar una lista de registros). Corresponde a un parámetro de salida, y será responsabilidad de la capa 3 usar dicha referencia para extraer la información y presentarla de una manera adecuada hacia el exterior. Prácticamente todos los lenguajes que tienen acceso a *SQL*, ofrecen una manera para llevar a cabo esta tarea: si estamos hablando de C, podríamos usar *SQL* incrustado para realizar una secuencia iteración del estilo: *open-fetch-close*; y si estamos con Java (que fue nuestra elección para SIUX), podemos aprovechar el manejador JDBC (*Java DataBase Connectivity*) que ya nos ofrece al mismo como un objeto *ResultSet* (también parte del estándar de iteracción *SQL*-Java), y con ello nos ofrece las facilidades esperadas para hacer la iteración, y mejor aún, nos da metainformación sobre las columnas

(nombre, tipo, posición etc).

El lector podrá notar que, a manera de proceso alterno corriendo en paralelo, continuamos promoviendo el estilo declarativo-funcional. Esta consulta podría haber sido el escenario para poner algo de enunciados de control que asignaran los parámetros de búsqueda en variables; pero preferimos posponerlo todo a la condición de la consulta, resultando en que la poca lógica que tenía el algoritmo a implementar, quedó reducida a una sola expresión (con excepción del manejo de la excepción). Estamos conscientes de que el estilo funcional de programación puede resultar extraño a las personas que llevan mucho tiempo programando imperativamente (en el mercado laboral podemos encontrar a programadores con más de 20 años de experiencia). Pero nuestra apuesta es por evitar las variables, a menos que su presencia sea justificada (aunque este tema, podría entrar en una discusión sobre los valores estéticos o la elegancia de los programas, lo cual definitivamente ya no trataremos, por pertenecer al mundo de las preferencias personales). Como también fue mencionado, toda vez que sea posible, tendrán prioridad las funciones que formen parte del estándar; ésta fue una de estas veces, incluyendo a la versión infija de las funciones (operadores).

La mención del lenguaje Java como opción de implementación que acabamos de hacer en párrafos precedentes, no debe confundir los papeles de este lenguaje y de *SQL-PSM* en la presente propuesta. En este documento, estamos explicando (o intentamos al menos) dos aspectos en paralelo, que representan entidades distintas (aunque relacionadas). Estamos presentando SIUX, que es una amalgama de infraestructura más ciertas convenciones. Dicha infraestructura sigue una arquitectura de cuatro capas, en cada una de las cuales se implementaron ciertos servicios para hacer factible la propuesta; y cada una de esas implementaciones fue elaborada en el lenguaje que se juzgó más adecuado (para las capas 1 y 2, *SQL-PSM*; para la tercera y cuarta: *Java*, *XSLT* y *XForms*). Por otro lado, tenemos el lenguaje propuesto para implementar los SI particulares, auxiliados por esta infraestructura: capa 1 y 2, con *SQL-DDL* y *PSM*; capa 3 no requiere programación y capa 4 tampoco, a menos que las interfaces generadas automáticamente no satisfagan, en cuyo caso se podría codificar manualmente en *XForms*. En los capítulos correspondientes a las capas 3 y 4, se hablará más de estos lenguajes relacionados con *XML* (*XSLT* y *XForms*), aunque de momento, nos pareció oportuna esta intervención aclaratoria.

3.4. Mantenimientos gratis con SIUX

Dada la trivialidad del código de mantenimiento, las probabilidades de que generen tedio en los programadores son bastante altas, y ello aumentaría el riesgo de que sintieran deseos de romper los acuerdos entre las capas de la arquitectura, concretamente de la que dicta que "toda fun-

cionalidad del backend, deberá existir como un procedimiento almacenado PSM", característica que le permitirá ser publicado automáticamente, como un *WebService*. Entonces, para evitar esta tentación, SIUX ofrece un servicio de generación de estos procedimientos para mantenimiento de tablas. Una invocación como la que sigue generaría el código necesario:

```
call siux2_init ('mtto','es');

call siux2_mtn.gen4table ('mtto_bitacora');
```

Nótese que antes de invocar al generador de código, se está inicializando la infraestructura para la capa 2 (indicando nombre de la aplicación e idioma asociado a los mensajes). De hecho, antes de usar cualquier método de la infraestructura SIUX, hay que inicializarla para la capa en cuestión (ello aplica para las dos primeras capas). La rutina invocada en segundo lugar generaría procedimientos como los dos presentados anteriormente (para altas y consultas), junto con otro par (para bajas y actualizaciones); a todos ellos, se les pondría como nombre la tabla y algún sufijo que indique la operación: *add*, *con*, *del* y *upd*, respectivamente. Sin embargo, habría algunas variantes, pero ilustrémoslas con el código generado por delante (exceptuando la indentación, por supuesto):

```
procedure mtto_bitacora_add
(
  in_numero         in  numeric,
  in_tipo           in  numeric,
  in_fecha          in  date,
  in_duracion_hrs   in  numeric,
  in_descp          in  varchar,
  out_numero        out numeric,
  out_tipo          out numeric,
  out_fecha         out date,
  out_duracion_hrs  out numeric,
  out_descp         out varchar
)
as
begin
  insert into mtto_bitacora
   (
    numero,
    tipo,
    fecha,
    duracion_hrs,
    descp
  )
  values
```

```
  (
    in_numero,
    in_tipo,
    in_fecha,
    in_duracion_hrs,
    in_descp
  );
  returning
    numero,
    tipo,
    fecha,
    duracion_hrs,
    descp
  into
    out_numero,
    out_tipo,
    out_fecha,
    out_duracion_hrs,
    out_descp
end mtto_bitacora_add;
/
```

Ahora sí, pasemos a comentar algunos detalles en cuanto al generador de procedimientos de mantenimiento:

a) El procedimiento incluye todos las columnas de la tabla, como parámetros de salida. La razón de ello es que la tabla podría tener definidos procedimientos para calcular columnas derivadas (en la capa 1), y al hacer una inserción, algunos de los valores indicados podrían cambiar de valor. El recurso común para estos casos en otros lenguajes, sería la siguiente sucesión de pasos: generar por adelantado la llave primaria (el siguiente número de actividad, en este caso), hacer la inserción y posteriormente hacer un *select* con la llave empleada para ver si cambió algún campo. En nuestro caso, estamos aprovechando que la capa 1 podría definir la llave (se presentó un ejemplo en el capítulo anterior, con una secuencia y un procedimiento); así como la facilidad de Oracle para regresar al mismo tiempo de la inserción, todas las columnas (muy bien, esta característica no es parte del estándar ... ¡pero debería serlo!, es bastante práctica).

b) El código generado para el procedimiento no incluye la sección que atrapa las excepciones e invoca al manejador de SIUX para las mismas. La razón de ello radica en las fronteras con la capa siguiente (capa 3). El hecho es que no podemos (o no creímos conveniente) publicar los procedimientos almacenados en *WebServices*, directamente en su forma actual; a pesar de las convenciones usadas, podría haber cierta heterogeneidad en los mismos. Nos pareció más claro y limpio generar la interfaz a partir de una sola aglomeración de procedimientos, que pertenecieran a un único paquete por aplicación.

Hablaremos más de esto párrafos adelante, pero por ahora diremos que es en ese paquete homogéneo, agrupador de todos nuestros procedimientos, donde tiene lugar la inclusión de las instrucciones para atrapar y repropagar las excepciones.

c) Los generadores de código de la infraestructura SIUX, que no son muchos e incluyen al mantenimiento, no usan archivos como intermediarios. Guardan el código en variables cadena y lo inyectan directamente a la base de datos. Sin embargo, ello no debe preocupar, ya que el manejador guarda los listados de los procedimientos en catálogos internos (*dba_source*, para Oracle).

d) Ya no lo mostramos para evitar un exceso, pero el código para el procedimiento de consulta incluiría parámetros de entrada adicionales: de hecho, todas las columnas de la tabla tratarían de ser usadas como parámetros de búsqueda en forma de rangos (con excepción de las cadenas muy grandes, que se usarían como patrones para cazar con el operador *LIKE*).

e) Nadie ha dicho que todos los mantenimientos posibles serían satisfechos con este código tan simple; pero estamos seguros de que satisfarían muchos casos reales. Si surgieran necesidades muy minuciosas para una tabla que se escaparan de estas consideraciones, se generarían manualmente los procedimientos (siguiendo los lineamientos mencionados) y listo; se puede usar el resto de la infraestructura sin problemas. También es claro que los procedimientos que implementen la lógica de las reglas del negocio caerían en esta categoría.

3.5. Interfaz de la capa 2

Terminemos de platicar los detalles de esta capa (la interfaz homogénea que salió a relucir en la lista anterior). Una vez que tuviéramos todos los procedimientos listos para nuestra aplicación, pasaríamos a seleccionar aquéllos que deseamos exportar hacia el exterior (muchos seguramente serán de uso interno: métodos privados o protegidos, usando terminología OO). En nuestro ejemplo de mantenimiento de tabla, los procedimientos fueron automáticamente generados por la infraestructura, y en otros casos menos triviales, seguramente invertiríamos más tiempo en implementarlos. El asunto es que, una vez que contemos con ellos, tenemos que agruparlos en un solo paquete. Los paquetes son una construcción de Oracle que permite aislar lógica (procedimientos) y variables, de una manera similar a las clases de la OO: tenemos encapsulamiento de datos y la asociación de lógica relacionada con ellos.

Entonces, nuestra labor en la capa 2 debería ser terminar con ese paquete, que deberíamos llamar con el código genérico de nuestra aplicación (el mismo utilizado para los catálogos internos

de SIUX). Los paquetes tienen una declaración y su definición, donde únicamente tendríamos que poner invocadores a nuestros procedimientos originales. Por ejemplo, para nuestra aplicación que tiene como única meta darle mantenimiento a la tabla *mtto_bitacora*, bastaría algo así:

```
-- declaración
create or replace package mtto as
  procedure bitacora_add
   (
    in_numero          in  numeric,
    in_tipo            in  numeric,
    in_fecha           in  date,
    in_duracion_hrs    in  numeric,
    in_descp           in  varchar,
    out_numero         out numeric,
    out_tipo           out numeric,
    out_fecha          out date,
    out_duracion_hrs   out numeric,
    out_descp          out varchar
  );

  ...
end mtto;
/

-- definición
create or replace package body mtto as
  procedure bitacora_add
   (
    in_numero          in  numeric,
    in_tipo            in  numeric,
    in_fecha           in  date,
    in_duracion_hrs    in  numeric,
    in_descp           in  varchar,
    out_numero         out numeric,
    out_tipo           out numeric,
    out_fecha          out date,
    out_duracion_hrs   out numeric,
    out_descp          out varchar
  )
  as
    mtto_bitacora_add
     (
      in_numero        ,
      in_tipo          ,
      in_fecha         ,
      in_duracion_hrs  ,
      in_descp         ,
      out_numero       ,
      out_tipo         ,
      out_fecha        ,
      out_duracion_hrs,
      out_descp
    );

    exception
```

```
      when others
        siux1_error.handle_error ();
  end bitacora_add;
end mtto;
/
```

Nuevamente, la simplicidad del código para este paquete es tal, que amerita su generación automática por parte de SIUX. Con la siguiente invocación, logramos un código semejante al anterior (que por cierto, mostramos abreviado, para evitar más aburrimiento del necesario):

```
call siux2_init ('mtto','es');

call
  siux2_ifc.backend_interface
   (
    'mtto',
    'mtto_bitacora_add'
  );

...

call
  siux2_ifc.backend_interface
   (
    'mtto',
    'mtto_bitacora_con'
  );
```

Con las pasadas invocaciones, estamos creando y agregando al paquete *mtto* (nombre de nuestra aplicación ejemplo) los procedimientos que deseamos formen parte de la interfaz del *backend* de nuestro SI. La única mención adicional que merece este punto es el recordatorio, de que es en ese código generado por la rutina *siux2_ifc.backend_interface* donde se agregan las secciones que manejan los errores y excepciones.

Despediremos a esta capa 2, presentando una terna de funciones auxiliares que nos servirán en la siguiente capa. Estamos hablando de las funciones *str2num*, *str2date* y *str2str*; todas ellas, pertenecen al paquete *siux2_par*, que es parte de la infraestructura SIUX para este nivel. La intención de estas funciones es centralizar todo lo que sea validaciones. Si hacemos una pequeña retrospectiva, veremos que en todo momento asumimos unos procedimientos que ya tienen tipados sus parámetros de entrada (reciben directamente números, fechas o cadenas). Sin embargo, para que estos procedimientos puedan recibir los datos de esa forma, se necesita que alguien convierta las cadenas que vendrán de la interfaz de usuario (la cual no tipifica los datos) y haga la conversión

(que los números tengan el signo y el punto donde deben, la cantidad necesaria de decimales, que las fechas sean válidas, etc. Afortunadamente, todo eso ya lo hace el SMBD por nosotros, pero hay que pedirlo).

Esta conversión correrá a cuenta de la siguiente capa, pero queremos que las reglas para hacerlo sean las que dicta la capa actual. Aquí es donde definimos los datos, aquí es donde sabemos cuánto miden las columnas de las tablas, qué tipo tienen, qué precisión, etc. Como no queremos segmentar dichas reglas a lo largo de dos capas de la arquitectura, lo que hacemos es implementarlas aquí, y pedirle a la capa vecina que simplemente las invoque y se olvide del asunto. Retomaremos este tema en el siguiente capítulo (aunque sólo sea someramente).

3.6. Control de transacciones SQL

Una inquietud válida que pudiera surgir, al ser considerada la posibilidad de utilizar una infraestructura como la presente, es el manejo de aquella parte de los programas, que tiene que ver con el bloqueo a los recursos, o bien, con el nivel de aislamiento que tendrán los programas con respecto a los cambios que hagan los demás. Sin embargo, es probable que esta inquietud surja de costumbre con la arquitectura de dos capas que usualmente manejan muchos sistemas: el *frontend* realiza una conexión directa con el *SMBD* y es este último quien se encarga de ejecutar algunas de las instrucciones para el control de transacciones (atomicidad de un conjunto de instrucciones, por describirlas de manera resumida).

La propuesta que estamos presentando no intenta cambiar o resolver de distinta manera los problemas asociados a este control. Simplemente se presenta una alternativa para el lenguaje en el cual implementamos las reglas del negocio. La única diferencia, a manera de ventaja, sería que podemos convivir más estrechamente con *SQL*. Sin embargo, los mismos mecanismos usados en otras arquitecturas para solventar problemas de concurrencia de procesos, deberán ser usados o por lo menos considerados.

Mencionemos por ejemplo, una pregunta que fue realizada al autor de estas líneas durante la revisión *SIUX*: supongamos que tenemos una aplicación estilo cajero automático, y que consultamos un saldo, vemos una cierta cantidad y posteriormente deseamos sacar cierto monto en base a esta primera consulta. ¿Cómo garantizar la consistencia de los datos, es decir, que efectivamente podamos sacar lo que vimos en la pantalla sin que otro proceso interfiera? Aquí hay varias cosas por aclarar. Primero, una aplicación de cajero automático tanto en un sistema bancario tradicional como en la propuesta *SIUX*, maneja una arquitectura de por lo menos 2 capas, de las cuales la interfaz del cajero sólo es la primera. El *backend* ofrece programas para las dos operaciones

mencionadas: consulta y retiro.

Una característica importante de estos sistemas por capas, donde las interfaces de usuario únicamente muestran o piden datos (sin involucrarse en su semántica), es que los servicios del *backend* son independientes entre sí. Esto quiere decir que no existe ningún contexto previo a la invocación de la consulta o el retiro, que tenga que ver con alguna sesión del usuario con el sistema. Así, el servicio de retiro no tiene por qué asumir que el usuario consultó su saldo previamente, y por lo tanto deberá realizar sus propias validaciones. Y es en ese programa, en el retiro, donde al consultar el saldo se bloquerá el registro para garantizar que nadie más lo pueda modificar hasta que él haya terminado.

Otra opción para garantizar la integridad de los datos, sería que el bloque de instrucciones *SQL* del programa de retiro tuvieran un nivel de aislamiento (*isolation level*) que indicara serialización. Así, garantizaríamos que, entre la lectura del saldo (esta vez sin bloqueo) y su actualización no se interponga ninguna otra operación, de algún otro programa. Para el caso concreto de *SIUX*, este control no se realiza en la capa que habla directamente con el *frontend*, ya que ahí únicamente ofrecemos un servicio de conversión de datos (usando *XML*). Es hasta la capa 2, discutida en esta sección, donde se incluirían ambos servicios, implementados como procedimientos almacenados y donde se incluría ya sea la cláusula *SELECT ... FOR UPDATE* para bloqueo por registro, o bien, para el caso concreto de Oracle *PL/SQL* se indicaría el nivel de aislamiento máximo (*SERIAL*) con la instrucción *SET TRANSACTION*.

Otra pregunta válida, aunque un tanto desviada de los aspectos que cubre la propuesta, podría ser: ¿cuál esquema es mejor, bloqueo por registro o nivel de aislamiento serial? Nuevamente, usar una propuesta como *SIUX* no cambia, ni mejora, ni empeora este tipo de situaciones; en tanto que estemos manejando manipulación simultánea de datos a través de un *SMBD*, estas preguntas surgirán (independientemente de que usemos el estándar *SQL99-PSM* o algún otro). Además, la respuesta a esta pregunta no puede ser universal dado que depende del patrón de acceso que tengamos. En nuestro ejemplo de un programa para actualizar el saldo, suele bastar con bloquear el registro, porque ese tipo de operaciones bancarias usualmente involucran inserciones de registros para tener memoria de la afectación (movimientos) y éstos se pueden hacer en paralelo sin problema, así que no tiene caso serializarlos.

Capítulo 4

Capa 3: Interfaz del backend

4.1. Antecedentes: Servidores de Aplicaciones (SA)

Recapitulemos un poco, para apreciar mejor la naturaleza de esta capa 3 (y la siguiente). Hasta ahora, hemos visto lo que nos ofrece la infraestructura SIUX para las dos primeras capas, y hemos ejemplificado, aunque sea someramente, como podríamos aprovecharla en esos niveles. Podríamos decir que el *trabajo sucio*, el que depende estrictamente del problema particular que estemos tratando de resolver, queda enterrado en las capas 1 y 2 de nuestra arquitectura. Como resultado, a estas alturas contamos con un paquete en el SBMD, que agrupa y unifica toda la funcionalidad de nuestra aplicación (materializada a través de procedimientos almacenados, codificados en *SQL-PSM*). Lo que resta, son dos tareas: publicar dicha funcionalidad a través de algún estándar que ofrezca mayor flexibilidad (nada más universal que WebServices), para después crear las interfaces de usuario hacia esos servicios. La primera es la tarea de esta capa.

Si de automatizar se trata, probablemente la capa 3 (y la capa 4) dejen un sabor de boca mucho más dulce que las precedentes. Y es que aquí sí podemos delegar prácticamente todo el trabajo a la infraestructura. Toda la metainformación que guardamos en las capas 1 y 2, es aprovechada para publicar automáticamente los procedimientos como *WebServices*. Entonces, en este capítulo nos remitiremos a justificar la elección de este estándar, aderezando con el contexto necesario para saborearlo: *XML*.

Recordemos la intención de esta tercera capa ¿por qué es conveniente?, ¿acaso no podemos invocar directamente los procedimientos almacenados y ya? Es verdad, podríamos invocar directamente a la lógica implementada en *PSM* no solamente desde *XForms*, sino desde algún otro lenguaje para creación de interfaces de usuario (bastaría el software de conexión a la BD). Pero hay una desventaja en esa elección, y es que ya soportamos la penosa (pero gratificante) elección de un SMBD, a cambio de poder aprovechar *SQL*. Ahora, si optamos por no superponer esta

capa 3, estaríamos dictando la tecnología que deberíamos usar para la siguiente capa, o bien, nos encontraríamos limitando el número de otros sistemas que pueden "conversar" con el nuestro.

No, definitivamente no podemos ni estamos dispuestos a ceder nuevamente en cuanto al uso de estándares; estamos a favor de los sistemas abiertos y tenemos una oportunidad de redimirnos. Implementaremos la lógica de nuestro negocio en *SQL*, pero ocultaremos este hecho lo mejor que podamos hacia el exterior; nadie tiene por qué saberlo. De hecho, ¿qué tal que aceptamos cualquier plataforma, como invocador? La opción de *WebServices* nos permite precisamente eso: no importa qué sistema operativo, lenguaje, manejador de base de datos, etc., se encuentre usando el sistema que desea comunicación con nosotros; mientras hable *WebServices*, podremos entendernos (ello incluye al software seleccionado para implementar las interfaces). De hecho, esta tecnología tiene como una de sus altas prioridades permitir la comunicación entre sistemas.

Además, existen otras justificaciones más ortodoxas para interponer una capa adicional, entre la funcionalidad en crudo de nuestro sistema y el exterior. La publicación de nuestros servicios, independientemente de que sea realizada usando algún estándar, requiere de ciertas facilidades comunes que no tendría caso implementar por separado en cada procedimiento; y aquí salen al rescate los sistemas conocidos como *Servidores de Aplicaciones* (*SA*). Estos metasistemas (sistemas que ayudan a construir otros sistemas), ofrecen el *pegamento* necesario para exportar servicios entre aplicaciones. Seguramente el lector ya ha escuchado hablar de ellos; los siguientes nombres podrían evocar ciertos recuerdos: CICS, EJB, Tuxedo, Contenedores de Servlets. Estos sistemas, algunos de los cuales también reciben denominaciones como *monitores de transacciones*, se encargan de permitir el acceso a nuestros servicios a gran escala, proporcionando las siguientes facilidades (sólo por mencionar las que vienen rápidamente a la mente):

0. **Homogeneidad:** A pesar de estar restringidos a cierta plataforma, podríamos tener varios *sabores* en la implementación de nuestra funcionalidad (situación que no es rara, en las empresas de la vida real, dada la heteregeneidad de los programadores que pasan por ahí). El seleccionar cierto servidor de aplicaciones, por lo menos garantiza que todas esas versiones se presenten ante el resto de los sistemas de una misma forma. Cuidado aquí: homogeneidad no implica estandarización (aunque combinando un servidor de aplicaciones con *XML*, podremos tener ambas cosas: esa es la oferta de los *WebServices*).

1. **Concurrencia, escalabilidad y desempeño:** Probablemente, la *concurrencia* sea una de las principales razones que orille a las personas a pensar en SA: la posibilidad de atender más de una invocación al mismo tiempo (en especial el aprovechar la infraestructura que administra los procesos o hilos involucrados). Porque el ahorrarnos la programación para estar creando, monitoreando, destruyendo, clonando, etc. los procesos (o alguna de sus variantes), en el siste-

ma operativo anfitrión, es ya una gran ventaja (y muchos dolores de cabeza menos).

Pero muy relacionada con la *concurrencia*, está la *escalabilidad* que consiste en la propiedad de un sistema para procesar más información sin necesidad de cambiar su lógica (únicamente, a través de la inclusión y configuración de más recursos físicos). Y cuando pensamos en más procesamiento de información, la vía natural es permitir que entren cientos o miles de peticiones simultáneas a nuestros servicios; de ahí la relación entre estas dos propiedades. También relacionada con la escalabilidad, está el *balanceo de carga* (distribución del trabajo en varios equipos). Ver más adelante en esta lista.

Cabe mencionar que la *escalabilidad* no tiene sentido sino pedimos cierto *desempeño* en el sistema. No tiene ningún caso aceptar simultáneamente muchas peticiones, si a final de cuentas todas pasan por un *cuello de botella* (las datos, por ejemplo). Entonces, a pesar de que los servidores de aplicaciones nos habilitan estas propiedades, requieren que la lógica de nuestro sistema tenga cierto grado de paralelismo.

2. **Accesibilidad:** Uno de los principales atractivos de los servidores de aplicaciones, es que nos permiten olvidarnos de los detalles de bajo nivel relacionado con las comunicaciones. En un escenario común, los usuarios de nuestros servicios (entendiendo aquí, otros programas) radicarán en otros equipos, y sería necesario usar software de comunicación para contactarnos. Aún aprovechando que las interfaces *TCP/IP* son bastante estables en nuestros días, su programación a nivel detallado, teniendo en cuenta todos los pormenores de las redes de computadoras; podría ser una tarea nada trivial (hay cantidad de literatura especializada al respecto).

Pero cobijándonos bajo un *SA*, podemos olvidarnos de programar los sockets directamente, y pensar a nuestros servicios únicamente como funciones; serán invocadas remotamente, sí, pero las podemos seguir viendo como meras funciones. Evidentemente, el personal de soporte sería el encargado de configurar al *SA* para lidiar con los parámetros de las comunicaciones; así que los programadores usualmente, pueden dormir tranquilos a este respecto.

3. **Administración de recursos compartidos:** El ofrecimiento de la funcionalidad del sistema a través de un *SA*, suele implicar que la lógica está implementada en el mismo lenguaje que usamos para la publicación (lo cual no es cierto para SIUX). Esta suposición implica, entonces, la necesidad de ofrecer acceso eficiente y centralizado a los recursos compartidos que pueden necesitar nuestros servicios. Las áreas de memoria compartida o las conexiones de red hacia otros sistemas, serían ejemplos muy comunes; de hecho, este último es el único que nos atañe

para nuestra propuesta.

Dentro del ambiente de ejecución de la base de datos, el manejador no da acceso a todos los objetos de la misma, sin ningún problema. Pero fuera de ésta, en algún lenguaje de programación externo, necesitaríamos una conexión con el SMBD y las conexiones son caras (abrirlas, mantenerlas y cerrarlas, implica un costo relativamente elevado para los participantes en la comunicación). Una alternativa para este problema, que puede aplicar para cualquier recurso *costoso*, es el tener sólo un número determinado de recursos (que puede oscilar dentro de cierto rango, en función de la demanda), calculado para los recursos disponibles. Cuando alguien pida un recurso, en lugar de crearlo, usarlo y destruirlo, lo toma de este depósito y lo regresa al término de su uso; evidentemente, la cantidad de peticiones simultáneas que soportaríamos con este esquema, es igual al tamaño de nuestra reserva (pensando en que los recursos con conexiones a la BD, usadas para invocar procedimientos *PSM*). El término común para esta estrategia es *Pool Connection*, y es ampliamente ofrecido por los servidores de aplicaciones.

4. **Balanceo de carga:** En los sistemas a gran escala, no es raro ver varios servidores que pretendan distribuir el trabajo del *backend*. Independientemente de que el *SBMD* podría por sí mismo implementar algún esquema de cómputo distribuido (permitiendo tener varios equipos físicos, dedicados al manejo de nuestra BD); los servidores de aplicaciones también podrían tener esta característica. El que sea un solo sistema (un *SA*) el encargado de recibir todas las peticiones hacia el *backend* y contestarlas, le da la posibilidad de redireccionar las mismas hacia los equipos disponibles, en función de la carga de trabajo que tengan los mismos (evidentemente, el *SA* deberá contar con algún mecanismo para medir el grado de ociosidad de los nodos administrados).

   En el caso de una propuesta como SIUX, donde el trabajo rudo es realizado por el SBMD y el SA sólo servirá para publicar la funcionalidad, es evidente que la distribución de la carga, de existir, tendría que ser a los niveles de las capas 1 y 2; así que convendría tener un servidor de aplicaciones disponible en cada equipo con el manejador de la BD, para que el controlador central, al delegar una petición dada, realmente segmente la carga: no serviría de nada tener N servidores de aplicación, cuando todos llaman a la misma instancia de SMBD. Mejor tener ambos en pares, instalados en todos nuestros equipos (así podrían procesar una unidad de trabajo completa).

5. **Seguridad:** Aunque usualmente la seguridad de los *SI* se implementa al nivel de las interfaces de usuario, un esquema consistente debería de colar estas prevenciones hacia las capas internas. A nivel de invocación de funcionalidad, los SA pueden habilitar la seguridad, permitiendo que sólo ciertos usuarios puedan invocar ciertos servicios (la asociación suele hacerse

declarativamente, mediante configuración).

Aunque SIUX no obliga a una implementación específica de la seguridad, recomienda nuevamente aprovechar las facilidades de *SQL*, ofrecidas extensivamente por el SBMD. Dichos sistemas, en particular la elección Oracle, contienen un alto grado de granularidad (apoyados en gran medida por el estándar), para la asignación de privilegios a los usuarios (que pueden ser agrupados en perfiles o roles). Entonces, bajo esta recomendación, la tarea de la capa 3 sería simplemente seleccionar la conexión a la BD, con el perfil asociado al usuario invocador (podría ser incluso, que el usuario empleado para autentificarse al sistema en su conjunto, fuera uno manejado directamente por el SBMD).

WebServices (*WS*) ofrece todas estas ventajas aunque podrían no ser su prioridad, pero el hecho de que se implemente encima de un servidor de aplicaciones existente, le da por herencia todas estas facilidades (recuerde que la prioridad para *WS* es facilitar la interoperabilidad entre sistemas). Los usuarios de estos sistemas podríamos ver que la combinación resultante asemeja un *SA* extendido. Por otra parte, tenemos una característica muy atractiva de *WS*: es un estándar ampliamente aceptado en la industria (aunque apenas está siendo descubierto o implementado por muchos). Una de las causas por las cuales el estándar promete mucho y es aceptado, es que emplea como lenguaje para expresar los datos lo que parece ser "la opción" para representar la información, y que llegó para quedarse: *XML*. Así que toca el turno de este documento para tratar de explicar los aspectos más generales de este otro estándar (obviamente no pretendemos suplir la cantidad tan extenuante de literatura, tanto impresa como en internet, que se puede encontrar al respecto; solo pretendemos dar un atajo).

4.2. XML: el estándar para representar datos.

Para el lector no versado en estándares de moda relacionados con la *Web*, ir al buscador de su preferencia y pedirle la información relacionada con *WebServices*, lo puede poner frente a una maleza impresionante de documentos, términos y estándares relacionados: no es muy difícil perderse. ¿Por dónde comenzar? Por las piedritas, diría la pedagogía: definamos *XML*.

A pesar de una creencia común, *XML* no es *un lenguaje particular*, sino un metalenguaje: esto es, un lenguaje que sirve para definir otros lenguajes. ¿Y qué tipo de lenguajes? Pues del tipo que usamos en el software para representar datos. Las raíces de *XML* (*Extensible Markup Language*), se remontan a otro metalenguaje que lleva un largo rato satisfaciendo los deseos de la industria editorial: SGML (*Standard Generalized Markup Language*). Dicho metalenguaje sirve para especificar la estructura de los documentos de las publicaciones, permitiendo separar en cierta medida

el contenido del *maquillaje*; los autores pueden teclear sus textos en alguna instancia de *SGML* (donde estén distinguidos, por ejemplo, unidades como capítulos, párrafos, títulos) y posteriormente la editorial procesaría dicho documento (siguiendo la definición de ese lenguaje particular) para darle el formato adecuado, previo a su impresión. Un hijo pródigo y mucho más conocido de *SGML*, es *HTML* (*HyperText Markup Language*), creado para representar documentos para la Web, y que tiene asociado cierta semántica de presentación (interpretada y procesada por los navegadores de internet).

SGML no define el significado de los lenguajes particulares que pueda generar, ello es tarea de los diseñadores de dichos lenguajes, mismos que plasman esa semántica en el software específico que los procesará. *SGML* sólo se encarga de marcar el aspecto léxico, sintáctico y parte del procesamiento de sus lenguajes hijos. *XML* es una simplificación de *SGML*, de hecho un subconjunto del mismo, especializado en la representación de datos para la *Web*, lo cual es un enfoque más genérico que simplemente pensar en documentos que serán impresos o visualizados (como suele suceder con *SGML*); aunque *XML* también puede tratar a los documentos con relativa facilidad.

Pero, ¿cómo lucen los lenguajes instancias de *XML*? Veamos un ejemplo relacionado con nuestro mantenimiento de tabla. Imaginemos que deseamos una estructura de datos genérica, que represente la información de una actividad de nuestra bitácora (misma que corresponde a un registro de la tabla *mtto_bitacora*). Si estuviéramos restringidos a cierto lenguaje de programación, donde resulta más familiar la idea de *estructuras de datos*, pensaríamos en algo así:

```
typedef struct
{
  int   numero;
  int   tipo;
  char  fecha[11];
  float duracion_hrs;
  char  descp[2049];
} mtto_bitacora;
```

Así lo haríamos en lenguaje C, pero nuestra intención es manejar una estructura de datos cuyo tipo que la defina sea *genérico*; esto implica que no deseamos depender de un lenguaje en particular. Si rescatamos algo de nuestras clases sobre estructuras de datos, podremos recordar que éstas se nos presentaron en abstracto (*ADT* por sus siglas en inglés), y que los lenguajes sólo fueron usados para ejemplificarlas. Así, tipos como los registros, listas, pilas, árboles binarios, etc., cobran vida por sí mismos y sobre pseudocódigo se pueden estudiar sus propiedades. El tipo que nosotros necesitamos, vendría siendo un *registro*, que consiste en una colección no homogénea de otros tipos, que pueden ser a su vez simples (como nuestro caso) o compuestos, y que son

accesibles a través de un nombre. Nuestro registro representa registros de una tabla de nuestro modelo de datos, y como en nuestras tablas no estamos recurriendo a tipos compuestos para las columnas (aunque el *SMBD* lo soporta), siempre tendremos registros con miembros simples (así lo pide *SIUX*, de momento).

Pero regresemos al problema que recién nos planteamos: ¿cómo representar el tipo que defina nuestra estructura de datos, sin depender de ningún lenguaje de programación en particular? Una opción podría ser, definirlo en un lenguaje que sea estándar y que tenga manera de interactuar con los diversos lenguajes de programación existentes. Es decir, que lo único que podemos hacer, es ofrecer de manera neutra la definición del tipo estructurado; pero su manipulación ya correrá a cargo de la programación usual (aunque siguiendo las pautas del estándar y de nuestra especificación). Eso es precisamente lo que nos ofrece *XML*, un terreno neutro que sirve para representar y estructurar nuestra información; junto con algunas interfaces estándar de programación que definen métodos para acceder las estructuras de datos representadas. Adicionalmente a estas reglas de interacción entre lenguajes existentes y el naciente *XML*, se han creado nuevos lenguajes de programación especialmente diseñados para manipular estos tipos de datos (*XSLT* sería un ejemplo notable, que volveremos a mencionar en el siguiente capítulo).

Entonces, supongamos que deseamos representar nuestros datos con *XML* (convencidos de que así, lo haríamos de manera neutral). Como *XML* es un metalenguaje, lo primero que tendríamos que hacer es definir el lenguaje particular para representar nuestro tipo registro. Esto lo podemos lograr, empleando la sección del estándar llamada *DTD* (*Document Type Definition*). Veamos una posible definición del tipo *mtto_bitacora*:

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<!-- Definición de la estructura de datos: mtto_bitacora -->

<!ELEMENT mtto_bitacora  (numero,tipo,fecha,duracion_hrs,descp)>

<!ELEMENT numero        (CDATA)>
<!ELEMENT tipo          (CDATA)>
<!ELEMENT fecha         (CDATA)>
<!ELEMENT duracion_hrs  (CDATA)>
<!ELEMENT descp         (CDATA)>
```

La primera línea del listado anterior, se llama "declaración del documento" y sirve para indicar que estamos por mostrar *XML* (en la versión y codificación de carácteres indicada). Luego tenemos un comentario y por último las definiciones de nuestro tipo: estamos diciendo cuáles son los componentes de una actividad de bitácora, y que éstos a su vez, tienen como componentes texto

(el equivalente aproximado a los tipos "cadena" de los lenguajes convencionales de programación). *XML* también permite definir atributos a los elementos que representan nuestros componentes, pero de momento no les prestaremos atención ya que no los usamos. Con esto hemos definido nuestro lenguaje particular, y ahora es justo preguntarnos ¿cómo se vería una instancia de nuestros datos, en este recién nacido lenguaje? Después de todo, en nuestra definición no estamos incluyendo de manera explícita la sintaxis para nuestro lenguaje; ello es innecesario porque *XML* ya la predefine (es parte del estándar). Esto quiere decir que todos los lenguajes *hijos* de *XML* lucirán muy parecidos (hermanos, al fin y al cabo); todos compartirán características léxicas y sintácticas que están predefinidas, y ello permite que las especificaciones de cada uno sean muy simples. Veamos un ejemplo donde se asume que pusimos el listado anterior en un archivo llamado *mtto_bitacora.dtd*:

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<!-- Una instancia de mtto_bitacora -->

<!DOCTYPE mtto_bitacora SYSTEM "mtto_bitacora.dtd">

<mtto_bitacora>
  <numero>666</numero>
  <tipo>3</tipo>
  <fecha>2004-06-27</fecha>
  <duracion_hrs>7.5</duracion_hrs>
  <descp>Documentación de la capa 3</descp>
</mtto_bitacora>
```

Hay varios detalles a recalcar de este ejemplo; hagámoslo puntualmente:

1. La apariencia común de todos los lenguajes creados con *XML*, consiste en un árbol de etiquetas (le decimos así, porque los elementos presentan anidamiento). Cada etiqueta tiene un nombre y está delimitada por los signos de mayor y menor que. Para que un documento *XML* esté bien formado, todas las etiquetas deben cerrarse (algo análogo a pedir que en las expresiones de los lenguajes, todos los paréntesis sean cerrados). Estas reglas son parte del estándar *XML*, y por ello no es necesario que las incluyamos en la definición de nuestros lenguajes particulares.

2. Tanto la sintaxis para el metalenguaje de *XML*, como para los lenguajes que genera, son parte del estándar y ambos se conocen (tal vez confusamente), como *XML*. De hecho, nótese que ambos tienen la declaración que indica tal situación, al principio de los listados.

3. Con la línea que contiene la leyenda *DOCTYPE*, estamos indicando que el documento que sigue pretende ser una instancia del lenguaje definido en el archivo *mtto_bitacora.dtd*. Si observamos la estructura de este *enunciado* o instancia de nuestro lenguaje creado, veremos que acata las reglas al pie de la letra: consta de un elemento llamado *mtto_bitacora*, que es un dato estructurado y tiene como miembros a todos los elementos componentes que indicamos, en el orden especificado (el orden es un agregado de usar *XML*, porque en los lenguajes de programación, suele ser indistinto quién aparezca primero en la lista de componentes de los registros). Entonces, podemos decir que nuestro documento no sólo está bien formado (sintácticamente correcto a bajo nivel), sino que es válido (sintácticamente, cumple las reglas estructurales impuestas por nosotros).

Existen especificaciones de interfaces de programación (*APIs*), que permiten acceder y manipular tipos y datos definidos en *XML*, a través de lenguajes de programación comunes (*DOM* y *SAX*, por ejemplo). Así, podríamos estar comunicando la información de nuestra capa 1, de una manera neutra entre los diversos componentes de la arquitectura (capas 3 y 4, para ser más precisos). Cualquiera que sea nuestra elección para procesar este documento (perdón, esta estructura de datos) nos deberá proporcionar automáticamente un servicio de validación, que indique si los datos que estamos recibiendo están bien formados, y son válidos (estructuralmente correctos). *XML* podrá parecer extenuante para las personas, pero para los programas es muy fácil de manipular y además, puede viajar en la red como texto común y corriente - que será analizado sintácticamente por el receptor; y posiblemente sea serializado nuevamente, en caso de ser reenviado a otro programa externo.

4.3. Estructuras de datos estándar: XMLSchema

Hasta este punto las cosas serían prometedoras (tenemos un lenguaje para representar de manera neutra nuestros datos), de no ser por un detalle bastante molesto: lo que en realidad estamos definiendo con *XML*, es un registro donde todos sus elementos (o componentes) son de tipo cadena. Al ver una instancia y observar los valores, podemos imaginar que ése representa un número o aquél una fecha, porque tenemos la estructura de datos tipificada en nuestra mente (aparte de que los nombres pueden ser sugerentes). Sin embargo, nada en *XML* está forzando esa situación y no tiene manera de auxiliarnos, si de repente un programa generara una instancia como la siguiente:

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<!-- Una instancia de mtto_bitacora -->
```

```
<!DOCTYPE mtto_bitacora SYSTEM "mtto_bitacora.dtd">

<mtto_bitacora>
  <numero>XXX</numero>
  <tipo>III</tipo>
  <fecha>200412/06-</fecha>
  <duracion_hrs>{7}.5</duracion_hrs>
  <descp>Documentación de la capa 3</descp>
</mtto_bitacora>
```

Si la miramos fijamente (bueno, no tanto), veremos que estructuralmente cumple con las reglas de nuestro lenguaje; a final de cuentas, el contenido de cada componente es una cadena. Pero nosotros sabemos que nuestra intención original, era que no sólo fueran cadenas, sino que además representaran números, fechas y demás valores de forma correcta. A pesar de que ya ganamos algo, la solución que de momento nos ofrece *XML* puro, no deja de ser incompleta. Usarlo así, nos obligaría a realizar la conversión de esas cadenas en tipos nativos del lenguaje, y ahí validaríamos todas las violaciones que se intentasen hacer (como las mostradas en este ejemplo). Como es de suponerse, no estamos descubriendo el hilo negro de nada (sólo tratando de explicar algo que ya se sabe); este problema se presentó hace muchos años y ya le dieron una solución que se muestra satisfactoria: si de estructuras de datos se trata (y no de meros documentos, donde todo son cadenas de carácteres), no usen *XML* puro, sino uno de sus lenguajes instancia (*XMLSchema*) que fue especialmente creado para tal fin.

XMLSchema (a veces abreviado *XSD*), es un lenguaje complejo y algo extenso, así que no pretendemos mostrar aquí una explicación detallada del mismo. Sólo daremos lo indispensable, para permitir al interesado continuar con la lectura del presente trabajo. Diremos entonces que gracias a *XMLSchema*, ahora sí es posible definir estructuras de datos independientemente del lenguaje de programación. Tenemos un sistema de tipos portátil, donde están incluidos los tipos simples esperados (números,fechas, boleanos, etc.), así como las estructuras arborescentes (de las cuales los registros y las listas no son más que casos particulares). Se puede inclusive, agregar algunas validaciones a la definición de los tipos, modularizando aún más su manejo. Veamos cómo podríamos migrar la definición de nuestro registro a *XSD*:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!--
    Definición de nuestra estructura de datos mtto_bitacora,
    usando XMLSchema
-->

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
  <xsd:complexType name="mtto_bitacora">
    <xsd:sequence>
      <xsd:element name="numero"        type="xsd:decimal"/>
      <xsd:element name="tipo"          type="xsd:decimal"/>
      <xsd:element name="fecha"         type="xsd:dateTime"/>
      <xsd:element name="duracion_hrs"  type="xsd:decimal"/>
      <xsd:element name="descp"         type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Nuevamente, procedamos a listar los detalles que juzgamos importante notar en el listado:

1. Estamos usando espacios de nombres *XML* (*XML-Namespaces*). Este es un mecanismo para tener distintos conjuntos de nombres para los elementos y atributos, en un mismo documento *XML*. Funciona asociando un prefijo (mediante un atributo del elemento donde deseamos usarlo) con un *URI* (*Uniform Resource Identifier*), que es una especialización de *URL* (*Uniform Resource Locator*), lo cual nos da dos cosas al mismo tiempo: un nombre único (a manera de nombre de variable) y una ubicación en la red para localizar algún documento relacionado (esto no es obligatorio, pero se recomienda). La principal razón de usar *URLs* como *URIs*, es porque las direcciones de páginas son fáciles de recordar, y proporcionan un mecanismo simple para garantizar unicidad en los nombres.

   Una vez que definimos el prefijo y lo asociamos con su identificador, se puede utilizar anteponiéndolo a los nombres de los elementos y atributos que estén relacionados, y con ello permitimos tener nombres repetidos, siempre y cuando pertenezcan a espacios diferentes de nombres (lo cual podemos saberlo viendo los nombres *calificados* o prefijados). Por ejemplo, en este documento estamos definiendo el prefijo *xsd*, para incluir a todos los elementos que pertenezcan al lenguaje *XML* que estamos presentando: *XMLSchema*; y si observamos todo el documento, veremos que únicamente estamos incluyendo elementos con este prefijo. Cuando no ponemos prefijo en la declaración del espacio de nombres, quiere decir que estamos bautizando con un identificador todos aquéllos elementos del documento que no estén prefijados (espacio de nombres predeterminado, podríamos llamarle).

   Podríamos hacer lo mismo con los atributos, es decir, prefijarlos; toda vez que también pertenecen al dialecto de *XMLSchema*. Sin embargo, en los ejemplos revisados hasta el momento, esto no parece una práctica común. Por otra parte, nótese que los valores de los atributos llamados *type*, también tienen un prefijo. Esto parece ser una convención, ya que el contenido de los atributos queda fuera de la jurisdicción del estándar *XML-Namespaces* (sin embargo, incluir los prefijos le da claridad al documento y evita confusiones).

2. Las equivalencias con el metalenguaje *XML*, en cuanto a la definición de nuestra estructura de datos, parece saltar a la vista de una forma considerablemente obvia. Observe que, por ejemplo, para indicar que deseamos definir un tipo registro, es necesaria la secuencia de elementos: *xsd:complexType* y *xsd:sequence*. Para los tipos simples (componentes del tipo registro), basta un elemento *xsd:element* con su atributo *type*. También se aprecia la correspondencia entre tipos *SQL* y tipos *XMLSchema*. Recordemos que en nuestras capas 1 y 2 acordamos (por recomendaciones de *SIUX*), usar únicamente tres tipos de datos simples: números de precisión arbitraria (*numeric*), que incluyen tanto enteros como cantidades con decimales; fechas (*date*) que incluyen marcas de tiempo (*timestamps*) y cadenas (*varchar* y *char*, aunque este último lo dejamos sólo para tablas, ya que en los procedimientos siempre ponemos *varchar*). Los equivalentes respectivos de *SIUX* para estos tipos *SQL-Oracle*, son los siguientes: *xsd:decimal*, *xsd:dateTime* y *xsd:string*.

4.4. XMLSchema y WebServices

Recapitulemos un poco para justificar la ruta que hemos tomado. Se supone que la intención de este capítulo es explicar qué son los *WebServices*, dado que fue la elección de nuestra propuesta para publicar la funcionalidad del *backend*. Para ello, primero introdujimos a los servidores de aplicaciones (*SA*), puesto que suelen ser la base para montar *WebServices*. Y luego, comenzamos a explayarnos un poco respecto a *XML* y a una de sus instancias: *XMLSchema*. La razón de esta última desviación, radica en que precisamente *XMLSchema* es el lenguaje empleado por *WebServices*, para indicar los tipos de los parámetos de entrada y salida de los servicios. Aclarado el punto, continuemos con otro ejemplo de *XMLSchema*. Supongamos que estamos interesados en representar con un tipo a los parámetros de salida del procedimiento de búsqueda *mtto.bitacora_con* (que de hecho sólo es uno: el que representa el cursor abierto, con el resultado de la búsqueda). Podríamos proceder como sigue:

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<!--
    Definición de un tipo en XMLSchema, que represente los
    parámetros de salida del procedimiento mtto.bitacora_con
-->

<xsd:complexType name="out_mtto_bitacora_con">
  <xsd:sequence>
    <xsd:element name="out_cur">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="row">
```

```
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="numero" type="xsd:decimal"/>
                <xsd:element name="tipo"   type="xsd:decimal"/>
                <xsd:element name="fecha"  type="xsd:dateTime"/>
                <xsd:element name="duracion_hrs" type="xsd:decimal"/>
                <xsd:element name="descp"  type="xsd:string"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Sigamos la definición de este tipo, para ver los elementos nuevos (en comparación con el ejemplo anterior):

1. El primer par de elementos *xsd:complexType* y *xsd:sequence*, dice que vamos a definir un tipo estructurado; para ser más específicos: un registro llamado *out_mtto_bitacora_con.*

2. Los siguientes cuatro elementos (*xsd:element*, *xsd:complexType*, *xsd:sequence* y *xsd:element*); nos indican que el primer componente del tipo registro *out_mtto_bitacora_con*, es un elemento llamado *out_cur*. A su vez, este elemento tiene otro tipo estructurado, que consta de una lista de elementos llamados *row* (que representan los registros de la tabla). Los valores de los atributos *maxOccurs* y *minOccurs* del último elemento *xsd:element*, señalan que no hay restricción ni limitantes respecto al tamaño de dicha lista.

3. El siguiente par de elementos (*xsd:complexType* y *xsd:sequence*), nos dice que el tipo cada elemento *row* de la lista *out_cur*, tiene un tipo registro.

4. Los últimos cinco elementos *xsd:element* (en cuanto a nivel de anidamiento), nos listan los componentes de los registros *row*. Todos ellos tienen tipos simples.

Para terminar nuestra breve y modesta introducción a *XMLSchema*, veamos cómo luce una instancia del tipo que apenas definimos (que no es otra cosa que un registro cuyo único elemento es una lista de registros). Este tipo representa precisamente, la salida del procedimiento *mtto.bitacora_con.*

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<!-- Instancia del tipo out_mtto_bitacora_con -->

<out_mtto_bitacora_con>
  <out_cur>
    <row>
      <numero>1</numero>
      <tipo>1</tipo>
      <fecha>2003-12-20</tipo>
      <duracion_hrs>7.0</duracion_hrs>
      <descp>Surge el catalizador, iconización con AmyLee (hasta canta).</descp>
    </row>

    <row>
      <numero>7</numero>
      <tipo>3</tipo>
      <fecha>2004-05-15</tipo>
      <duracion_hrs>5.0</duracion_hrs>
      <descp>Expo. estigmas, retorno al hogar, sublimación (MonAnge).</descp>
    </row>

    <row>
      <numero>666</numero>
      <tipo>10</tipo>
      <fecha>2004-05-27</tipo>
      <duracion_hrs>4.0</duracion_hrs>
      <descp>Ver miserable Apocalipsis Now Redux, comienza el final.</descp>
    </row>
  </out_cur>
</out_mtto_bitacora_con>
```

4.5. Publicación estática de WebServices

Muy bien, ahora que ya estamos un poco familiarizados con *XML* y *XMLSchema*, regresemos el centro de atención al estándar *WebServices*. Éste define una forma de publicar funcionalidad hacia el exterior, de manera transparente para las plataformas involucradas, dado que utiliza como medio de expresión para los datos *XML*. Entonces, un *WebService* tiene dos elementos fundamentales: tipos *XMLSchema* que definen la entrada y salida, una dirección en la red para invocarlo (*URL*) a través de algún protocolo también estándar de internet, *HTTP* por ejemplo (acrónimo de *Hypertext Transfer Protocol*).

Existe toda una fauna de protocolos y estándares asociados con *WebServices*, que es una tecnología emergente. Sin embargo, a pesar de que está en continua evolución (por lo menos en esta época), ya existe una parte del estándar que es suficientemente estable como para poder usarla (y desde luego, las implementaciones tanto comerciales como libres, respaldan tal afirmación). Entre

estos estándares asociados, probablemente los que merezcan mayor atención para nuestros fines sean sólo dos: *WSDL* y *SOAP*.

Comencemos con *WSDL* (*Web Services Description Language*). Se trata de un lenguaje instancia de *XML*, que sirve para especificar los servicios. Aquí es donde incluimos la *firma* de nuestros pedazos de funcionalidad que deseamos exportar, es decir, decimos qué tipo de datos reciben y qué tipo de datos arrojan. Aquí también es donde indicamos la dirección en internet donde podrán invocar al servicio (un *URL*, que también es *URI*). Y también en este lenguaje, es donde se indica la manera de codificar los mensajes para invocar *WebServices*; sabemos qué será el contenido de las cartas, pero necesitamos unos sobres para depositarlas. Dichos sobres también serán *XML* (sí, otros lenguajes instancia); el más famoso hasta el momento, podría ser SOAP (*Simple Object Access Protocol*).

Pero no extenderemos más los comentarios respecto a estos dos estándares, porque para nuestra propuesta no jugaron un papel tan importante (sí jugaron, pero no hubo necesidad de mucha intervención por parte de la infraestructura). Para explicar las razones de esto, veamos cuál es la idea detrás de la publicación de un *WebService*. Supongamos por ejemplo, que *SIUX* sólo nos ofrece las capas 1 y 2, y nos deja morir solos para las restantes. Entonces todo lo que tendríamos es un paquete en Oracle, es decir, un conjunto de procedimientos que implementan la lógica de nuestra aplicación (mantenimiento de la tabla *mtto_bitacora*). También supongamos que nos decidimos por *WebServices* y que ya tenemos nuestro producto instalado. Dicho producto asume que implementamos nuestra funcionalidad en Java, lo cual no es ningún problema (por ejemplo, Oracle provee la generación de las clases necesarias para invocar los procedimientos de nuestro paquete).

Entonces, emocionados por estar a punto de publicar nuestro primer *WebService (WS)*, nos ponemos a leer la documentación. El siguiente paso, según el manual, es crear la especificación del *WS* en el lenguaje *WSDL*. Tampoco es probable que este paso nos cause problemas; independientemente de que sea sencillo o no codificarlo manualmente, los productos suelen ofrecer generadores de dicha especificación, usando como entrada algún lenguaje de configuración más simple, o bien, directamente el código que pretendemos publicar. Seguimos leyendo y nos enfrentamos a una nada graciosa situación: los tipos *XMLSchema* incluidos en el archivo *WSDL*, y que indican la firma de nuestros procedimientos, serán usados para generar clases auxiliares que los representen (en Java, siguiendo con el hipotético producto).

¿Y para qué se generan esas clases auxiliares?, ¿a qué o quién auxiliarán? Pues sucede que "nos ayudarán" a los desarrolladores, ofreciéndonos una representación nativa del lenguaje, para las estructuras de datos que modelan la firma de nuestros prodecimientos. Pero, ¿quién le dijo al producto, que esas clases generadas por él nos servirían para nuestro código previamente

implementado? Es poco probable que haya coincidencia, por lo que tendríamos que realizar una conversión entre las clases que nos generó el producto a partir de nuestro *XMLSchema*, y las clases o tipos simples que esperan nuestros procedimientos. Aun cuando nos encontráramos en el remoto caso de que pudiéramos ahorrarnos esa conversión (tal vez pudimos forzar al producto a que usara nuestras clases), nos veríamos obligados a codificar un invocador, cuya única meta sería llamar a nuestro procedimiento (que suponemos ya tiene piel de Java, a estas alturas).

Codificar una clase invocadora por cada *WebService* que quisiéramos publicar, ¡qué pereza!, nada más desesperante que el código *paja* creado manualmente. Pero aquí no termina el martirio, lo más probable es que se nos pida codificar por lo menos otra clase (una para la parte del servidor y otra para el cliente). Dada la línea que pretendemos mantener con nuestra propuesta, este esquema es poco tolerable. Así que optamos por una solución, donde no es necesario codificar manualmente nada para cada *WebService*, pero que sin embargo no publica las *firmas* de las funciones como lo propone el estándar (unas cosas, a cambio de otras). Obviamente, la generación de código Java que represente a nuestro paquete de la capa 2 tampoco es necesaria (sólo lo estábamos mencionando para darle contexto a la explicación).

4.6. Publicación dinámica de WebServices con SIUX

La idea con *WebServices* es que cualquiera que los desee invocar pueda hacerlo únicamente con la información que contenga su definición *WSDL* (de hecho, los clientes antes de la invocación seguramente pedirán esa información). *SIUX* también ofrece publicación de las firmas de los procedimientos, pero no dentro de la definición *WSDL* del servicio, ya que dentro de nuestra infraestructura, para resolver el problema de tener un invocador dinámico de *WS* (y evitar la codificación particular para cada uno), se creó un sólo *WebService*. A este servicio le llegan datos *XML* que cumplen parcialmente con el *XMLSchema* de algún procedimiento, cuyo nombre está incluido en la petición (*URL*). La parcialidad radica en que sólo se verifica la existencia de ciertos elementos, y no el orden/profundidad de aparición o sus tipos. Esa información, le basta al procedimiento *PSM* para realizar la invocación, usando al manejador *JDBC* del *SMBD* (Oracle). Cabe mencionar que antes de invocar al procedimiento se convierten los parámetros de entrada a los tipos que indica su firma (realizando con ello, otra parte de la validación de los datos de entrada). También con ese mismo manejador de la base de datos, se puede extraer la metainformación de la respuesta del procedimiento que no estuviera incluida en los parámetros de entrada (el resultado de una búsqueda, por ejemplo).

De esta forma, hacia la implementación de *WebServices*, nosotros ofrecemos un solo servicio genérico: recibe prácticamente cualquier dato *XML*, y arroja otro igualmente genérico como

resultado. Es labor del servicio interpretar las entradas para decidir en tiempo de ejecución a qué procedimiento almacenado desean invocar. Este *WebService* genérico, llamado *Psm2XML*, se convierte así en el intermediario que se encarga de ponerles piel *XML* a nuestros viejos procedimientos *PSM*. Son los mismos registros y listas de registros, pero con traje nuevo. Así nos evitamos el código intermedio, que no haría otra cosa más que estorbar. Muy bien, no seamos tan injustos: un efecto positivo de esas clases auxiliares que proponen los productos, es que el esquema *XML* sea verificado, y de momento nosotros no hacemos tal validación en la versión dinámica de nuestro *Web Service*. Sin embargo, ello no se debe a que no podamos, simplemente fue por cuestiones de tiempo -los datos que envían las interfaces de usuario tienen una ligera diferencia con los esquemas de los procedimientos que se generan también automáticamente; así que habría que hacer una homogenización para habilitar la validación de los esquemas.

La implementación de *WebServices* que fue seleccionada se llama *Axis* y es parte de la enorme infraestructura de software libre que ofrece la *ASF* (*Apache Software Foundation*). Está hecho en Java, y ello dictó el lenguaje para implementar ese *WebService* genérico. Afortunadamente, fue relativamente sencilla esta implementación (una vez que visualizamos la opción dinámica de invocación). Pero, ¿no dijimos que todo *WS* debe tener una definición en *WSDL*? Es verdad, y Axis la genera por nosotros. Nuestra única responsabilidad es implementar el invocador e instalarlo con unas cuantas instrucciones de configuración.

Ya se mencionó también, al principio de esta sección, que las implementaciones de *WebServices* suelen montarse sobre un servidor de aplicaciones, y que incluso a la fusión resultante se le puede clasificar nuevamente como un *SA*. En el caso de nuestra propuesta, Axis cae dentro de este grupo y experimentamos montándolo sobre *Tomcat*, otro proyecto de la *ASF*. Tomcat es un servidor de aplicaciones que implementa parte del estándar *J2EE* (*Java2 Enterprise Edition*). Bajo su jerga, Tomcat es un *Contenedor de Servlets y JSPs* (entre otras cosas). Hablaremos más sobre esta tecnología en el siguiente capítulo (capa 4); por ahora simplementa apuntaremos que estos estándares (más relacionados con el lenguaje Java, que con *XML*), son utilizados para desarrollar aplicaciones para la Web (precisamente, el sabor que deseamos para nuestras interfaces de usuario). El acoplamiento entre *Axis* y *Tomcat* es bastante natural, situación que no debe extrañar dado que tiene el mismo origen (*ASF*).

Es común que las capas superiores usen directamente algún servicio de las inferiores, y aquí no es la excepción. A pesar de que la invocación de los procedimientos corre a cargo del estándar *JDBC*, en esta capa tiene lugar la validación de los datos con tipos simples, que nos están pasando como entrada; esto es, en el dato *XML* que recibimos, sólo vemos los parámetros de entrada como meras cadenas, pero usando las funciones que mencionamos al final del capítulo anterior (*str2num*, *str2date* y *str2str*, del paquete *siux2_par*), procedemos a validarlas. En terminología de compiladores, diríamos que aquí es donde *analizamos sintácticamente* los datos de entrada, pero

propagamos la excepción adecuada, en caso de ser necesario (es agradable ver cómo las excepciones de *SQL-PSM* se transforman en excepciones de Java). La información de esta excepción, es la que a su vez viaja hacia la interfaz del usuario (proporcionando el mensaje y código correspondiente). Una alternativa para esta validación sería usar el *XMLSchema* de nuestros procedimientos; sin embargo, las validaciones que hace el manejador de la BD nos parecen más robustas en este sentido, y por ello lo delegamos a la capa 2.

A pesar de esta mecánica de validación, sí ofrecemos (adicionalmente al invocador genérico de *PSM* como *WebServices*) un servicio que informe sobre la firma de nuestros procedimientos (codificada en *XMLSchema*). Se trata de otra clase en Java (*Psm2Xsd*), que dado el nombre de un procedimiento o paquete *PSM-Oracle*, usa la metainformación que guarda el SMBD junto con los catálogos de las capas 1 y 2 de *SIUX*, para construir la especificación de los tipos. Aunque esta funcionalidad no es ofrecida como un *WebService* (no tendría mucho sentido), es igualmente fácil de acceder: basta cualquier cliente *HTTP* (*Hypertext Transfer Protocol*), el protocolo a través del cual viaja la mayoría de los datos en Internet (principalmente *HTML*, así que ¿por qué no también *XML*?).

La publicación de esta funcionalidad corre a cargo del *SA Tomcat*, y se hace a través de un *Servlet* (veremos un poco más de ellos en la siguiente sección, pero de momento quedémonos con la idea, de que son generadores de contenido dinámico). Entonces, en la configuración de *Tomcat* asociamos un *URL* con nuestro *Servlet* (que a su vez usa los servicios de la clase *Psm2Xsd*), y cualquier cliente *HTTP* que lo pida, obtendrá como resultado la firma en *XMLSchema* del procedimiento solicitado. Habría ligeras modificaciones a estos *XMLSchemas* generados automáticamente, con respecto a los ejemplos mostrados en párrafos previos: la razón es que omitimos cierta información, para evitar que distrajera innecesariamente. Pero veamos cómo lucen en realidad: este sería el esquema de los procedimientos para nuestra aplicación de mantenimiento (nuevamente estamos omitiendo los métodos para bajas y modificaciones, por considerar que tienen el mismo patrón que el alta). Estamos omitiendo las partes que pueden ser inferidas, para no atiborrar de código *XML* las páginas de este documento (y ésta será nuestra política con todos los listados que sean demasiado extensos):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:siux="http://www.chacaweb.org/siux"
>
  <xsd:complexType name="bitacora_add">
    <xsd:sequence>
      <xsd:element
        siux:descp="Tipo de actividad"
```

```
        siux:label="Tipo"
        siux:len="12"
        name="in_tipo"
        type="xsd:decimal"
      />
      ...
      <xsd:element
        siux:descp="Descripción de actividad"
        siux:label="Descripción"
        siux:len="2048"
        name="in_descp"
        type="xsd:string"
      />
      <xsd:element
        siux:descp="Código de retorno"
        siux:label="Codret"
        siux:len="12"
        name="out_errcode"
        type="xsd:decimal"
      />
      <xsd:element
        siux:descp="Descripción Codret"
        siux:label="Descripción Codret"
        siux:len="2048"
        name="out_descp_errcode"
        type="xsd:string"
      />
      <xsd:element
        siux:descp="Número de actividad"
        siux:label="Número"
        siux:len="12"
        name="out_numero"
        type="xsd:decimal"
      />
      ...
      <xsd:element
        siux:descp="Descripción de actividad"
        siux:label="Descripción"
        siux:len="2048"
        name="out_descp"
        type="xsd:string"
      />
    </xsd:sequence>
  </xsd:complexType>
...
  <xsd:complexType name="bitacora_con">
    <xsd:sequence>
      <xsd:element
        siux:descp="Número de actividad (min)"
        siux:label="Número (min)"
        siux:len="12"
        name="in_numero_min"
        type="xsd:decimal"
      />
      <xsd:element
        siux:descp="Número de actividad (max)"
        siux:label="Número (max)"
        siux:len="12"
        name="in_numero_max"
```

```
  type="xsd:decimal"
/>
...
<xsd:element
  siux:descp="Fecha de realización de actividad (min)"
  siux:label="Fecha (min)"
  siux:len="7"
  name="in_fecha_min"
  type="xsd:dateTime"
/>
<xsd:element
  siux:descp="Fecha de realización de actividad (max)"
  siux:label="Fecha (max)"
  siux:len="7"
  name="in_fecha_max"
  type="xsd:dateTime"
/>
...
<xsd:element
  siux:descp="Descripción de actividad"
  siux:label="Descripción"
  siux:len="2048"
  name="in_descp"
  type="xsd:string"
/>
<xsd:element
  siux:descp="Código de retorno"
  siux:label="Codret"
  siux:len="12"
  name="out_errcode"
  type="xsd:decimal"
/>
<xsd:element
  siux:descp="Descripción Codret"
  siux:label="Descripción Codret"
  siux:len="2048"
  name="out_descp_errcode"
  type="xsd:string"
/>
<xsd:element siux:cur="yes" name="out_cur">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="row">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element
              siux:descp="Número de actividad"
              siux:label="Número"
              siux:len="12"
              name="out_cur_numero"
              type="xsd:decimal"
            />
            ...
            <xsd:element
              siux:descp="Descripción de actividad"
              siux:label="Descripción"
              siux:len="2048"
              name="out_cur_descp"
              type="xsd:string"
```

```
                />
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
</xsd:schema>
```

Entre los detalles a notar tenemos: el procedimiento de alta no tiene al número de actividad como entrada, porque se detectó una secuencia que lo calcula (*mtto_bitacora_seq_numero*); se incluyen, como parte de sus salidas, todas las columnas de la tabla (puesto que el procedimiento generado así las tiene). Las declaraciones de los parámetros de entrada y salida fueron aumentadas con atributos que contienen información de los catálogos de *SIUX* (*siux:len*,*siux:label* y *siux:descp*); los parámetros de salida tipo referencia de cursor (*out_cur*) tienen el atributo *siux:cur* (todos estos atributos tienen su propio espacio de nombres: *siux*). Se están agregando explícitamente el código de error y su descripción a la lista de parámetros de salida. La firma para el procedimiento de consulta (*bitacora_con*) muestra que la versión generada por *SIUX* maneja más parámetros de entrada (todas las columnas de la tabla), que los mostrados en nuestro ejemplo manual.

Pero, es prudente preguntarnos si los clientes del *WebService* genérico no decidieran usar este *XMLSchema* (las interfaces *XForms* que probamos no lo hicieron), puesto que nada los obliga a lo contrario (mientras manden los datos obligatorios) ... ¿entonces, para qué sirven? La respuesta a esta pregunta abre la puerta para que se pase al siguiente capítulo: se usan para generar automáticamente las interfaces de usuario. Dentro de poco presentaremos un pequeño lenguaje (sí, otro lenguaje hijo de *XML*), que se diseñó para minimizar la cantidad de información a indicar en la construcción de una interfaz de usuario. También, como parte esencial de la infraestructura en la capa 4, tenemos un transformador que toma como entrada una especificación en ese lenguaje (*SFX*, por *SIUX Forms*) más el *XMLSchema* de nuestros procedimientos (generado automáticamente por la capa 3). El resultado: una interfaz codificada en *XForms*.

4.7. XMLSchema para las tablas de SIUX

Ahora veremos otra aplicación del estándar *XMLSchema*, aunque de una naturaleza un tanto distinta a las mostradas hasta ahora. En lugar de usarlo para especificar algún tipo que modele datos del *SI* particular que estamos construyendo, lo emplearemos para especificar en un solo bloque de información todo lo relacionado con las tablas bajo el control de *SIUX*. Recordemos,

que cuando en el capítulo dedicado a la capa uno presentamos la manera de introducir la metainformación relacionada con la tabla, pedíamos que se invocaran a rutinas de *SIUX* en cierto orden.

Aunque para ciertas personas podría funcionar bien este esquema de llamar a rutinas para alimentar los catálogos de *SIUX*, podríamos estar dependiendo mucho de su disciplina o de sus costumbres, dado que no existe ningún impedimento para que no invoquen a las rutinas de la infraestructura en los lugares precisos alrededor de la creación de la tabla. Para solventar esta situación, y aprovechando la uniformidad que existe en la sequencia de instrucciones *SQL* involucradas, se creó un *XMLSchema* donde tanto la definición de la tabla, como los parámetros para las rutinas de la *SIUX* están incluídos; obligando así, por lo menos, a que sea perceptible la información relacionada con la estructura y restricciones de la tabla - lo cual eventualmente podría propiciar que se actualicen de manera sincronizada todos los elementos asociados con la misma-. Cabe mencionar que las dos opciones se dejan disponibles, para que el programador o implementador decida cuál se acopla mejor con sus necesidades y estilo.

Pospusimos hasta este momento la presentación de esta característica opcional de *SIUX*, porque no tenía mucho caso hablar de ella sin la introducción del estándar *XMLSchema*. Mostremos pues, sin más preámbulo, el *XMLSchema* empleado para crear el tipo de datos que representará a la información relacionada con una tabla bajo *SIUX*.

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="siux_table">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="info">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="app" type="xsd:string"/>
              <xsd:element name="name" type="xsd:string"/>
              <xsd:element name="descp" type="xsd:string"/>
              <xsd:element name="lang" type="xsd:string"/>
              <xsd:element name="audit">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="yes"/>
                    <xsd:enumeration value="no"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>

        <xsd:element name="columns">
```

```
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="col">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="type">
              <xsd:complexType>
                <xsd:simpleContent>
                  <xsd:extension base="xsd:string">
                    <xsd:attribute
                       name="prec"
                       type="xsd:integer"
                       use="optional"
                    />
                    <xsd:attribute
                      name="scale"
                      type="xsd:integer"
                      use="optional"
                    />
                    <xsd:attribute
                      name="len"
                      type="xsd:integer"
                      use="optional"
                    />
                  </xsd:extension>
                </xsd:simpleContent>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="null">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="no"/>
                  <xsd:enumeration value="yes"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="label" type="xsd:string"/>
            <xsd:element name="descp" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="constraints"  maxOccurs="1" minOccurs="0">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="primary_key"  maxOccurs="1" minOccurs="0">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="cols" type="xsd:string"/>
            <xsd:element name="descp" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

```
            <xsd:element name="foreign_key" maxOccurs="1" minOccurs="0">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="name" type="xsd:string"/>
                  <xsd:element name="src_cols" type="xsd:string"/>
                  <xsd:element name="dst_cols">
                    <xsd:complexType>
                      <xsd:simpleContent>
                        <xsd:extension base="xsd:string">
                          <xsd:attribute
                            name="table"
                            type="xsd:string"
                            use="required"
                          />
                        </xsd:extension>
                      </xsd:simpleContent>
                    </xsd:complexType>
                  </xsd:element>
                  <xsd:element name="descp" type="xsd:string"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>

            <xsd:element name="check" maxOccurs="unbounded" minOccurs="0">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="name"  type="xsd:string"/>
                  <xsd:element name="expr"  type="xsd:string"/>
                  <xsd:element name="descp" type="xsd:string"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>

            <xsd:element name="proc" maxOccurs="unbounded" minOccurs="0">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="name"  type="xsd:string"/>
                  <xsd:element name="descp" type="xsd:string"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Demos ahora un último repaso del estándar *XMLSchema*, con ayuda de este ejemplo que forma parte de la infraestructura *SIUX*. Puede apreciarse que se trata de un ejemplo un poco más elaborado que los anteriores, dado el nivel de anidamiento mostrado en las etiquetas. Sin embargo, no pierda de vista el lector que el nivel de anidamiento de etiquetas en los dialectos *XML*, no necesariamente implica que la estructura definida contendrá esa jerarquía (sintácticamente hablando).

Puede ser el caso, como lo es aquí, que el anidamiento se traduzca en acumulación de propiedades, en una especialización de los elementos que conforman el tipo de datos.

El *XMLSchema* mostrado define al elemento "siux_table", el cual se muestra como un tipo complejo. De todos los tipos complejos que soporta el estándar, aquí escogimos una secuencia; que está conformada por los siguientes elementos: *info*, *columns* y *constraints*. Cada uno de estos componentes, representa respectivamente: la información relativa a la tabla, la definición de las columnas y las restricciones impuestas a la tabla. A su vez, estos elementos cuentan con su propio tipo (que también es estructurado).

El elemento *info* es, nuevamente, un tipo complejo secuencia, donde incluímos los parámetros de las rutinas que necesitamos invocar alrededor de la creación de la tabla: la aplicación a la que pertenece la tabla, su nombre, una descripción, lenguaje a usar y una bandera para activar la auditoría en sus registros. Con respecto a esta última opción (*audit*), vemos algo nuevo: estamos derivando un nuevo tipo en base a uno existente; del tipo para cadenas (*xsd:string*) estamos diciendo que sólo queremos aquéllos valores listados en la enumeración: *yes*, *no*.

Después viene la sección de columnas, que consiste en una lista de registros (bueno, de elementos *col*); nuevamente observamos la presencia de los atributos *maxOccurs* y *minOccurs* para delimitar el número de ocurrencias del elemento (en este caso, abrimos ambos parámetros para tener listas de tamaño arbitrario). Después, dentro del elemento *col* vemos nuevamente estructura. Se trata de una secuencia de elementos con tipos básicos, que incluyen en ese orden: nombre, tipo, nulidad, etiqueta y descripción de la columna. Los primeros tres sirven para formar la instrucción *create table*, mientras que los dos últimos se usan para la llamada a la rutina que agrega la información de la columna a los catálogos de *SIUX*. Lo nuevo en esta parte, es la definición del tipo para el campo *type* de la estructura; nuevamente se trata de una derivación (especialización del tipo existente *xsd:string*), pero agregando también atributos (*prec*, *scale* y *len*) que se emplean para indicar la precisión y longitud de las columna de tipo numérico y cadena.

Por último, tenemos la sección de restricciones, donde observamos una secuencia de los diferentes tipos de ofrece *SQL*. Las llaves primarias sólo pueden presentarse una o cero veces, mientras que el resto (foráneas, condiciones estáticas y procedimientos de validación) pueden aparecer un número arbitrario de veces. Veamos un ejemplo de cómo luciría la especificación de una tabla para *SIUX* bajo esta nueva herramienta. En lugar de una sequencia de instrucciones *SQL*, tendríamos un listado como el siguiente (que "casualmente" es el ejemplo de mantenimiento de tabla que venimos acarreando):

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<siux_table
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="siux_table.xsd"
>
  <info>
    <app>mtto</app>
    <name>bitacora</name>
    <descp>Bitácora de actividades</descp>
    <lang>es</lang>
    <audit>yes</audit>
  </info>

  <columns>
    <col>
      <name>numero</name>
      <type prec="12">numeric</type>
      <null>no</null>
      <label>Número</label>
      <descp>Número de actividad</descp>
    </col>

    <col>
      <name>tipo</name>
      <type prec="12">numeric</type>
      <null>no</null>
      <label>Tipo</label>
      <descp>Tipo de actividad</descp>
    </col>

    <col>
      <name>fecha</name>
      <type>date</type>
      <null>no</null>
      <label>Fecha</label>
      <descp>Fecha de realización de actividad</descp>
    </col>

    <col>
      <name>duracion_hrs</name>
      <type prec="3" scale="2">numeric</type>
      <null>no</null>
      <label>Duración(hrs)</label>
      <descp>Duración en horas de actividad</descp>
    </col>

    <col>
      <name>descp</name>
      <type len="2048">varchar</type>
      <null>no</null>
      <label>Descripción</label>
      <descp>Descripción de actividad</descp>
    </col>
  </columns>

  <constraints>
    <primary_key>
      <name>cpk</name>
```

```
      <cols>numero</cols>
      <descp>Ya existe una actividad con ese número</descp>
    </primary_key>

    <foreign_key>
      <name>cpk</name>
      <src_cols>tipo</src_cols>
      <dst_cols table="tipo_actividad">numero</dst_cols>
      <descp>El tipo de actividad ingresado, no existe en catálogo</descp>
    </foreign_key>

    <check>
      <name>cch_a001</name>
      <expr>numero > 0 and duracion_hrs > 0</expr>
      <descp>Valor inválido para el número o la duración de actividad</descp>
    </check>

    <check>
      <name>cch_a002</name>
      <expr>extract (year from fecha) > 2000 and fecha &lt;= '2006-12-08'</expr>
      <descp>La fecha de la actividad no pertenece al rango  (2001-01-01,2006-12-08]</descp>
    </check>

    <check>
      <name>cch_a002</name>
      <expr>
        (to_char (fecha,'D') between 2 and 6) and -- 1 = domingo
         to_char (fecha,'W') &lt;> 2              and
         (last_day (fecha) - fecha) > 5
      </expr>
      <descp>Fecha inválida: sólo tareas de L-V, descontando segunda semana del mes y los últimos 5 días del mismo.</descp>
    </check>

    <proc>
      <name>cch_a004</name>
      <descp>Nuestro sistema marca las [%1], lo cual lamentablemente, se encuentra fuera de horario.</descp>
    </proc>
  </constraints>
</siux_table>
```

Un efecto lateral, no muy agradable, de usar *XML* para la especificación de nuestras condiciones; consiste en las precauciones necesarias al emplear signos que pertenezcan a dicho lenguaje. Definitivamente, los candidatos a dar más problemas son los signos de mayor y menor que. Nótese que para solventar esta situación, tenemos que hacer uso de las referencias a dichos símbolos en lugar de escribirlos literalmente (en nuestro caso, para el símbolo de menor que). Lo único que resta indicar, y que fue dejado aparte intencionalmente, es la definición de las validaciones que implican la codificación de un procedimiento (como el *cch_a004* de este ejemplo). Se dejaron fuera del *XMLSchema* por incluir expresiones arbitrarias *SQL*, lo que complicaría considerablemente la creación de un tipo de datos que las incluyera (además que podría ser poco práctico en *XML*, se podría volver muy cargado, sintácticamente hablando).

Por último, mencionaremos que se hizo un programa transformador de esta estructura de datos a *SQL*. Es decir, que sólo agregamos una capa adicional entre el usuario y la capa 1 de *SIUX*. El lenguaje ideal para este tipo de conversiones estructurales es *XSLT*, del cual hablaremos un poco más en el siguiente capítulo. Por ahora, sólo diremos que si bien el tipo de datos definido con ayuda del estándar *XMLSchema* ya nos ahorra gran parte de las validaciones, quedan unas cuantas que no fueron expresadas y que cuya aplicación es tarea de este programa conversor. Por ejemplo, el que los atributos del campo *type* de las columnas, estén informados de acuerdo al tipo ... una cadena no debería tener precisión, ni una fecha longitud, por mencionar algunos casos.

Capítulo 5

Capa 4: Interfaz hacia el usuario

5.1. El papel de las interfaces de usuario

A estas alturas contamos con un *backend* homogéneo, donde toda la funcionalidad es publicada a través de *WebServices*. Ocultamos hacia el resto del mundo qué plataforma estamos usando (sistema operativo, SBMD, lenguajes de programación, etc.) y nuestros invocadores seguramente lo agradecerán. Si consideramos que nuestros usuarios del *backend* podrían ser tanto humanos como otros programas, podemos apreciar que sólo los primeros implican una capa adicional: mientras que los *WebServices* pueden ser invocados directamente por cualquier implementación que respete el estándar, los usuarios humanos necesitan de un intermediario gráfico para usar la funcionalidad ofrecida. De crucial importancia es que ocultemos los detalles tecnológicos, para que alguien armado únicamente con las reglas del negocio, pueda usar el sistema sin preocuparse por saber qué es *SQL* o *XML*. Sobre esta capa ahondaremos en el presente capítulo.

Desde la introducción del presente trabajo, mencionamos que *XForms* sería el estándar propuesto para implementar las interfaces del usuario (*GUI* por las siglas en inglés de *Graphical User Interface*); sin embargo, probablemente las ganancias de esta elección no puedan ser apreciadas en su plenitud, a menos que demos primero un pequeño vistazo a las formas tradicionales de programación para este nivel. En especial, las aplicaciones para internet siempre han dado dolores de cabeza en esta parte, debido a la naturaleza misma de la tecnología involucrada.

Pero antes de comenzar un recorrido tecnológico, aclaremos de una vez el punto básico de las interfaces de usuario en nuestra propuesta: su responsabilidad será únicamente de intermediario entre el *backend* y el humano. Sólo se remitirán a ofrecer al usuario una forma de pedir servicios, y de presentarle los resultados de una manera legible; eso es todo. No permitimos que las interfaces de usuario tengan alambrada ciertas partes de las reglas del negocio, ya que ello violaría claramente las reglas del juego entre las capas de la arquitectura. Entonces, cuando hablemos de *lógica* en este

nivel, de ninguna manera nos refereimos a las reglas del negocio, sino únicamente a la lógica del control de las pantallas. Podríamos metaforizar diciendo que las interfaces de usuario son ciegas en cuando a la semántica de los datos; ellas simplemente los hacen accesibles y será tarea del usuario interpretarlos.

5.2. Tecnología de ventanas y MVC

Bien, ahora comencemos con las interfaces clásicas: las que usan ventanas. Éstas son comúnmente programadas en lenguajes como Visual Basic o Java, y actualmente contamos con software que permite programar algunas partes del aspecto de las pantallas, de manera visual (parecido al software de diseño gráfico). Estos programas generan el código necesario para implementar la "vista" de la pantalla y permiten al programador incluir manualmente la lógica de control. Independientemente de que usemos un generador de código para aligerar la carga de trabajo, es prácticamente un hecho que el control de la interfaz lo tengamos que programar en algún lenguaje con sabor imperativo (al menos, en el mundo comercial).

Sin embargo, también podríamos decir que esa programación imperativa es una especialización, dado que está orientada a eventos. Los eventos se generan cuando el usuario interactúa con los controles de la pantalla y son activadas secciones de código que asociamos con los mismos. Una vez activada la sección correspondiente, la misma produce algún cambio en el comportamiento de la interfaz (llama algún servicio o cambia la apariencia del algún control) y, generalmente, dicho cambio se hace perceptible al usuario.

Por llevar un buen rato entre nosotros, la programación de interfaces de usuario se ha visto beneficiada por las innovaciones y adelantos de otras áreas de la Computación. Particularmente útil le resultó la perspectiva de la *Programación Orientada a Aspectos* (*AOP*, por las siglas en inglés). En ésta, se reconoce que los sistemas están compuestos por varias "lógicas" que interactúan entre sí y persiguen propósitos independientes, aunque relacionados; entonces, se trata de ofrecer lenguajes que reflejen esa relativa independencia y permitan implementar cada aspecto por separado. El manejo de excepciones, que ya mencionamos en capítulos anteriores, podría ser un claro ejemplo de un "aspecto" que los lenguajes modernos aislan. En realidad, este paradigma de *AOP* no se usa de manera independiente, sino que se integra a los paradigmas existentes (imperativo u orientado a objetos, por mencionar los más famosos para *SI*).

Aplicando el paradigma *AOP* al diseño y programación de interfaces de usuario, obtenemos tres aspectos básicos de las mismas: los datos que van y vienen del *backend* (modelo), el control (la lógica de control mencionada) y la vista (la presentación de los datos). Si colectamos los

comentarios entre paréntesis, tendremos las siglas del acrónimo en inglés (*MVC*) que fue asignado a este modelo; el paquete "Swing" del lenguaje Java es probablemente su encarnación más reciente y conocida. Si no usamos este patrón de modelo-vista-controlador, tendríamos que codificar cada uno de esos tres aspectos de manera intercalada (lo cual suele complicar el desarrollo).

No debe confundirse esta modularización a nivel de aspectos en la capa 4, con la separación en capas de la arquitectura en general, aunque la existencia de la segunda puede beneficiar a la primera. Cuando hablamos de que *MVC* propone separar los "datos", nos referimos a la comunicación con el *backend*, a la petición del servicio y no a la propia modularización del modelo de datos y reglas del negocio. Así está el panorama actualmente para el diseño y la programación de interfaces de usuario, con tecnología de ventanas. No estamos diciendo que ya sea una pauta genérica el uso del patrón *MVC*; sólo que éste ya se encuentra accesible a los desarrolladores. Seguramente ya están circulando por el mercado herramientas *RAD* que auxilian en la generación de código automático (siguiendo la pauta del patrón).

Una situación más complicada ocurre con el desarrollo de aplicaciones con interfaz *web*, dado que aquí la tecnología no caza de manera natural con la idea que tenemos de interfaz de usuario. Un modelo, al parecer ya clásico, es el de autómata de estados finito, y precisamente el *estado* no es algo que esté pensado manejar en la *Web*. Entonces, aunados a los problemas comunes del desarrollo de una interfaz de usuario, si seleccionamos el internet como medio de expresión tenemos que agregar otros más. De hecho, no es raro que los desarrolladores de internet simplemente se conformen con lograr una funcionalidad equivalente a la tecnología de ventanas; lo cual ya no les deja mucho tiempo, ya no digamos para reflexionar si están usando un patrón *MVC*, sino para revisar si usan una arquitectura por capas.

5.3. Evolución histórica de la tecnología Web

Pero, ¿por qué es más complicado desarrollar interfaces de usuario para la *Web*? Para explicarlo, revisemos primero algunos aspectos básicos de internet en general. El componente central de esa enorme red de computadoras que llamamos internet, y que solemos asociar a menudo con la *Web* (que sería sólo una parte), son los protocolos. Dichos protocolos permiten que las computadoras y sus programas puedan "platicar" independientemente de la plataforma de *hardware* o *software* que tengan instalada. Los protocolos también están modularizados, de manera que capas superiores piden "servicios" a las inferiores. De este diseño por capas surge el apelativo de "pila" que asignamos a la base de los protocolos para internet: *TCP/IP*.

Encima de los protocolos genéricos *TCP/IP*, tenemos a los de aplicación, y entre éstos, en-

contramos al famoso *HTTP* (*Hypertext Transfer Protocol*). ¿A qué debe su fama? Resulta que este protocolo es nada menos que la base de lo que llamamos *la Web*. En un principio, sólo fue concebido pensando en un mecanismo a través del cual las computadoras pudieran intercambiar información tipo texto (documentos). Este protocolo, al igual que muchos otros, maneja un esquema de cliente servidor; entonces, sus implementaciones deben ofrecer la parte del servidor, que consiste en un software que radica en la computadora donde están los documentos a compartir. Por otro lado, las implementaciones de la parte del cliente deben ofrecer un mecanismo para pedir esos documentos al servidor y también para presentarlos al usuario de alguna manera amable (actualmente conocemos a este software cliente como *navegador*). El concepto de *amabilidad* por parte de los visualizadores cambia con la época; para las generaciones actuales que tienen acceso a una *Web* multimedia, seguramente daría risa ver las interfaces sólo texto de antaño, consideradas amistosas en aquellos días.

Tan casados estamos hoy en día con el aspecto visual, que prácticamente todos los documentos que se comparten en la *Web* yacen codificados en un lenguaje para formateo de datos llamado *HTML* (*Hypertext Markup Language*); de hecho, este lenguaje es el hijo pródigo de *SGML* (comentado en el capítulo anterior). En *HTML* no sólo podemos incluir texto, sino que también podemos indicar sus atributos de apariencia (tipo de letra, color, tamaño, etc.), aparte de incluir referencias a objetos multimedia, de los cuales, sin duda, las imágenes son los más famosos (dado que la inclusión de videos o música se suele lograr con mecanismos externos al lenguaje *HTML*). Y claro, también podemos *bajar* archivos de internet usando *HTTP* - aunque ya existe otro protocolo más especializado para tal fin: *FTP* (*File Transfer Protocol*). La referencia a estos archivos que podemos bajar, o a otros documentos, se indica en *HTML* a través de una etiqueta que nombramos como *liga*.

Estas ligas son el mecanismos a través del cual podemos *navegar* en internet, saltando de un documento a otro. Las mismas contienen una dirección que le indica a nuestro cliente *Web* dónde se encuentra el siguiente documento a mostrar, y están codificadas usando el estándar *URL* (*Uniform Resoure Locator*). En su forma más simple, un *URL* indica tres cosas: el protocolo de aplicación que debemos usar para extraer el objeto referido (*HTTP* en su mayoría), la dirección *TCP/IP* donde radica (compuesta por una dirección *IP* y un puerto *TCP*) y el nombre del recurso referido (que asemejan una ruta de un sistema de archivos). Por ejemplo, el siguiente *URL* hipotético *http://www.freedom.org.mx:666/tesis/doc/siux.html*, se refiere a un documento accesible en la ruta */tesis/doc/siux.html*, en el puerto 666 del servidor *HTTP*, cuya dirección *IP* está asociada al nombre *www.freedom.org.mx* (las implementaciones de un protocolo adicional, *Domain Name Service* (*DNS*), *resolverían* el nombre y nos darían la *IP* asociada).

Entonces, mientras que los servidores *HTTP* sólo deben preocuparse por ese protocolo, lo que conocemos por *navegador* es un software que aparte de ser cliente *HTTP*, también debe entender

el lenguaje *HTML* para visualizar los documentos referidos por los *URLs*, una vez que cuenta con ellos (a los usuarios no les agradaría mucho ver el documento en *HTML* "crudo"). Si excluimos a las dichosas aplicaciones para internet, la *Web* se reduce a un conjunto de servidores *HTTP* que tienen acceso a repositorios de documentos y que los comparten con una horda de ansiosos clientes *HTTP* (a través de *URLs*). Uno de estos documentos, hecho en *HTML* y referido a través de un *URL*, es lo que llamamos comúnmente "una página de internet" (aunque claro, la pensamos en su versión gráfica, lograda gracias al *navegador*).

Hasta este punto no se aprecia ningún inconveniente grave: se puede compartir información sin problemas. Pero ¿qué pasa, si aparte deseamos que las "páginas" de internet nos ofrezcan cierta funcionalidad? Lo que sucede es que nos comenzará a dar dolor de cabeza. Dado que todo descansa sobre el protocolo *HTTP*, y éste a su vez no requiere "recordar" lo que hizo previamente el cliente (del protocolo), entonces no tenemos manera de manejar contexto. En otras terminologías, llamaríamos a este protocolo *transaccional* (sólo sirve para invocaciones aisladas, no para conversaciones). Y precisamente, contexto es lo que requieren las interfaces de usuario para funcionar correctamente (para saber qué comportamiento tomar en función de interacciones previas del usuario). Para resolver este problema, en el lado del servidor es donde, adicionalmente a la implementación *HTTP*, se han agregado otros componentes que permitan simular ese contexto (aparte de permitir el acceso al *backend*).

La naturaleza del protocolo *HTTP* también determina las reglas del juego para programar las interfaces de usuario. En primera, el mecanismo para permitir al usuario tener algo de interacción con las páginas es muy limitado. *HTML* fue concebido como un lenguaje para formatear documentos, y dada la necesidad de tener páginas con funcionalidad se le agregó un componente adicional: las formas. Podemos pensar a las formas *HTML* como un tipo especializado de interfaz de usuario, donde tenemos una serie de controles para insertar datos (texto libre, listas de selección, etc.) y unos controles para enviar una petición a cierta dirección (*URL*), donde nos estará esperando un programa para procesar la solicitud. Hay un estándar que define la manera en que los datos asociados a esos controles deben viajar a través de paquetes *HTTP*. Apegándose a la naturaleza del protocolo subyacente, las formas *HTML* no ofrecen ningún mecanismo para simular una interfaz con estados. Cada vez que el usuario solicita una página que contiene una forma, ésta es visualizada al igual que el resto del texto; el navegador se encarga de manejar la interacción entre el usuario y los controles de la forma; y en cuanto se activa alguno de los controles para enviar la solicitud, la página actual (incluyendo la interfaz de la forma) desaparece para siempre. Si el usuario vuelve a cargar la página en cuestión, la forma será inicializada (perdiendo así toda la información previa).

Apreciemos el siguiente ejemplo, donde usando únicamente formas *HTML*, pretendemos implementar la parte de la interfaz para nuestro servicio de altas de actividad:

```
<html>
  <head>
    <title>Alta de actividad de bitácora</title>
    <link rel="stylesheet" type="text/css" href="maquilla.css"/>
  </head>
  <body>
    <form action="http://localhost/servicios/bitacora_add" method="post">
      <br/> <label>Tipo:</label>  <input name="tipo"  type="text"/>
      <br/> <label>Fecha:</label> <input name="fecha" type="text"/>
      <br/> <label>Duración(hrs):</label> <input name="duracion_hrs" type="text"/>
      <br/> <label>Descripción:</label>
      <textarea name="descp" rows="5" cols="10"></textarea>
      <br/> <input name="invoca" type="submit" value="Enviar"/>
    </form>
  </body>
</html>
```

Para que este ejemplo funcionara, se necesitaría que estuviera definido algún programa (*CGI* o equivalente) en el *URL* al que apunta el atributo *action* del elemento *form*. A propósito, también incluímos el uso de *CSS* como mecanismo para darle algo de maquillaje a esta página *HTML*. Definitivamente, éste no es el lugar para un tutorial de este estándar, pero dado que lo usamos en la presente propuesta, deseamos al menos dar la lector una "probada" de lo que implica. *CSS* (*Cascade StyleSheet Language*) es un mecanismo a través del cual, se asocian conjuntos de propiedades visuales con los elementos de algún documento *HTML* (o dialecto *XML* en general). Esta asociación se realiza a través de un atributo (*class*) o simplemente del nombre de los elementos (en cuyo caso aplicaríamos una misma regla a todos los elementos del documento, con el mismo nombre). La inclusión de la hora de estilo *CSS* se realiza al principio del documento, después del título, con el elemento *link*. Veamos una posibilidad para el maquillaje de los elementos que etiquetan los nombres de los campos de la forma:

```
/* Aspecto para los elementos 'label' de la forma HTML */
form > label
{
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-size: 15px;
  font-weight: normal;
  background-color: black;
  color: white;
}
```

Con el bloque anterior, estamos asociando atributos de tipo de letra y color con todos las etiquetas de nuestra forma. La primera línea después del comentario, es un selector que determina

sobre qué elementos aplicará la regla; en este caso, sobre todos los nodos hijos del elemento *form*, que tengan por nombre *label*. El párrafo anterior tendría que ser parte del archivo *maquilla.css* que estamos incluyendo al principio. Con *CSS* se pueden controlar muchos otros aspectos, como la ubicación, tamaño, indentación, etc. Es una buena opción para separar el aspecto visual de nuestras interfaces con formas *HTML*, y en general, de los documentos *HTML*.

Bueno, pero regresemos a la discusión: esto es todo lo que nos ofrecen juntos *HTTP* y *HTML*, y encima de su oferta se han montado a través de los años, toda serie de artificios, capas o incluso infraestructuras completas; todo para simular que tenemos las herramientas necesarias para construir interfaces con estados, y que tenemos acceso directo a los eventos de interacción usuario-controles. Procederemos a listar algunas de estas tecnologías, a manera de brevario histórico (seguramente se encontrarán mejores referencias en otro lado, pero tratamos de darle cierta independencia a este documento):

Interfaz de acceso común (CGI: Common Gateway Interface). Es probablemente el inicio de la programación *Web*, y le agrega poco a la oferta *HTTP+HTML*. Esta propuesta establece un contrato entre los servidores *HTTP* y los programas que atenderán las peticiones originadas en las formas *HTML*, de manera que los primeros puedan informar a los segundos los valores introducidos en los campos (que se reciben de los navegadores) y los segundos prometen generar alguna respuesta en *HTML*, después de procesar la petición. También es necesario que los servidores *HTTP* ofrezcan un mecanismo para publicar estos programas a través de un *URL*, y de asociar estas direcciones con programas ejecutables. Prácticamente no hay restricción para el lenguaje en el cual podemos programar el *CGI*: mientras éste pueda leer los valores asociados a los campos de la forma y pueda generar texto como respuesta, es aceptado. Nótese que se suele asociar el nombre del mecanismo completo (*CGI*), con el componente programa.

Si pensamos en un entorno *UNIX* - lo cual no es casualidad, puesto que la mayoría de los servidores *HTTP* tiene alguna variedad de este sistema operativo - diríamos que el servidor *HTTP* le transfiere la información de la petición al programa, a través de variables de ambiente (entonces cualquier programa que las pueda leer, puede ser un *CGI*). También diríamos que el programa le comunica la respuesta (código *HTML*) al servidor, a través de su salida estándar, cuyo contenido sería redireccionado al navegador que originó la petición. Obviamente, con el paso de los años se fueron creando bibliotecas con las funciones más usuales para programación *CGI*, y lenguajes como *PERL* merecen mención especial en este rubro.

Preprocesadores de HTML (Server Side Scripting Languages). Dado que es muy tedioso generar *HTML* dinámico a través de instrucciones de bajo nivel (como el *printf* de C, por ejemplo), y observando que del código generado sólo determinadas porciones requieren ser

dinámicas, se crearon los preprocesadores de *HTML*. Como su nombre lo indica, estos sistemas dan una pasada previa a los archivos con *HTML*, buscando secciones especiales marcadas para ellos. En estas secciones se incluye código de algún lenguaje (usualmente interpretado) que al ser reconocido por el preprocesador es ejecutado y su salida (análoga a la de un *CGI*) es agregada al documento original. Así se procesan todas las secciones y como resultado tenemos un documento *HTML* donde ciertas partes se generaron dinámicamente.

Como es de suponerse, algunas de las partes que requieren del dinamismo, son aquellas relacionadas con las formas *HTML*. Por ejemplo, el código de una forma en particular puede ser generado en cada ocasión, usando ciertas variables que representan los valores introducidos hasta el momento en los campos; o bien, algún control de selección que necesite generar sus opciones en tiempo de ejecución, también requeriría este tratamiento. A diferencia de los *CGI*, los preprocesadores de *HTML* se suelen integrar al mismo servidor *HTTP*, por lo que la comunicación entre ambos es más directa. No es raro que estos lenguajes, asociados a preprocesadores, oculten los detalles de bajo nivel para obtener la información de la petición de una forma *HTML*, y que la ofrezcan con elementos nativos como las variables; asimismo, estos lenguajes ofrecen alguna ilusión de contexto, permitiendo guardar y administrar por cuenta propia valores que vienen de la forma *HTML*, entre invocación e invocación. Por otra parte, esta integración con el servidor *HTTP* supone una ejecución más eficiente, toda vez que no es necesario crear procesos nuevos para cada petición - aunque los mecanismos *CGI* también desarrollaron algunas opciones con procesos ligeros o hilos (*threads*), para solventar este problema.

Hoy en día, siguen siendo famosos algunos de estos lenguajes y sus preprocesadores. De hecho, la fama de varios de ellos probablemente se deba a sus licencias de código abierto. El internet es un lugar donde la tradición ha sido usar los sistemas operativos de la familia *UNIX*, y al ser éstos la cuna del software libre, dieron entrada a la adopción de muchos proyectos de ese tipo para aplicaciones en la *Web*. Ejemplos famosos de estos lenguajes son *PERL* y *PHP*, donde este último ha cobrado tanta fama que ya existen aplicaciones enteras (también de código libre) bajo su sello, e inclusive, algunas infraestructuras. La antípoda del mundo comercial para *PHP* sería el producto de Microsoft llamado *ASP* (que también ha tenido una penetración considerable en el mercado).

Servlets. Los *servlets* son parecidos a los *CGI*, en el sentido de que producen de manera autónoma la siguiente respuesta al cliente *HTTP*; pero difieren en que su implementación es un servidor de aplicaciones por sí solo (llamado contenedor de *servlets*), y no una mera incrustación al servidor *HTTP*. Esta opción cuenta con una especificación, dónde viene implicado el lenguaje a usar: Java.

Existe cierta controversia (o existía al menos) respecto a la naturaleza de los lenguajes para desarrollar aplicaciones en internet: ¿deben ser interpretados y débilmente tipados o todo lo contrario? Los lenguajes como *PHP* (que tomaremos como digno representante de la opción de los preprocesadores), han ganado muchos adeptos y aunque éste ha incluido un compilador (que optimiza bastante los programas), definitivamente sigue siendo débilmente tipado; lo cual significa que los tipos de sus variables no son resueltos hasta el tiempo de ejecución y que, inclusive, una variable podría adquirir muchos tipos distintos durante su vida en un flujo de ejecución particular. En contraste, tenemos a la propuesta del lenguaje *Java*: una infraestructura bien definida, donde los componentes para atender las peticiones se programan en ese lenguaje (que es fuertemente tipado) e inevitablemente requieren de compilación (aunque existen opciones para automatizar esta parte, el tiempo que implican podría parecer elevado comparado con los lenguajes de la opción anterior). Una diferencia más profunda entre estas propuestas, sería el paradigma de cada lenguaje: imperativo para *PHP* y orientado a objetos para *Java* (aunque *PHP* tiene algunos elementos de la OO, no deja de tener carencias en este sentido).

Es verdad que no existe la mejor solución en cuestión de tecnología, puesto que la habilidad y preferencias del programador son determinantes. Sin embargo, hay otras diferencias más que inclinarían la balanza para la solución de *Java*. Lenguajes como *PHP* fueron creados explícitamente para internet, por lo que las tareas de programación características del manejo de formas *HTML* son relativamente fáciles de manejar; sin embargo, cualquier otra cosa que necesitemos (programación en general), necesitará de un puente para hacerlo accesible (las bibliotecas genéricas usualmente están programadas en C o Java, no en PHP). En contraste, *Java* es un lenguaje genérico y no cambió para adaptarse a la *Web*. Usando el propio lenguaje se definió toda una infraestructura de clases, que implementan los componentes necesarios (pero podemos seguir usando cualquier otro componente desarrollado en Java); mientras que *PHP* es excelente como una introducción al desarrollo *Web*, pero exige poco al programador en cuestión de disciplina y modularidad. Y a medida que las aplicaciones se hacen más grandes y complejas, éste podría ser un lujo demasiado caro. El último aspecto es la estandarización: lenguajes como *PHP*, aunque libres, son productos específicos, y tanto *Java* como los *Servlets* cuentan con una especificación, dando lugar a tener libertad de implementación (de las que existen opciones de código abierto y cerrado, gratuitas y con precio, etc.).

Infraestructuras montadas sobre servlets. Otra característica que le pronosticó buenos augurios a los *servlets* cuando nacieron, fue el hecho de que ofrecían una sólida y robusta base para agregar nuevas capas de componentes; que promovieran elevar el nivel de abstracción para programar aplicaciones *Web*, o por lo menos, que facilitaran ciertas tareas. La existencia - rallando en la saturación - de esos sistemas montados sobre *servlets*, corroboran esas profecías. Como primera mención, tenemos a los *JSP* (*Java Server Pages*) que es un intento por evitar que

los programadores extrañen o se sientan tentados a regresar a opciones como *PHP*. Se creó un lenguaje de etiquetas y un preprocesador para las mismas en Java (con su propia especificación). Las implementaciones de los *JSPs* generan código de tipo *Servlet* automáticamente. Aunque efectivamente ofrece mayor flexibilidad a los programadores, en nuestra opinión abre la puerta a los viejos vicios, como mezclar el maquillaje de las páginas con la lógica de control.

Pero si de separar lógica de control y maquillaje se trata, *XML* es la referencia obligada; y es que desde su creación este metalenguaje ha sido la tentación para estas infraestructuras. Algunas parecen haber logrado su integración, y asumiendo *backends* que "hablan" *XML*, integran la información que éstos regresan con aquella propia de la página, para después transformar el código *XML* en *HTML* e inclusive, algunos llegan a modularizar el maquillaje usando un lenguaje adicional creado exprofeso para ello y que se acopla a la perfección con *HTML*: *CSS* (*Cascade StyleSheet Language*). Con éste último es posible separar el aspecto del maquillaje de gran parte de los documentos *HTML*.

XForms. Desde el punto de vista de esta propuesta, *XForms* se ubica en la cúspide de esta escala evolutiva de tecnologías. No sólo es estándar (aceptado y definido por la industria), sino que aplica consistentemente el patrón *MVC*: maquillaje y lógica de control, nunca habían estado más separadas. Además, también adopta a *XML* como su medio de expresión y puede convivir con otros dialectos instancias de este metalenguaje (puede invocar *WebServices*, por ejemplo). Por si todo esto fuera poco, hay un par de grandes atractivos: ¡las interfaces de usuario se programan declarativamente! así es, adiós al código imperativo y tedioso para controlar las acciones a tomar, en función de las entradas del usuario. El otro atractivo es la independencia de tecnología: aunque *XForms* cuenta actualmente con implementaciones para la *Web*, no está casado con esa opción. De hecho, el estándar nos permite programar la lógica de nuestras interfaces una sola vez, sin preocuparnos por detalles de nivel tan bajo, como las dificultades para conservar el contexto en un ambiente *Web*. Será responsabilidad de las implementaciones ofrecer diversos "sabores" para ejecutar nuestras interfaces: ventanas tradicionales, interfaces *Web* ¿y por qué no?, hasta las subestimadas interfaces modo texto para consola (no excluyamos tampoco, las emergentes interfaces para dispositivos móviles).

Tampoco podemos decir que *XForms* sea la panacea de las interfaces, pero lo es por lo menos para las necesidades de gran parte de los sistemas de información; se presenta como una opción bastante superior a las anteriores - probablemente su única desventaja actual sea su corta edad: apenas se está dando a conocer el estándar y las implementaciones están en proceso de maduración ... pero es sólo cuestión de tiempo, en la opinión del autor de estas líneas -.

5.4. Hola Mundo en XForms

Después de este breve recorrido por las tecnologías para desarrollar en la *Web*, diremos que los ejemplos listados estuvieron ampliamente motivados por las experiencias personales del autor y si no fueron mencionadas otras ofertas, fue porque se percibió una falta de autoridad "moral" al respecto (nada como ensuciarse los dedos, para darse una idea de la tecnología). Es claro que situamos a *XForms*, como la selección natural de la infraestructura *SIUX* para la capa 4. Pero trataremos de justificar más esta opinión, ejemplificando el uso del estándar.

Continuemos la tradición de *Ritchie* (el padre del lenguaje C), y comencemos con una *XForm* que simplemente salude al mundo que ve nacer al estándar. Este sería el código correspondiente:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- Una xform mínima: ejemplo "hola mundo" -->

<html
  xmlns="http://www.w3.org/2002/06/xhtml2"
  xmlns:xf="http://www.w3.org/2002/xforms"
  xmlns:css="http://www.w3.org/TR/REC-CSS2"
>
  <head>
    <xf:model>
      <xf:instance>
        <data>
          <saludo>Hola mundo!!!</saludo>
        </data>
      </xf:instance>
    </xf:model>
  </head>

  <body>
    <xf:output ref="/data/saludo" css:class="texto1"/>
  </body>
</html>
```

Hay varios detalles a notar en este pequeño listado:

1. El documento que especifica la interfaz es, a final de cuentas, un documento *XML*, lo cual nos lleva a inferir o recordar que *XForms* es otro lenguaje instancia del metalenguaje *XML* (al igual que *XMLSchema* del capítulo anterior).

2. Nótese que estamos usando 3 espacios de nombres *XML*: uno para los elementos sin prefijo (que apunta al *URL* asociado al estándar *XHTML*, una versión *XML* de *HTML*); otro para los elementos de *XForms*; y uno más para los nombres propios de *CSS* (que por cierto sólo usamos en un atributo).

3. El lenguaje definido por el estándar *XForms*, no es independiente y necesita de algún otro dialecto *XML* que le dé hospedaje. En este caso, el anfitrión es *XHTML* (obsérvese que nos estamos absteniendo de incluir cualquier elemento que no tenga que ver con la interfaz *XForm*, únicamente usamos un par de agrupadores, pero nada más). Entonces, el lenguaje huésped nos sugiere que la implementación que tenemos en mente con este ejemplo, es una para la *Web*. Aunque ello es cierto en este caso, no tiene que serlo siempre: una implementación de *XForms* podría usar *XHTML* como lenguaje anfitrión, y ofrecer una solución de ventanas (por ejemplo).

4. El modelo en una interfaz *XForm*, se especifica en el elemento *model*. Dentro de éste, definimos un esqueleto que contendrá todos los datos que vayamos a manejar (elemento *instance*), así como las invocaciones a los servicios del *backend* y su asociación con las partes de los datos que afectan. En este caso, sólo pusimos la parte de los datos, no estamos invocando a ningún servicio.

5. Todo lo que esté fuera del elemento *model* de *XForms*, es considerado parte de la lógica de control (especificada estructural y declarativamente). Sin embaro, no hay un elemento agrupador, así que aprovechamos uno del lenguaje anfitrión (*XHTML*) y manejamos como convención el que todo contenido del elemento *body* (pertenece al espacio de nombres predefinido) será parte de la lógica de control de la interfaz. En este caso, el único control que estamos incluyendo es un *output*, que simplemente permite visualizar algún dato texto del modelo. La manera de indicar qué dato se va a mostrar, es a través de *XPATH*, otro estándar relacionado con *XML*, aunque no una instancia del mismo. *XPATH* define un lenguaje para acceder partes de documentos *XML*, que asemeja (en su versión más sencilla) las rutas de los sistemas de archivos.

6. Note que sólo estamos indicando los datos y los controles; en ningún lado dice algo referente a la apariencia de la interfaz. Cuando la implementación para la *Web* transforme el código aquí mostrado en puro *XHTML*, también seleccionará los atributos y elementos que den algún formato predeterminado a nuestra pantalla (posiblemente, usando *CSS* para ganar modularidad en ese aspecto). Sin embargo, si quisiéramos sobreescribir este *maquillaje* predeterminado, podemos forzarlo si nuestra implementación de *XForms* usa *CSS*, simplemente asignando el valor adecuado al atributo *class* (espacio de nombres para *CSS*) de los elementos. Dicho valor,

deberá representar algún grupo de instrucciones de formateo definidas por nosotros y que tendremos que poner a disposición del navegador, para cuando éste reciba la página generada por el procesador de *XForms* (una sencilla instrucción en *XHTML* bastaría para tal fin).

Para el ejemplo anterior, debemos recalcar que el procesador no tendrá necesidad de recordar ningún estado, ya que la propia interfaz no tiene ninguna funcionalidad que lo reclame. A pesar de que las interfaces hechas en *XForms* tienen predefinido un ciclo de vida iterativo y con estados (inicialización, validación, etc.), si el programador no incluyó ningún control, esos cambios serán inaccesibles para el usuario (como fue este caso). La situación normal, es que el procesador de *XForms* (como suele llamársele a las implementaciones del estándar) tenga varias interacciones con el navegador, que transite por diversos estados y vistas, modificando los datos del modelo gracias al *backend*. En cada uno de estos ciclos, se genera una conversión del lenguaje *XForm* a *XHTML* para los controles.

Antes de continuar, démosle la justa mención a la implementación de *XForms* que usamos: *Chiba*. Creada por *Joern Turner*, y mantenida actualmente por varias personas más, *Chiba* puede presumir actualmente de ser la única implementación libre del estándar que yace en el lado del servidor. Y es que tenemos dos alternativas para implementar los procesadores de *XForms*: en los navegadores de internet o en el servidor *Web* (que podría ser un servidor de aplicaciones). Mientras que el lado del servidor tiene algunas limitantes (dado que requiere envios *HTTP* para captar cualquier evento), también tiene la ventaja de ser directamente aplicable en la actualidad; aceptémoslo, pasará todavía algún tiempo antes de que los navegadores más famosos soporten el estándar naciente. Pero si dejamos en el servidor la implementación, y sólo dependemos de la oferta *HTTP + HTML* para montar *XForms*, permitiremos a los navegadores actuales entrar al juego ... esa es precisamente la apuesta de *Chiba*.

La implementación *Chiba* de *XForms*, tiene un diseño gratificantemente modular que permitió hacerle las adecuaciones necesarias, sin mucha dificultad. Estas adaptaciones son una de las contribuciones de la infraestructura *SIUX*, en esta capa 4 (la otra es un generador de interfaces *XForms*, que trateremos más adelante). Las modificaciones fueron necesarias no por defectos, sino porque *Chiba* se apega lo más que puede al estándar, y éste aún tiene ciertos huecos que están por llenarse en revisiones futuras. Entonces, como no pensábamos esperar a que alguno de los desarrolladores del proyecto se apiadara de nuestras ansiosas necesidades, se decidió hacer las implementaciones por nuestra cuenta (afortunadamente no fueron muchas). Obviamente, esta facilidad sería simplemente imposible con alguna implementación comercial de código cerrado (por algún momento, se consideró usar la implementación de *Novell*, pero nos retractamos por la falta de soporte).

Aparte de estas modificaciones a ciertas clases de *Chiba* (está hecho en Java), se cambiaron

los transformadores de *XForms* a *XHTML*, porque no deseábamos depender de ningún atributo visual que no fuera agregado por medio de *CSS*. Estos programas se elaboraron en *XSLT* (*eXtended Stylesheet Language Transformations*). Dicho lenguaje permite transformar documentos *XML* en otras estructuras (no necesariamente *XML*) y tiene la peculiaridad de ser prácticamente funcional en su paradigma (nos atrevemos a sugerir que también es declarativo); por lo cual dicho lenguaje encaja a la perfección en nuestra propuesta. Otro punto que merece mención, es que los transformadores que se elaboraron usan *XHTML* en su mínima expresión (en cuanto al número de etiquetas empleadas): usando solamente etiquetas *div* y atributos, es posible simular cualquier estructura *XML*. Necesariamente, esta decisión implicó desarrollar nuestra propio diseño con *CSS* (es raro ver facultades artísticas en los desarrolladores, y ésta no fue la excepción).

Este patrón de uso de *XHTML*, acerca el futuro no tan lejano donde *XML* será directamente procesado por los navegadores, aunque ello también involucrará lenguajes que le digan a los navegadores *qué* hacer con los datos *XML* (un ejemplo podría ser recibir los datos de una página en *XML*, y adicionalmente recibir un documento en *XSLT* que indique cómo transformarlo en *XHTML*). Es decir, la separación de los datos y su presentación se indica desde el servidor, pero el cliente es el responsable de mezclar ambos aspectos.

5.5. Invocación de WebServices con XForms

Es hora de retomar nuestra pequeña aplicación ejemplo, el mantenimiento de la tabla *mtto_bitacora*. Veamos cómo implementaríamos en *XForms* la interfaz para ofrecer al usuario el servicio de alta de actividades.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<html
  xmlns="http://www.w3.org/2002/06/xhtml2"
  xmlns:chiba="http://chiba.sourceforge.net/2003/08/xforms"
  xmlns:css="http://www.w3.org/TR/REC-CSS2"
  xmlns:xf="http://www.w3.org/2002/xforms"
  xmlns:xhtml="http://www.w3.org/2002/06/xhtml2"
>
  <head>
    <xf:model>
      <xf:instance>
        <data>
          <bitacora_add>
            <in>
              <in_tipo/>
              <in_fecha/>
              <in_duracion_hrs/>
```

```
            <in_descp/>
          </in>
          <out>
            <out_errcode/>
            <out_descp_errcode/>
            <out_numero/>
            <out_tipo/>
            <out_fecha/>
            <out_duracion_hrs/>
            <out_descp/>
          </out>
        </bitacora_add>
      </data>
    </xf:instance>

    <xf:submission
      id="sub_bitacora_add"
      action="soapsm://localhost/axis/services/psm2ws?oper=dario.mtto.bitacora_add"
      method="post"
      ref_in="/data/bitacora_add/in"
      ref_out="/data/bitacora_add/out"
      replace="instance"
    />
  </xf:model>
</head>

<body>
  <xf:group css:class="siux_form">
    <xf:group css:class="links"/>

    <xf:group css:class="title">
      <xf:label>Alta de Bitácora</xf:label>
    </xf:group>

    <xf:group css:class="fields">
      <xf:input ref="/data/bitacora_add/in/in_tipo" xhtml:maxlength="12">
        <xf:label>Tipo</xf:label>
      </xf:input>
      <xf:input ref="/data/bitacora_add/in/in_fecha" xhtml:maxlength="10">
        <xf:label>Fecha</xf:label>
      </xf:input>
      <xf:input ref="/data/bitacora_add/in/in_duracion_hrs" xhtml:maxlength="3">
        <xf:label>Duración(hrs)</xf:label>
      </xf:input>
      <xf:textarea
        ref="/data/bitacora_add/in/in_descp"
        css:class="fieldta"
        xhtml:cols="10" xhtml:rows="5"
      >
        <xf:label>Descripción</xf:label>
      </xf:textarea>
      <xf:output ref="/data/bitacora_add/out/out_errcode">
        <xf:label>Codret</xf:label>
      </xf:output>
      <xf:textarea
        ref="/data/bitacora_add/out/out_descp_errcode"
        css:class="fieldta"
        xhtml:cols="10" xhtml:rows="5"
      >
```

```
        <xf:label>Descripción Codret</xf:label>
      </xf:textarea>
      <xf:output ref="/data/bitacora_add/out/out_numero">
        <xf:label>Número</xf:label>
      </xf:output>
      <xf:output ref="/data/bitacora_add/out/out_tipo">
        <xf:label>Tipo</xf:label>
      </xf:output>
      <xf:output ref="/data/bitacora_add/out/out_fecha">
        <xf:label>Fecha</xf:label>
      </xf:output>
      <xf:output ref="/data/bitacora_add/out/out_duracion_hrs">
        <xf:label>Duración(hrs)</xf:label>
      </xf:output>
      <xf:textarea
        css:class="fieldta"
        ref="/data/bitacora_add/out/out_descp"
        xhtml:cols="10"
        xhtml:readonly="readonly"
        xhtml:rows="5"
      >
        <xf:label>Descripción</xf:label>
      </xf:textarea>
    </xf:group>

    <xf:group css:class="opts">
      <xf:trigger>
        <xf:label>Aceptar</xf:label>
        <xf:action>
          <xf:dispatch name="xforms-submit" target="sub_bitacora_add"/>
          <xf:refresh/>
        </xf:action>
      </xf:trigger>
    </xf:group>
  </xf:group>
 </body>
</html>
```

Por fin nos encontramos frente a frente con una interfaz hecha en *XForms*, no dejemos que la cantidad de líneas de "código" *XML* nos intimide. Es verdad que *XML*, o más bien, sus lenguajes instancia no ofrecen precisamente compacidad (es el precio por la sencillez y uniformidad). Pero las mismas secciones que vimos en el ejemplo "hola mundo" están aquí, con excepción de una invocación al *backend*. Vayamos en orden, primero vemos que en la sección de los datos agregamos una estructura que refleja las entradas y salidas del servicio *bitacora_add*. Hay una discrepancia entre esta estructura de datos y aquella que reporta la capa 3 de *SIUX* a través del *XMLSchema*: mientras que el *SIUX* reporta una sola sequencia de campos, aquí vemos un agrupamiento por concepto de entrada o salida. La razón de esta separación radica en facilitar la invocación del servicio, ya que, cuando lo hagamos, necesitamos decirle al procesador *XForm* qué porción de nuestro esqueleto de datos servirá para tomar la entrada y cuál está destinado a la salida del *WebService*. Hacer esta indicación sin un agrupamiento como el mostrado sería poco elegante y tedioso. Fuera de esta diferencia, el lector puede corroborar la fidelidad de este cascarón de datos,

con la estructura especificada por el *XMLSchema* que mostramos en el capítulo anterior para este servicio.

Entonces, lo que vemos dentro del elemento *instance* del espacio de nombres para *XForm*, sirve para indicar la estructura de datos que contendrá toda la información que la interfaz estará manipulando (no sólo la estructura, también pueden darse valores iniciales). Las interfaces *XForm* fueron creadas para ser intermediarios entre sistemas y usuarios; cuando una persona quiera invocar algún servicio, llenará ciertos controles de la interfaz y con ello, indirectamente, estará llenando cierta sección de los datos de la instancia (dado que en *XForm* los controles siempre deben estar asociados con algún dato de este cascarón que estamos comentando).

Después de la instancia, pero siguiendo dentro del modelo, vemos las etiquetas que especifican la comunicación con la capa anterior: las invocaciones a los servicios del *backend*. La etiqueta *submission* del espacio de nombres para *XForms* le proporciona al procesador la siguiente información:

id: Un identificador (único en todo el documento *XML*) para distinguir esta invocación de otras. A través de este valor será cómo los controles de la interfaz le comunicarán a esta parte del modelo que se desea una invocación al servicio, y ello implicará la afectación de los datos involucrados.

action: La dirección *Web* donde podemos localizar el servicio. Efectivamente, se trata de los ya mencionados *URLs*, aunque cuando los presentamos dimos la explicación pensando en documentos. Aquí se trata de la invocación de un programa, así que revisemos nuevamente: primero tenemos el protocolo a usar para comunicarse con el servidor de aplicaciones que ofrece el servicio. El nombre *soapsm* no es ningún estándar sino una convención de *SIUX* para indicarle a *Chiba* que use nuestro invocador de *WebServices - SOAP* se refiere al protocolo de presentación a usar con los datos de la invocación; ya sabemos que serán *XML* pero deben de ir contenidos en una estructura que haría el papel de "sobre" del mensaje. Así que, una vez que *Chiba* utilice internamente al conector desarrollado con *Axis*, el nombre del protocolo será reemplazado por *http* (que es el estándar preferido para publicar *WebServices*). *Chiba* tiene varios conectores, que nos permiten comunicarnos con servidores que implementan tecnologías diversas; la única manera de indicarle qué lenguaje deseamos usar para pedirles algo, es con el prefijo del protocolo del *URL* (*soapsm* en este caso).

Después tenemos el nombre de la máquina que debe resolver a una *IP* (*localhost* es un nombre que usualmente se asigna a la dirección local *127.0.0.1*); como no hay puerto *TCP* indicado explícitamente, se asume el estándar para *HTTP* (es decir, el puerto 80). Luego tenemos una ruta parecida a la de un archivo (*/axis/services/psm2ws*); a diferencia de los documentos, para

el caso de los programas (*como un WebService*) estas rutas suelen ser completamente arbitrarias y no necesitan de una justificación en el sistema local de archivos. Por último, tenemos un signo de interrogación y una pareja de variable-valor. Esta es la notación para transmitir parámetros a los programas que se publican en la *Web* (en particular, del *Servlet* que *Axis* utiliza para publicar los servicios que administra; recuerde que *Axis* es nuestra implementación de *WebServices*). Aquí se aprovechó esta notación para indicar qué servicio deseamos invocar (*bitacora_add*, para este ejemplo).

La notación de puntos para el nombre designa respectivamente: esquema de la base de datos, nombre del paquete (que coincide con el usado para la aplicación) y el nombre del procedimiento almacenado. Recuerde que nuestro *WebService* genérico *psm2ws* es un intermediario dinámico entre cualquier procedimiento almacenado que ofrezca la capa 2 del *backend* y la interfaz de usuario. Notamos aquí cierta porosidad que existe en nuestra arquitectura: la capa 4 tiene conciencia de la capa 2 (sabe que ofrece tal o cual procedimiento), sin embargo no se comunica directamente con ella, sino que usa de puente funcionalidad de la capa 3.

method: Como a final de cuentas nuestra petición *XML* dentro de un sobre *SOAP* viajará a través de un paquete *HTTP*, podemos decidir qué parámetros usamos del mismo. *HTTP* permite dos formas de enviar los datos: *GET* y *POST*. La principal diferencia entre los dos métodos radica en la no repetibilidad del segundo. A manera de anécdota, mecionaremos que hasta hace algunos años, una característica de ciertas implementaciones del protocolo *HTTP* (*MIE*) imponían ciertas limitantes de tamaño a los datos que podían viajar en un envío *GET* - conviene esta precaución, ya que nos encontramos invocando servicios de cualquier índole, cuyas entradas y salidas podrían ser considerablemente grandes.

En realidad, mencionamos el uso de este atributo por completez: ésta sería la función que tendría, si el conector seleccionado de *Chiba* lo usara. Pero para nuestro caso particular, el conector designado con el nombre *psm2ws* (parte de la infraestructura *SIUX*), no lo emplea - aunque probablemente las funciones de *AXIS*, a final de cuentas deciden usar éste formato de serialización cuando arman sus paquetes *HTTP*.

ref_in y ref_out: Indican la rutas *XPATH* para localizar los nodos *XML* de los datos, que servirán como entrada y salida del servicio invocado (un *WebService* en este caso). Originalmente, el estándar *XForms* sólo contaba con una sola ruta, así que el mismo lugar de donde salían los datos era sobreescrito con la respuesta (ver siguiente atributo). Para evitar ello, se adicionaron estas rutas a la sintaxis y se preparó a *Chiba* para manejarlas (éste fue uno de los cambios que se hicieron, y gratificantemente se vio en la página del estándar que pronto será integrada esta

facilidad).

replace: El valor de este atributo, implica la acción a tomar por parte del procesador de *XForms*, una vez que haya sido realizada la invocación al servicio. Para nuestro ejemplo, al indicar *instance* estamos pidiendo que sólo sea remplazada la parte de los datos indicada en el atributo *ref_out*. Otros posibles valores son: *all*, que reemplaza la *XForm* completa por el código *XML* que nos regrese el servicio, o bien, *none* para señalar que no deseamos reemplazar ninguna sección (para aquéllos casos donde sólo nos interesa la mera acción del servicio).

Después del modelo, vemos que los controles están agrupados en una estructura *group*. Esta etiqueta es un agrupador genérico de *XForms* y no interfiere en la lógica de control (es sólo para clarificar la jerarquía de los controles al programador o bien, para facilitar la transformación posterior en *XHTML* o algún otro lenguaje; ambas aplican para *SIUX*). En el caso de la infraestructura que estamos proponiendo, de todas las posibles interfaces que se podrían construir bajo *XForms*, estamos restringiéndonos a un solo patrón y así garantizamos la uniformidad y simplicidad de nuestra oferta. La ganancia por seguir esta regla es poder usar las facilidades de la capa 4 de *SIUX*.

5.6. Modularidad en las interfaces de SIUX: Vistas.

Una interfaz de usuario para *SIUX* está dividida en varias "vistas", de las cuales sólo podrá estar activa una a la vez. En nuestro ejemplo, sólo hay una vista (así que no requerimos aún de un selector). Cada una de estas vistas representa una porción de la funcionalidad total de la interfaz, y cuenta con una estructura propia que predefine *SIUX*. A los componentes de cada vista los identificamos con los valores del atributo *class*, del espacio de nombres para *CSS*, que se usó para marcar las secciones correspondientes. Ese atributo se usa posteriormente para darle el formato visual adecuado a las vistas, cuando llegue el momento de convertirlas en *XHTML*:

ligas (links): Esta sección define los controles necesarios para permitir una navegación por la interfaz, de la cual forma parte la vista. Podemos pensar a la interfaz en su totalidad como un árbol de nodos, donde cada uno es una vista o un selector de vistas. Si en algún punto determinado del árbol quisiéramos regresar a un nivel superior, recurriríamos a estas ligas. Para nuestro ejemplo únicamente tenemos una vista, así que no es necesario tener controles de navegación (pero no desespere ud., en un ejemplo posterior ilustraremos esta parte de las vistas de *SIUX*).

título (title): El título de la vista, no sólo es desplegado en la misma a manera de leyenda informativa. También es empleado en los selectores de vistas.

campos (fields): La principal función de las vistas es permitir la invocación de algún servicio del *backend*, para lo cual deben aceptar a través de controles, las entradas para el mismo; así como contar con otros controles para desplegar los resultados de la invocación. Ambos tipos de controles quedan englobados bajo el nombre genérico de "campo" bajo *SIUX*, y se incluyen en esta sección. Una excepción son los servicios que regresan listas de registros (los llamados *out_cur*), dado que éstos son desplegados en una sección aparte. Aunque son un parámetro de salida, y por tanto caerían bajo la definición dada de campo, la estructura que contienen reclama su propio espacio (muy bien; aceptamos que hay otra razón más y es facilitar la conversión de las vistas en *XHTML*, especialmente la especificación de su maquillaje con *CSS*).

Observamos la aparición de dos nuevos tipos de controles (*input* y *textarea*), los cuales sirven para ingresar a la interfaz los parámetros necesarios para invocar al servicio en cuestión; la diferencia entre ambos radica en el tamaño del texto que pretenden capturar, ya que las áreas de texto (*textarea*) están pensadas para valores grandes. Para el primer caso, nótese que aprovechando la información que nos proporcionó la capa 3 vía un *XMLSchema*, podemos especializar más el código *XHTML* que será generado (el atributo *siux_len* ha pasado su valor al atributo *maxlength* del espacio de nombres *XHTML*). Esta especificación de la longitud máxima no tiene que ver tanto con la apariencia de nuestra vista (por ello está incluido aquí), sino con la funcionalidad (independiente de cómo luzca el control, sólo aceptará determinada cantidad de carácteres). Una situación opuesta ocurre con el número de columnas y renglones para las áreas de texto, las cuales no tienen que ver tanto con el tamaño máximo de los valores ingresados para aceptar (que de hecho, queda indeterminado), sino el tamaño visual del control. A pesar de que no necesitamos incluir aquí esos atributos (por no ser parte esencial de la lógica de control), *SIUX* nos permite sobreescribir los valores predeterminados.

También observamos, con cierta satisfacción, que la información ingresada en los catálogos de la capa 1 referente a las columnas de las tablas (nombre, etiqueta) han viajado hasta este lugar de manera consistente y automática (de momento, lo único que requiere codificación manual es la interfaz *XForm*, pero veremos más adelante que también podemos generarlas automáticamente). Esta es parte de la filosofía de *SIUX*, el especificar sólo una vez este tipo de información (principalmente, en las capas 1 y 2) y habilitar la comunicación necesaria entre las capas para que ésta se vaya transmitiendo y reutilizando. Tampoco es requisito que tengamos todo en catálogos explícitos, ya que los *SMBD* cuentan con catálogos para prácticamente toda la metainformación de interés. Por ejemplo, la longitud máxima no fue incluida en ningún catálogo, sino que fue tomada de la propia definición de la tabla y propagada a través de los distintos componentes

hasta esta capa 4.

opciones (opts): Las funciones de los controles situados bajo esta sección, varían de acuerdo al tipo de vista. Para nuestro ejemplo, donde queremos invocar al servicio de altas de actividad, el único control que aparece aquí es uno que activa la invocación del *WebService bitacora_add* (esto se logra enviando el evento adecuado a la parte del modelo asociada con ese servicio; es decir, el elemento *submission* con el identificador *sub_bitacora_add*). Para otro tipo de situaciones, estos controles podrían habilitar el flujo hacia otra vista.

Si recordamos la mención de algunos párrafos arriba, donde se dice que una propuesta de modelación para las interfaces de usuario son los autómatas de estado finito, entonces ya tenemos un candidato para representar los estados del autómata que simule una interfaz de *SIUX*: la vista. Las transiciones entre las vistas serían determinadas por los eventos que sean generados, gracias a la interacción del usuario con los controles - ubicados en las secciones de ligas (*links*) y opciones (*opts*) de las vistas. De hecho, lo que vendría siendo la función de transición de nuestro autómata, también podría ser vista como el mecanismo de navegación en nuestra interfaz de *SIUX* (que tiene estructura arborescente). Para platicar de estas transiciones, primero debemos clasificar las vistas.

Vistas de invocación a WebServices: Son las que ya hemos presentado, ejemplificando con la invocación al servicio *bitacora_add*.

Listas de vistas: Selectores de vistas (que a su vez podrían ser otras listas). Pueden ser pensadas como una versión abstracta de los menús de funciones, tan famosos en las interfaces de ventana.

Vistas auxiliares de control: A veces, las vistas que invocan *WebServices* son partidas en varios pedazos (que también son vistas), para lograr el efecto deseado en la interacción. Por ejemplo, si deseamos que el usuario confirme su invocación antes de efectivamente llamar al servicio; o bien, cuando requerimos que las salidas sean mostradas en una pantalla aparte. Otra opción podrían ser los procesos que únicamente deseamos invocar una sola vez por interacción (después de que fue invocado el servicio, obligamos al usuario a reiniciar la vista antes de permitir una nueva llamada). En todos estos casos, requerimos de una o más vistas auxiliares (así como las transiciones entre éstas y la vista principal). Entonces, la asociación de estos tipos de vista con un servicio del *backend*, se da indirectamente: puede que no invoquen al servicio, pero tienen transiciones con una vista que sí lo hace.

Vistas detalle: Ya dijimos que cuando los servicios regresan una lista de registros (como las consultas), entonces ese parámetro de salida se maneja separado de la lista de campos de entrada/salida de la vista. Un patrón común en las interfaces de usuario es el de lista-detalle, donde a primer nivel (la vista con el resultado del servicio), sólo vemos ciertos campos de los registros en la lista; al "taladrar" alguno de ellos, entramos en una nueva vista donde pueden ser mostrados con mayor comodidad el resto de los campos. Estas son las vistas detalle en *SIUX*, y junto con las listas de vistas, son el mecanismo para darle profundidad a nuestra interfaz de usuario (pensándola como un árbol).

Aparte de proporcionar más información de los registros de una búsqueda, estas vistas detalle también son el lugar donde podemos ligar los resultados de los servicios. Es decir, el resultado de un servicio (de tipo consulta) puede ser reutilizado como la entrada de otro servicio. Entonces, en la sección de opciones de estas vistas detalle, pueden aparecer controles que nos activen otras vistas de invocación de *WebServices*, con algunos campos prellenados (utilizando algún registro regresado en la primera invocación como contexto).

Aprovechando la mención de todos los tipos de vistas que contempla la infraestructura *SIUX*, revisemos como queda establecido el mecanismo de navegación por toda la interfaz (que estamos pensando como un árbol de vistas). Para adentrarnos un nivel abajo, podemos seleccionar el detalle de algún elemento de un parámetro de salida tipo "lista de registros" (que sería parte de los campos de alguna vista), o algún elemento de una lista de vistas; también podemos considerar que seleccionar alguna opción de una vista detalle nos adentra más en el árbol. Para movernos entre las vistas de un mismo nivel, podemos interactuar con los controles de opción de las vistas involucradas en la invocación de un servicio. Para subir a niveles superiores que tenemos como contexto, podemos usar el control con las ligas de la vista (ubicado en la sección con el mismo nombre).

Apliquemos estos términos sobre las interfaces de *SIUX* a nuestra interfaz de mantenimiento - que practicamente será un microtutorial de cómo implementar ciertos patrones comunes de interfaces de usuario, empleando *XForms*. Pensemos en la versión final de la interfaz de mantenimiento para nuestra aplicación: deseamos que tenga un menú desde donde sean accesibles el alta y la consulta. Para la consulta, se desea que los registros tengan detalle y que éste sea reutilizable para invocar las operaciones de baja y modificación. Como estas dos operaciones (baja y actualización) son más delicadas, pediremos que para bajas nos den confirmación y que sea no repetible (en una misma sesión con la vista); también que la actualización muestre sus salidas por separado. Para tener un pretexto de introducción a los controles de selección, también pediremos que el tipo de la actividad sea tomado del catálogo correspondiente.

5.7. El ejemplo de mantenimiento

Como la presentación del código completo de la interfaz en *XForms* ocuparía varias páginas, y ello dificultaría las explicaciones relacionadas, sólo iremos mostrando los trozos relevantes. Comencemos con la parte del modelo, de la cual ya sabemos, tenemos dos secciones: la instancia inicial de los datos, y las definiciones de las invocaciones a los servicios del *backend*. Respecto a la primera parte, naturalmente habría que agregar más esqueletos, dado que pretendemos invocar a cuatro servicios distintos. Adicionalmente, agregaremos una sección de datos de control (recuerde que sólo mostraremos la parte más relevante del listado; donde se vean tres puntos suspensivos, quiere decir que la parte faltante se puede inferir y por ello fue omitida):

```
<xf:model>
  <xf:instance>
    <data>
      <ctrl>
        <back>
          <bitacora_add/>
          <bitacora_con/>
          <bitacora_del/>
          <bitacora_upd/>
        </back>
      </ctrl>

      <bitacora_add>
        <in>
          <in_tipo/>
          ...
        </in>
        <out>
          <out_numero/>
          ...
        </out>
      </bitacora_add>
      <bitacora_con>...</bitacora_con>
      <bitacora_del>...</bitacora_del>
      <bitacora_upd>...</bitacora_upd>

      <tipo_actividad_con>
        <in/>
        <out/>
      </tipo_actividad_con>
    <data>
  </xf:instance>
...
</xf:model>
```

La parte de control que agregamos, previa a los moldes para los parámetros de entrada/salida de los servicios, sirve para permitir la navegación hacia niveles superiores en la interfaz. Como

tendremos vistas que estarán hasta tres niveles abajo de la raíz de nuestra interfaz, el control ubicado en las ligas de la vista podrá tener varias opciones para regresar (una por cada nivel precedente). El usuario puede escoger, gracias a un control de selección (que se mostrará adelante), pero necesitamos que el nombre de la vista sea determinado en tiempo de ejecución (por ello, necesitamos esas variables por cada vista principal de invocación). Esta característica, por muy obvia que pueda parecer, no es parte del estándar *XForms* y fue de los agregados de *SIUX* para *Chiba*.

También agregamos un nuevo servicio, que obviamente asumimos como disponible en el *backend*: *tipo_actividad_con*. Éste es el servicio de consulta del catálogo de tipos de actividades de bitácora, y lo agregamos a los datos porque lo invocaremos más adelante (recuérdese que pedimos que el tipo de la actividad fuera seleccionado de una lista predeterminada). Este servicio no será accesible al usuario a través de una vista, así que no requiere su sección en los datos de control. Recuerde que la codificación del procedimiento de consulta y su publicación serían cortesía de la infraestructura *SIUX* (según las facilidades revisadas en los capítulos previos).

La siguiente parte a definir del modelo son las invocaciones a los *WebServices* de mantenimiento que nos ofrece la capa 3. La única diferencia con el ejemplo previo que vimos a detalle, es que aquí tendríamos que incluir uno por cada servicio. Nótese que cada uno ocupa su correspondiente pedazo de los datos de la instancia:

```
<xf:model>
  <xf:instance>
  ...
  </xf:instance>

  <xf:submission
    id="sub_bitacora_add"
    action="soapsm://localhost/axis/services/psm2ws?oper=dario.mtto.bitacora_add"
    method="post"
    ref_in="/data/bitacora_add/in"
    ref_out="/data/bitacora_add/out"
    replace="instance"
  />
  <xf:submission
    id="sub_bitacora_con"
    action="soapsm://localhost/axis/services/psm2ws?oper=dario.mtto.bitacora_con"
    method="post"
    ref_in="/data/bitacora_con/in"
    ref_out="/data/bitacora_con/out"
    replace="instance"
  />
  ...
  <xf:submission
    id="sub_tipo_actividad_con"
    action="soapsm://localhost/axis/services/psm2ws?oper=dario.mtto.tipo_actividad_con"
```

```
      method="post"
      ref_in="/data/tipo_actividad_con/in"
      ref_out="/data/tipo_actividad_con/out"
      replace="instance"
    />
  </xf:model>
```

Ahora pasemos a los controles. Dado que hemos venido saturando el texto con la idea de que podemos visualizar a las interfaces propuestas de *SIUX* como un árbol de vistas, y que estamos trabajando con un dialecto de *XML* (donde la arborescencia está implícita); podríamos estar tentados a diseñar la parte que atañe a la lógica de control con una estructura de árbol. Sin embargo, se optó por un diseño más sencillo: el árbol está bien para conceptualizar, pero para especificarlo es más claro y eficiente una simple secuencia de vistas (estados), donde cada una contará con las reglas de transición adecuadas que simulen su ubicación dentro de este árbol hipotético. *XForms* tiene una construcción para permitir la selección de un control entre una lista de opciones: se llama *switch* y tiene, como elementos hijos, nodos llamados *case* (haciendo analogía con las instrucciones condicionales de ciertos lenguajes de programación):

```
<body>
  <xf:switch>
    <xf:case id="init_ctrl" selected="true">
    ...
    </xf:case>
    <xf:case id="mtto_bitacora">
    ...
    </xf:case>
    <xf:case id="bitacora_add">
    ...
    </xf:case>
    <xf:case id="bitacora_con">
    ...
    </xf:case>
    <xf:case id="bitacora_con_detail">
    ...
    </xf:case>
    <xf:case id="bitacora_del">
    ...
    </xf:case>
    <xf:case id="bitacora_del_conf">
    ...
    </xf:case>
    <xf:case id="bitacora_upd">
    ...
    </xf:case>
    <xf:case id="bitacora_upd_out">
    ...
    </xf:case>
  </xf:switch>
```

```
</body>
```

Éste es el esqueleto principal para implementar nuestra lógica de control. El contenido de cada *case* corresponderá con una vista (de hecho, el nombre de las mismas fue puesto como identificador de los nodos). Veamos para qué servirá cada vista, al mismo tiempo que vamos develando los controles que contendrán los elementos *case*. En primer lugar tenemos a *init_ctrl*, que de hecho está marcada como la vista inicial. Aquí pondremos los inicializadores de las secciones de datos, que corresponden a controles de tipo selector (en nuestro caso, únicamente el tipo de las actividades, mismo que aparecerá en las pantallas de altas y modificaciones):

```
<xf:trigger>
  <xf:label>Iniciar interfaz</xf:label>
  <xf:action>
    <xf:dispatch name="xforms-submit" target="sub_tipo_actividad_con"/>
    <xf:toggle case="mtto_bitacora"/>
    <xf:refresh/>
  </xf:action>
</xf:trigger>
```

El elemento *trigger* del espacio de nombres para *XForms*, sirve para abstraer lo que comúnmente pensaríamos como un botón de la interfaz. Dentro del mismo colocamos una etiqueta (*label*) y la acción que deseamos se realice cuando alguien interactúe con el control. En este caso, la acción es compuesta (*action*) y consta de tres subacciones: enviar un evento al elemento *sub_tipo_actividad_con* del modelo, para indicarle que debe invocar al servicio asociado (y con ello actualizar los datos correspondientes al catálogo de tipos de actividad); la siguiente subacción activa otra vista (*toggle*), en este caso nos movemos a la vista con la lista de vistas (el menú de mantenimiento *mtto_bitacora*) y por último pide una actualización de la versión visual de la interfaz, con la subacción *refresh* para que los cambios sean perceptibles al usuario (siempre terminaremos nuestra acciones compuestas con un *refresh*). Por otra parte, el mecanismo para cambiar de vista será siempre con la activación del *case* que la contenga, dentro del único *switch* de nuestra interfaz. Veamos a continuación, el código para nuestra vista menú *mtto_bitacora*:

```
<xf:group css:class="siux_form">
  <xf:group css:class="title">
    <xf:label>Mtto. de bitácora</xf:label>
  </xf:group>
```

```
<xf:group css:class="view-list">
  <xf:trigger>
    <xf:label>Alta de actividad</xf:label>
    <xf:action>
      <xf:toggle case="bitacora_add"/>
      <xf:refresh/>
    </xf:action>
  </xf:trigger>
  <xf:trigger>
    <xf:label>Consulta de bitácora</xf:label>
    <xf:action>
      <xf:toggle case="bitacora_con"/>
      <xf:refresh/>
    </xf:action>
  </xf:trigger>
</xf:group>
</xf:group>
```

Ésta es una especialización del modelo genérico de vista que explicamos hace algunos momentos. Podríamos decir que no tiene la sección de ligas (*links*), ni de campos (*fields*) y que su sección de opciones (*opts*) ha sido reemplazada por esta lista de vistas. Observamos que dentro de la lista (*view-list*) tenemos controles parecidos al inicializador de la sección anterior, con la diferencia de que éstos nos llevan a otras vistas.

Ahora toca el turno a la vista que permite invocar al servicio de altas de actividades de bitácora, llamada en nuestra interfaz *bitacora_add*. En realidad, esta vista no tiene gran cambio en comparación con la mostrada anteriormente, con la excepción de dos detalles: la sección de ligas (*links*) ya no aparece vacía, dado que hay un nivel superior (el menú):

```
<xf:group css:class="links">
  <xf:trigger>
    <xf:label>Regresar</xf:label>
    <xf:action>
      <xf:toggle case="mtto_bitacora"/>
      <xf:refresh/>
    </xf:action>
  </xf:trigger>
</xf:group>
```

Y el control para introducir el tipo (ahora éste toma su valor del catálogo *tipo_actividad*). Recuerde que ya incluimos una sección en el modelo para albergar la respuesta del servicio *tipo_actividad_con*, y que incluimos un control para realizar este llamado (vista *init_ctrl*). De hecho, la llamada a este servicio se haría sin argumentos de entrada (el cascarón para los mismos

está vacío) y con ello nos traeríamos todo el catálogo de tipos de actividad. Sólo falta usar esta información en algún control, para permitir al usuario seleccionar el tipo de una lista predefinida:

```
<xf:select1 ref="/data/bitacora_add/in/in_tipo">
  <xf:label>Tipo</xf:label>
  <xf:itemset nodeset="/data/tipo_actividad_con/out/out_cur/row">
    <xf:label ref="numero"/>
    <xf:value ref="descp"/>
  </xf:itemset>
</xf:select1>
```

En el listado anterior estamos definiendo un control de selección (*select1*), dentro del cual, aparte de la etiqueta, incluímos un elemento *itemset*; éste nos sirve para asociar al control los registros de la lista *out_cur* (parámetro de salida el servicio *tipo_actividad_con*). Obsérvese que estamos indicando cuál de los campos de cada registro fungirá el papel de valor y cuál de etiqueta (el valor se asigna al dato ubicado en */data/bitacora_add/in/in_tipo* y la etiqueta es la mostrada en la lista de opciones del control).

5.8. Invocación a servicios de tipo consulta

Decidimos apartar los comentarios relativos a la invocación del servicio de consulta, por tener más detalles. Recordando el listado de la sección anterior, tenemos a la vista *bitacora_con*, que permitirá invocar al sevicio de consulta. Su definición sería muy parecida a la empleada para *bitacora_add*, con excepción de un elemento nuevo para representar la lista de registros que regresa el servicio asociado. Veamos sólo esta sección:

```
<xf:group css:class="list">
  <xf:group css:class="title">
    <xf:label>Lista de actividades</xf:label>
  </xf:group>
  <xf:repeat
    id="bitacora_con_list"
    nodeset="/data/bitacora_con/out/out_cur/row"
    css:class="items"
  >
    <xf:trigger>
      <xf:label ref="out_cur_numero"/>
      <xf:action>
        <xf:setindex
```

```
          index="count(preceding-sibling::row)+1"
          repeat="bitacora_con_list"
        />
        <xf:toggle case="bitacora_con_detail"/>
        <xf:refresh/>
      </xf:action>
    </xf:trigger>
    <xf:output ref="out_cur_tipo"/>
    <xf:output ref="out_cur_fecha"/>
    <xf:output ref="out_cur_duracion_hrs"/>
  </xf:repeat>
</xf:group>
```

Lo nuevo de esta parte de la vista *bitacora_con* es el control *repeat*; al igual que el control *select1*, éste asocia consigo mismo un conjunto de nodos (la lista de registros que arroja la búsqueda). Pero aquí, el control tendrá un comportamiento de listado, es decir que se pretende que todos los elementos sean mostrados al mismo tiempo. Si recordamos el esquema *XML* para el servicio *bitacora_con*, veremos que los registros van empaquetados en los elementos con etiqueta *row* y que, dentro de éstos, tenemos a los siguientes campos: *out_cur_numero*, *out_cur_tipo*, *out_cur_fecha*, *out_cur_duracion_hrs* y *out_cur_descp*. Con excepción de la descripción, todos estos campos están siendo incluídos en los elementos de la lista a mostrar (estamos asociando una estructura en los datos, con otra estructura en los controles; ambas lineales y homogéneas). El tipo, la fecha y la duración simplemente se despliegan con un control *output* (que ya vimos desde nuestro ejemplo "hola mundo"), pero el número de actividad luce dentro de un control un tanto extraño.

Veamoslo con detenimiento. El número de actividad yace dentro de un botón (bueno, su versión abstracta en *XForms*: *trigger*) y la etiqueta del mismo se toma precisamente del número de actividad - ésta fue otra de las modificaciones que se hizo a *Chiba*. La acción de este control es compuesta, como las que hemos venido presentando; las dos últimas subacciones ya son conocidas (activar una vista y refrescar), así que lo único nuevo es la primera subacción: la instrucción *setindex*. Esta instrucción sirve para asignar al índice de la lista representada por el control *repeat* (*bitacora_con_list*), la posición del elemento que seleccionen en tiempo de ejecución. Para lograrlo, como valor del atributo *index* estamos usando una expresión *XPATH*, la cual simplemente cuenta los nodos precedentes al actual y le suma uno.

A manera de comentario anecdótico, diremos que la instrucción *setindex* no funcionaba bien en el contexto mostrado, en la versión de *Chiba* que usamos. Aparte de agregar la funcionalidad necesaria para corregir el problema, se entró en una pequeña discusión (bueno, plática) con los autores, acerca de si nuestra interpretación del estándar era correcta o no en este punto. Uno de ellos coincidía con nosotros y el otro no; sin embargo al final se dijo que era una parte ambigua de la especificación (sin embargo, el hecho de que la implementación de *Novell* también coincida con nuestra interpretación, nos da cierta tranquilidad al respecto). Entonces, para el autor de estas

líneas, las acciones (como el *setindex*) sí deben evaluarse respecto al contexto circundante, que podría ser dinámico (el nodo actual del *repeat*, en este caso). Pero toda esto fue únicamente para permitir que al presionar el control con el número de actividad, se activara la vista con el detalle (el aprender a realizar las cosas más triviales con nueva tecnología, suele tener sus bemoles ... pero sólo en la primera vez).

Pasando a la vista para el detalle de la lista que vimos apenas, es decir para *bitacora_con_detail*, notaremos los siguientes cambios: la sección de ligas de la vista incluye un nivel más, tenemos campos de salida que apuntan al registro contexto (tomado del resultado de la consulta) y la sección de opciones contiene ahora invocadores para las vistas desde donde podemos lanzar los servicios de baja y actualización - pero antes de activar la vista, se están copiando los parámetros de entrada usando el registro contexto. Veamos las parte relevantes del listado:

```
<xf:case id="bitacora_con_detail">
  <xf:group css:class="siux_form">
    <xf:group css:class="links">
      <xf:select1 ref="/data/ctrl/back/bitacora_con">
        <xf:item>
          <xf:label>Mtto. de bitácora</xf:label>
          <xf:value>mtto_bitacora</xf:value>
        </xf:item>
        <xf:item>
          <xf:label>Consulta de bitácora</xf:label>
          <xf:value>bitacora_con</xf:value>
        </xf:item>
      </xf:select1>
      <xf:trigger>
        <xf:label>Regresar</xf:label>
        <xf:action>
          <xf:toggle case="/data/ctrl/back/bitacora_con"/>
          <xf:refresh/>
        </xf:action>
      </xf:trigger>
    </xf:group>

    <xf:group css:class="title">
      <xf:label>Consulta de bitácora</xf:label>
    </xf:group>

    <xf:group css:class="fields">
      <xf:output
        ref="/data/bitacora_con/out/out_cur/row[xf:index('bitacora_con_list')]/out_cur_numero"
        <xf:label>Número</xf:label>
      </xf:output>
      ...
      <xf:textarea
        css:class="fieldta"
        ref="/data/bitacora_con/out/out_cur/row[xf:index('bitacora_con_list')]/out_cur_descp"
        xhtml:cols="10"
        xhtml:readonly="readonly"
```

```
        xhtml:rows="5"
      >
        <xf:label>Descripción</xf:label>
      </xf:textarea>
    </xf:group>

    <xf:group css:class="opts">
      <xf:trigger>
        <xf:label>Baja de actividad</xf:label>
        <xf:action>
          <xf:setvalue
            ref="/data/bitacora_del/in/in_numero"
            value="/data/bitacora_con/out/out_cur/row[xf:index('bitacora_con_list')]/out_cur_numero"
          />
          <xf:toggle case="bitacora_del"/>
          <xf:refresh/>
        </xf:action>
      </xf:trigger>

      <xf:trigger>
        <xf:label>Actualización de actividad</xf:label>
        <xf:action>
          <xf:setvalue
            ref="/data/bitacora_upd/in/in_numero"
            value="/data/bitacora_con/out/out_cur/row[xf:index('bitacora_con_list')]/out_cur_numero"
          />
          ...
          <xf:setvalue
            ref="/data/bitacora_upd/in/in_descp"
            value="/data/bitacora_con/out/out_cur/row[xf:index('bitacora_con_list')]/out_cur_descp"
          />
          <xf:toggle case="bitacora_upd"/>
          <xf:refresh/>
        </xf:action>
      </xf:trigger>
    </xf:group>
  </xf:group>
</xf:case>
```

Comentemos un poco más sobre los elementos nuevos que aparecen en el recién mostrado código. Mientras que en la sección de ligas no hay nada nuevo que comentar, en la sección de campos vemos que las rutas *XPATH* para acceder a los valores que deseamos se han complicado un poco. Pero sólo un poco, y recordando la estructura de datos que regresa el servicio de consulta quedará aclarado su cambio. Primero tenemos la porción de la ruta que nos ubica justo frente a la lista de registros: */data/bitacora_con/out/out_cur/*. A partir de ese lugar lo que veremos es una lista de registos *row*, cada uno de los cuales puede ser accesado en *XPATH*, con una notación de indexado parecida a *C* (en sus arreglos). Por ejemplo, si quisiéramos ubicarnos en el tercer elemento, completaríamos la ruta así: */data/bitacora_con/out/out_cur/row[3]*.

Sin embargo, como deseamos que puedan ver cualquier registro de la lista, no tenemos un índice fijo; entonces invocamos a una función predefinida (*index*) por el procesador *XForms* (que

forma parte del estándar), y que nos regresa el índice actual del elemento *repeat* que recibe como argumento; en este caso fue *bitacora_con_list*, identificador del elemento usado para mostrar la lista de registros (recordemos que cuando el usuario interactúa con alguno de los registros de esta lista, manualmente asignamos la posición al índice del *repeat*). El último detalle a mencionar sobre la ruta, es que el nombre de la función está prefijado con el espacio de nombres para los elementos de *XForms* (*xf:*). En versiones anteriores de *Chiba*, esto era necesario para que funcionaran las funciones predefinidas de *XForms*.

Cambiando la mirada a la sección de opciones, observamos una lista de controles de activación - abstracción de botones -, que son elementos *trigger* del espacio de nombres *XForm* para este documento. Dentro de cada uno, aparte de la etiqueta y la acostumbrada acción compuesta (una acción formada de varias subacciones), tenemos las instrucciones que manchan la reputación declarativa que hemos venido construyendo para el estándar: *setvalue*. Así es, se trata de la temida instrucción de asignación, sobre la cual no queda otro remedio que usarla únicamente cuando sea necesario; y así lo tratamos de hacer en los ejemplos que realizamos para probar la infraestructura *SIUX* - de hecho, únicamente lo usamos en estas situaciones que estamos a punto de describir. Antes de activar la vista para invocar al servicio de baja o alta, a través de la instrucción *xf:toggle*, asignamos los campos que mostramos una sección arriba en esta misma vista, a la sección de los datos destinada para contener las entradas de los servicios deseados (*/data/bitacora_del/in/* y */data/bitacora_upd/in/*). En el atributo *ref* de la instrucción *xf:setvalue* asignamos el destino y en el atributo *value* ponemos una constante o una ruta que apunta al dato origen. En ambos casos, las rutas son expresadas en el lenguaje estándar *XPATH* (del estilo que explicamos unos párrafos arriba).

Y ya vemos la recursividad virando a la vuelta de la esquina; tan pronto como entramos a estas vistas principales para invocar a otros *WebServices*, los patrones de control que hemos venido relatando se repiten. Es importante recalcar que, de manera simultánea se intentó repasar los aspectos más importantes del estándar, y a la vez mostrar una opción de implementación para un patrón de interfaz de usuario; pero el código mostrado es sólo una manera de lograr el efecto deseado. De hecho, una situación parecida sucede a mayor escala en todo este documento, donde se presentan los elementos de los estándares a través de sencillos ejemplos; sin pretender decir que los pedazos de código propuestos son la única opción.

5.9. Flujo de una invocación en SIUX

Retomando el cuadro completo de la arquitectura que intentamos ilustrar en este ejemplo (mantenimiento de una tabla), apreciaremos que hemos terminado de ilustrar los elementos indis-

pensables que la componen. El diseño de la interfaz de usuario en *XForms* completa la cuarta y última capa. Este acontecimiento merece un pequeño recuento de la interacción que tendrían los componentes, cuando el usuario invocara a un servicio:

Capa 4: El usuario interactúa con la interfaz, que es implementada directamente por el navegador *Web*, a través de la interpretación de una forma *HTML*. Esta forma fue generada dinámicamente y enviada por el procesador *XForms*. De hecho, para el navegador del usuario final no existe nada relacionado con *XForms* ... él sólo ve código *XHTML* y paquetes *HTTP*.

Eventualmente el usuario llena los campos adecuados, digamos del servicio de búsqueda de actividades, y presiona el botón para invocar al servicio. El navegador atrapa el evento, realiza la codificación pertinente y envía el paquete *HTTP* al *URL* que venía indicado en la forma *XHTML*. Dicha dirección está asociada con el procesador de *XForms*, el cual recibe los datos en texto (con ciertas convenciones de codificación que datan desde la época de los *CGIs*); básicamente se trata de una lista de asignaciones, donde las variables representan las porciones de los datos de la interfaz *XForm*, así como representaciones de los eventos que acaban de ocurrir (como la activación del botón de invocación). El adaptador del procesador *XForms* para la *Web* convierte esta información a una representación genérica, independiente de la tecnología final de la interfaz, y la envía a otro módulo (al mismo tiempo que le delega el control del flujo).

En este otro módulo de *Chiba* (nuestra elección para el procesador *XForms*), ya se cuenta con la abstracción de *XML*, así como de los eventos como si estos hubieran ocurrido localmente. Entonces se procesa la acción asociada al control activado, y se ejecutan sus acciones componentes. Eventualmente ello da lugar a la invocación del *WebService*, usando el controlador que creamos como parte de *SIUX* (y que soporta las convenciones requeridas). Dentro de ese controlador yace código que emplea el *API* de la implementación seleccionada para *WebServices* (*AXIS*), y se realiza la invocación al servidor (el *URL* usado para esta tarea es básicamente el mismo que aparece en el modelo de la intefaz *XForm*).

Capa 3: La infraestructura servidor de *Axis*, implementación de *WebServices*, invoca al servicio que creamos para *SIUX*: *psm2ws*. Dicho servicio genérico no es más que una "piel" *XML* para los paquetes que implementamos en la capa 2. Entonces, le llega la petición a través de una documento *XML*, que modela una estructura de datos con los argumentos de entrada para el procedimiento de consulta. La implementación del servicio hace uso del manejador *JDBC* de Oracle, para pedir funcionalidad del manejador de la base de datos; particularmente, para invocar al procedimiento almacenado.

La principal tarea de este componente es una traducción de datos: del *XML* de entrada se extraen los campos tipo cadena que representan los parámetros de entrada del procedimiento, éstos se convierten a los tipos requeridos por su firma y se realiza la invocación; después los parámetros de salida son convertidos nuevamente en *XML*; todo ello respetando el esquema *XML* que conceptualmente asignamos a este servicio. En el caso particular de una búsqueda (o consulta), uno de los parámetros de salida del procedimiento será una referencia al cursor (elemento estándar de *SQL*, que emplean los *SMBD* para representar el resultado de una consulta). También es labor del *WebService* de *SIUX* traducir ese dato estructurado (representado con una clase de *Java*), en la estructura *XML* correspondiente (lo que hemos llamado "la lista de registros"). Es destacada la contribución del estándar *JDBC* en estos menesteres, ya que nos proporciona un manejo independiente del fabricante (mas no del lenguaje), para obtener metainformación de la base de datos (parámetros de los procedimientos, estructura de los mismos, etc.).

Como el servicio recibe en primera instancia cadenas para los parámetros de entrada (todo venía en un documento *XML*), la conversión a los tipos indicados para la invocación al procedimiento ocurre aquí. Sin embargo, dada nuestra intención de centralizar el manejo de errores, las funciones invocadas para este pequeño *análisis léxico sintáctico* son parte de la capa 2.

Capa 2: Entrando a esta capa nos olvidamos de *XML* y nos sumergimos en el viejo mundo de la programación imperativa estructurada, donde todo está codificado en *SQL99-PSM* (en su encarnación llamada *PL/SQL*, producto de la idiosincracia de Oracle). El primero que recibe la petición es el paquete *PL/SQL* genérico, que contiene todos los procedimientos y juega el papel de interfaz de toda la capa. Ahí se invoca a la implementación real del procedimiento, y se atrapa cualquier excepción que pudiera ocurrir (usando las austeras pero funcionales facilidades de *SIUX*).

Cuando llega el control al procedimiento real, que en nuestro caso fue generado automáticamente por las utilerías de *SIUX*, todo lo que vemos es la ejecución de una instrucción *SELECT* de *SQL*, y la asignación del resultado al parámetro de salida (referencia del cursor abierto). Todos los parámetros de entrada se emplearon en la expresión que restringe los registros a devolver.

Capa 1: Los componentes de *SIUX* a este nivel no se activarían en el caso de una consulta, a menos que se tratara de algún error. Definitivamente, la utilidad de esta capa sale a relucir cuando se trata de algún servicio que modifica nuestros datos, ya que es ahí, donde se activan todas las restricciones que implementamos y asociamos a las tablas (sean las implementadas declarativamente con cláusulas o las procedurales). También sería aquí donde entraría la actualización de los campos de auditoría.

5.10. Automatizando la codificación de interfaces

Con el repaso previo de cómo sería la colaboración entre los componentes arquitectónicos de *SIUX*, podríamos retirarnos a descansar, alegando que contamos con todos los elementos para desarrollar aplicaciones usando la propuesta. Pero no podemos darle la espalda al sentido pragmático que ha caracterizado al trabajo, así que observando por segunda vez a la construcción de las interfaces en *XForms*, sale a relucir el tedio que involucra dicha tarea. Tal vez sea menos doloroso el construir una interfaz en *XForms*, que en un lenguaje tradicional para esta labor como podrían ser *Visual Basic* o *Java*; pero dada la cantidad limitada de patrones de control que manejaremos, el *copy-paste* de moldes para instanciarlos con algún caso particular, se antoja aburrido después de unos cuantos intentos.

Pero esa uniformidad del código *XForm* de las interfaces es, precisamente, la motivación que abre la posibilidad para automatizar su generación. Aunque fue delegada al final, por si escaseaba el tiempo, se agregó a la propuesta la especificación e implementación de un pequeño lenguaje para implementar las interfaces, con los patrones que hemos revisado. Lo que nominamos como implementación de este lenguaje es un traductor hacia *XForm*, que se implementó en *XSLT* (por considerar esta situación un caso de uso ideal o clásico para tal lenguaje). De hecho, podríamos decir que la elaboración de ese convertidor o generador de interfaces *XForm*, incluyendo al diseño del lenguaje, fue el componente que demandó más trabajo de toda la arquitectura (aunque no necesariamente el que demandó más tiempo). Otro detalle relevante es que dicha conversión fue la que realmente usó la funcionalidad de la capa 3, que informa los *XMLSchema* para los servicios del *backend* (ya que es aquí donde necesitamos información acerca de los parámetros, como sus nombres, tipos, etiquetas, etc.).

Los requerimientos para este lenguaje intermedio se basaron en las necesidades que impone una sencilla aplicación de mantenimiento de tabla (como el ejemplo que hemos venido ilustrando); y obviamente sólo contempla los patrones de interfaz que hemos mostrado hasta este momento. La idea esencial es permitir la creación de una interfaz para invocar a un *WebService* proporcionando un mínimo de información; pero también facilitar el reuso de los resultados para otras invocaciones. Veamos como especificaríamos nuestra interfaz ejemplo, en este lenguaje intermediario:

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<!DOCTYPE chaca-form SYSTEM "siux-form.dtd">
```

```
<siux-form
  id="dario.mtto"
  label="Mantenimiento de bitácora"
  wsurl="soapsm://localhost/axis/services/dbpkg2ws?oper=dario.mtto."
>
  <wscalls id="mtto_bitacora" label="Mtto. de bitácora">
    <wscall
      oper="bitacora_add"
      conf="no"
      split="no"
      rep="yes"
      refr="yes"
      reset="no"
      label="Alta de bitácora"
    >
      <in>
        <in-par name="in_tipo" mode="sel">
          <wscall-ip oper="tipo_actividad_con">
            <out-ip>
              <out-par-ip name="numero"/>
              <out-par-ip name="descp"/>
            </out-ip>
          </wscall-ip>
        </in-par>
      </in>
    </wscall>

    <wscall oper="bitacora_con" label="Consulta de bitácora">
      <out>
        <out-par-list name="out_cur" label="Lista de actividades">
          <list-view incl="all"/>
            <detail-view>
              <detail-wscalls>
                <wscall
                  oper="bitacora_del"
                  conf="yes"
                  rep="no"
                  label="Baja de actividad"
                />
                <wscall
                  oper="bitacora_upd"
                  split="yes"
                  label="Actualización de actividad"
                >
                  <in>
                    <in-par name="in_tipo" mode="sel">
                      <wscall-ip oper="tipo_actividad_con">
                        <out-ip>
                          <out-par-ip name="numero"/>
                          <out-par-ip name="descp"/>
                        </out-ip>
                      </wscall-ip>
                    </in-par>
                  </in>
                </wscall>
              </detail-wscalls>
            </detail-view>
          </list-view>
        </out-par-list>
```

```
    </out>
   </wscall>
  </wscalls>
</siux-form>
```

Repasemos las secciones del listado anterior, explicando con este ejemplo los componentes del lenguaje. Las primeras líneas contienen el ya familiar indicador de que sigue un documento *XML*, junto con una asociación del mismo con la estructura definida en el archivo *siux-form.dtd*. Ahí es donde definimos la sintaxis del lenguaje intermediario, y dicha definición es usada posteriormente para validar sus instancias, cuando se intente correr la utilería para transformarlas en código *XForm*. Posteriormente observamos que el elemento raíz es un *siux-form*; sólo puede haber uno por documento y lo más relevante de sus atributos es el llamado *wsurl*, contenedor del prefijo para invocar los *WebServices* de la capa 3.

El único elemento hijo de la raíz, es un *wscalls*, que indica la presencia de una lista de vistas. Como lo mencionamos anteriormente, cuando presentamos los patrones para nuestras interfaces, los componentes de esta lista pueden ser invocaciones a servicios del *backend*, o bien, otras listas. En este caso, sólo tenemos invocaciones a *WebServices*, cada una de las cuales queda indicada por un elemento *wscall*. El primer elemento que apreciamos es aquél que representa la interfaz de invocación al servicio *bitacora_add* (procedimiento del paquete *mtto*). No es necesario que estos elementos tengan todos los atributos mostrados (que son una lista exhaustiva), pero lo hicimos así para mencionar el posible uso de cada uno.

oper: Nombre del *WebService* para el cual deseamos crear una interfaz de invocación. Como en realidad sólo determinaremos parte de la funcionalidad de toda la interfaz, le llamaremos simplemente *forma* para evitar confusiones. Retomando nuestra terminología anterior, esta forma pasaría a estar formada por vistas.

conf: Bandera para indicar si la forma tendrá confirmación en la invocación (lo que implicaría la adición de una vista para tal fin).

split: Indica si deseamos partir la vista principal, para dejar en la misma sólo los campos de entrada y enviar las salidas a otra vista (que se activará después de la invocación).

rep: Bandera para indicar si la invocación al servicio es repetible, o si únicamente deseamos habilitarla una sola vez. Obviamente, esta restricción aplica por interacción con la pantalla: una

vez que se realizó la primera invocación y enviamos el control a otra sección de la interfaz, el usario puede volver a invocar a la vista, aunque probablemente con otro juego de datos. Observe que el alta de una actividad no tiene problema en permitir la repetición del servicio, mientras que la baja sí (segundo elemento *wscall* del listado).

refr: Su inclusión fue motivada por el atributo anterior (*rep*) y es otra bandera para decir si necesitamos que, después de la invocación al servicio, se invoque a otro que suponemos actualizará la lista contexto (de registros). Entonces, este atributo sólo tiene sentido para invocaciones a servicios como la actualización y la baja de nuestro ejemplo, donde la lista de registros de la consulta les sirve de contexto (para obtener los datos de entrada y hacer menos tediosa su invocación).

reset: La intención original de este atributo era el poder especificar si deseábamos una limpieza de los controles usados para recibir los parámetros de entrada, una vez que saliéramos de la vista principal de la forma. Actualmente se tienen algunos problemas, ya que aún no podemos reiniciar selectivamente la parte de los datos asociada con la entrada del *WebService* en cuestión, sino que se reinician todos los datos.

Los atributos mencionados son los parámetros que guían gran parte del algoritmo para convertir estas instrucciones en una interfaz *XForm*, pero aún tenemos otros elementos importantes por mencionar. Dentro de un *wscall*, podemos tener una sección para asignar valores predeterminados a las entradas del servicio (*in*), o bien, para restringir qué subconjunto de los parámetros de salida deseamos mostrar (*out*). En el caso de la forma para invocar el servicio de alta y el de actualización, dentro de la sección de parámetros de entrada tenemos una sóla delimitación (elemento *in-par*): el parámetro *in_tipo* tomará sus valores de la lista que regrese un servicio de consulta (*tipo_actividad_con*), lo cual provocará que la forma para invocar estos servicios, incluya un control *xf:select1* (como el mostrado cuando introdujimos la codificación manual de la interfaz). Como puede deducirse, el elemento *wscall-ip* usado para restringir los posibles valores de un parámetro, es un caso particular del elemento más genérico *wscall* (a final de cuentas, ambos representan llamadas a servicios, sólo que el primero no involucra la intervención directa del usuario).

La segunda invocación de la lista *wscalls* corresponde al servicio de consulta *bitacora_con*, mismo que muestra ejemplos de uso para los parámetros de salida. En este caso, estamos indicando que dentro de los parámetros de salida del servicio (elemento *out*), existe un parámetro llamado *out_cur* que consiste en una lista de registros. Esto originará la inclusión de un par de vistas, la lista (*list-view*) que incluirá a todos los campos (*incl=all*) y el detalle (*detail-view*). Este último, aparte de mostrar todos los campos (comportamiento predefinido), tiene una lista de invocaciones

asociada (*detail-wscalls*) que incluye a los servicios de baja (*bitacora_del*) y actualización (*bitacora_upd*). Cada una de estas invocaciones usa recursivamente el elemento *wscall*, originando un comportamiento similar al ya descrito (la creación de la forma para invocar a los servicios mencionados, a través de un conjunto de vistas que interaccionan entre sí).

A final de cuentas, este lenguaje para expresar la especificación de las interfaces de usuario, no es otra cosa que una abreviatura para los patrones que mostramos al presentar *XForms*.

Capítulo 6

Conclusiones

Algo relevante de este trabajo es el cambio en su alcance desde la época en que fue concebido como idea hasta la fecha en que fue finalizada su documentación; inclusive en los mismos capítulos se puede apreciar el cambio: compare ud. el pretencioso estilo de la introducción con el carácter humilde del presente capítulo. Lo que inicialmente pretendía ser "la opción" para desarrollar *SI*, se convirtió en un mero prototipo que sugiere ciertos patrones y tecnología. A través de los capítulos que han pasado se presentaron algunas ventajas de la infraestructura, así que no pretendemos aburrir al lector repitiendo lo mismo. En su lugar, hablaremos de otros aspectos involucrados en este trabajo, como las motivaciones para realizarlo o las deventajas que podría tener.

6.1. Origen de la propuesta

Podríamos simplemente presentar la propuesta de *SIUX* como el resultado de un análisis exhaustivo y concienzudo de todas las opciones existentes, pero la verdad es que hubo otros factores menos "formales" que fueron mucho más contundentes en las decisiones tomadas. Así que, antes de dar las conclusiones de esta propuesta, daremos un breve recorrido por su historia tras bambalinas para contextualizar mejor lo que se hizo y por qué no, lo que se quería hacer en un principio. "SIUX" (que originalmente iba a llamarse *ChacaWeb*, pero se desistió para evitar asociación con cierto detergente), es el resultado de ciertas aspiraciones académicas, de inquietudes con respecto a tecnologías nuevas y de las experiencias del autor en el campo laboral (la cual no va más allá de la cantidad de años que caben en una mano). Todas ellas en su momento se vislumbraron como opciones para hacer una tesis, y al final prevaleció la idea de mezclarlas todas (idea un tanto terca, debemos admitir).

A lo que nos referimos como "aspiraciones académicas" es al paradigma funcional, mismo que ha visto cierta propaganda en las páginas precedentes. La idea original era realizar algún trabajo en

el área de compiladores (conversión de recursividad en iteración), pero se desistió por la cantidad de requisitos necesarios para comprender cabalmente la literatura relacionada (álgebra abstracta, principalmente). Ésta podría ser la vertiente que más se sacrificó en esta tesis, ya que "SIUX" no tiene nada que ver (de manera directa) con tecnología de compiladores para lenguajes funcionales puros; sin embargo, nos consolamos diciendo que la promoción de lenguajes como *SQL* (hablando de las expresiones y consultas), *XML* y sus dialectos (*XMLSchema*, *XSLT* y *XForms*), promueven en cierto sentido la declaratividad como estilo de programación. Y la declaratividad, aunque no siempre implica cero efectos laterales (es decir, que puede haber asignación), sí comparte con el paradigma funcional la preferencia por el alto nivel a la hora de programar (y la delegación de los detalles de bajo nivel a la herramienta).

Lo que bautizamos como "inquietudes tecnológicas" y "experiencia laboral", están estrechamente enlazadas. Las necesidades en el trabajo, especialmente la insatisfacción respecto a las soluciones actuales, hicieron al autor voltear la mirada hacia los estándares y tecnologías emergentes, a fin de buscar alternativas más adecuadas. Desde el primer y único contacto con desarrollo de *SI* para la *Web*, nació la determinación por usar *XML* y desde aquel entonces la evolución en dicho estándar y sus tecnologías hicieron más o menos naturales las elecciones para esta propuesta; de hecho, el empleo de *XML* podría ser la parte menos cuestionable de *SIUX* (sus ventajas han sido aludidas por muchos expertos, desde hace varios años).

Definitivamente, la parte que podría ser tachada de controversial en esta propuesta, es el empleo de *SQL99-DDL* y *SQL99-PSM* como los lenguajes para implementar las reglas del negocio. Y curiosamente, esta elección es la que tiene la historia más interesante detrás. Sucede que, allá por el año 2002, el autor de estas líneas se vió inmerso en un proyecto de migración de tecnología para un *SI* particular. Dicha migración implicaba, entre otras cosas, cambio del *SMBD* (de *Oracle* a *DB2*), así como el cambio del lenguaje para las reglas del negocio (*C* a *Cobol*). Entonces, gran parte de la tarea por hacer inicialmente, consistía en reingeniería a una gran cantidad de código en *C*, para después expresar los algoritmos en alguna forma que pudieran entender sin problema los programadores de *Cobol*.

Inicialmente se pensó en usar documentos con formato libre, donde simplemente se "platicaban" las cosas. Sin embargo, es de todos bien conocida la deficiencia y ambigüedad del lenguaje natural para expresar algoritmos estructurados (anidaciones de condicionales e iteraciones, principalmente), así como para indicar expresiones (aparte de los operadores básicos, hay funciones que se repetían mucho, así que ... ¿cuál biblioteca o lenguaje sería tomado cómo referencia?). La siguiente opción fue crear un pequeño lenguaje de especificación, donde a pesar de permitir el lenguaje natural (como complemento únicamente) se contaba con estructuras de control, expresiones y un juego básico de funciones (principalmente relacionadas con la manipulación de fechas). Al principio los documentos recordaban mucho a los programas originales de *C*, y se incluía una gran cantidad

de factores de bajo nivel que no interesaban a los programadores de *Cobol*. Eran comunes la quejas del estilo "enséñame sólo el SQL", aunque esa petición asumía inconscientemente el empleo de algún medio para comunicar las estructuras de control intercaladas entre las instrucciones *SQL*.

Con el tiempo se ganó más práctica en esas documentaciones, y se logró llegar a un punto donde sólo aparecían ciertas expresiones, estrucutras de control y *SQL*. Las estructuras de control era fáciles de comprender, puesto que son un concepto bastante universal entre los programadores. No así con las expresiones, las cuales desgraciadamente tenían más parecido con el lenguaje *C* (aunado a que en *Cobol*, no se acostumbra mucho usar expresiones). Entonces se decidió que un punto neutral, podría ser el tomar las expresiones del lenguaje *SQL* (el cual se suponía era territorio conocido para todos). Entonces, básicamente nos quedamos con *SQL* más estructuras de control, y se encontró que ello era suficiente para expresar la gran mayoría de los algoritmos del *SI*. Fue precisamente en este punto donde surgió la pregunta: ¿no existirá algún estándar que contenga estos elementos? La respuesta a esta plegaria fue "PSM", parte de la especificación de *SQL99*.

Ése fue el origen de la propuesta para la capa 2, y mencionado de dónde vino, procederemos a mencionar las ventajas y desventajas de esta propuesta (tratando de no repetir los aspectos ya mencionados). En general, en los capítulos previos se tendió a decir sólo el lado bueno de la moneda; ahora toca el turno a los inconvenientes. Pero antes de proceder, también queremos mencionar que ese proyecto de migración de plataforma fue, en algún momento, el centro en torno al cual giraba todo en este trabajo; sin embargo, se consideró que sería más valioso centrarse en aspectos tecnológicos (en lugar de ciertas reglas de negocio), por considerarlos más universales. No obstante, como vestigio de esta línea, se conservaron menciones de este proyecto para fines amenizadores en los párrafos que siguen.

6.2. El dilema de aprovechar SQL

El enviar gran parte de las validaciones estáticas (o invariantes) a la capa 1, libera al resto de las capas de la tediosa y penosa tarea de validar nuevamente ciertas condiciones. Incluso permite identificar y tratar muchos casos de manera declarativa (con expresiones únicamente), así como asociar los mensajes de error en cada caso. Sin embargo, y esto aplica en general para todo lo que tenga que ver con aprovechamiento de *SQL*, se tendrían que librar intensas discusiones (sino batallas) con los administradores de las base de datos (*dbas*). Y es que esto es nada menos que un capítulo más de esa eterna discusión entre la optimización de los programas o su nivel de abstracción. Los *dbas* se quejarán inmediatamente, diciendo que estamos locos si deseamos que cada vez que se intente una operación en las tablas haremos que el *SMBD* ejecute varias validaciones.

Dirán que ello es muy pesado.

Pero justo aquí tendría lugar una discusión interesante. Si nos restringimos a sistemas centralizados (como los que usaban en el proyecto de migración mencionado), surge una pregunta válida: si a final de cuentas alguien tiene que validar esas cuestiones, ¿qué más da dejárselas al *SMBD*?; ¿no ocuparán acaso ambas situaciones, recursos de la misma máquina? De hecho, el dejar en manos de los programas (capa 2) esas validaciones, puede llevar al caso donde se repitan sus evaluaciones (dado que es común la falta de acomplamiento entre los componentes de los sistemas grandes). Sin embargo, todas esas justificaciones pueden ser aplastadas por un hecho contundente: la empresa se encuentra al límite en el uso de sus recursos físicos (*hardware*), y apenas satisfacen la demanda actual. Es obvio que en dichas situaciones, no se puede aplicar el esquema que propone *SIUX*, dado que ello implicaría muy probablemente más recursos (esperando como ganancia mayor facilidad en el desarrollo de los sistemas, así como una mejor integridad de datos).

Este esquema parecería desalentador, pero contrastémoslo con las cantidades millonarias que se invierten en el desarrollo de sistemas en las grandes corporaciones. Si los tiempos de desarrollo se minimizan (con propuestas como *SIUX*), se ahorraría una cantidad de dinero suficiente para reforzar los equipos de cómputo. En otras palabras, los grandes gastos en sistemas actualmente yacen en el pago de los empleados, no en el *hardware* y la necedad en seguir ciertos esquemas obliga a los desarrolladores a perder gran parte del tiempo de sus proyectos, buscando maneras de adaptarse al entorno y las limitantes impuestas. Así, delegar las validaciones al *SMBD* podría requerir más recursos de *hardware*, pero ello probablemente no sería mucho más caro que la inversión de tiempo necesaria en adaptar los programas a la restricción de usar el *SMBD* lo menos posible.

Esa confrontación con los *dbas* se extiende a todos los terrenos donde se invoque al *SMBD*, es decir, la capa 2 de nuestra arquitectura. El simple hecho de saber que los programas en *PSM* correrán "dentro de la BD", llevará a los *dbas* a pensarlos como entes ineficientes por naturaleza. En realidad, son los patrones de uso de *SQL* lo que hacen ineficientes a los programas, independientemente del lenguaje empleado. Lo que sí podría ser polémico es la decisión entre dejar los cálculos a una instrucción de consulta (para que los datos salgan ya transformados) o delegar dichas evaluaciones a las instrucciones que iteran sobre los resultados de la búsqueda (que tendría que limitarse a traer la materia prima). Aquí regresamos a la misma pregunta: ¿no será acaso la misma máquina, la responsable de ejecutar las evaluaciones en ambos casos?, ¿no estamos acaso desperdiciando a *SQL*, limitando nuestras consultas a instrucciones básicas?

El autor tiene la impresión de que la mera apariencia sintáctica lleva a muchas personas a juzgar la complejidad computacional de una consulta *SQL*. Simpleza sintáctica no necesariamente implica eficiencia. No estamos diciendo que se deba usar a *SQL* sin ningún tipo de precaución; de

hecho, no consideramos tan trivial emplearlo de manera eficiente (se requiera tanto teoría como mucho práctica). Pero el aprovechar las características de *SQL* podría incluso optimizar seriamente los procesos; reemplazando por ejemplo, cientos o miles de consultas sencillas por una gran instrucción que traiga de una vez todo lo que necesitamos y que moleste mucho menos al *SMBD*. Sin embargo, estas ventajas podrían salir a relucir únicamente en ciertas configuraciones y plataformas. Por ejemplo, en ambientes *mainframe* (comúnmente *OS390* de *IBM*), todo parece estar diseñado para procesar grandes volúmenes de información (desde los procesadores hasta los programadores), principalmente en los programas que cierran la jornada diaria (*batch*). Pero hay un "detalle", y es que se promueve el trabajo con los datos a través de archivos, en lugar de acceder directamente las tablas (argumentando que ello es mucho más eficiente).

Sin embargo, como ejemplo contrastante, mencionaremos el caso del proceso de contabilidad del proyecto de migración que hemos referido anteriormente. Para fines didácticos (que lo entendiera fácilmente el programador de *Cobol*), dicho proceso fue traducido en su totalidad a una sóla instrucción de consulta *SQL*. Desde luego era una instrucción complicada, con varios niveles de anidamiento, agrupaciones, expresiones condicionales, etc.; es lo que podríamos llamar un caso extreno del lema "aprovechemos SQL", dado que absolutamente todo era hecho en ese lenguaje (incluyendo el formateo de los datos en su versión final, que se vaciaba en un archivo). El asunto es que, para fines de validación del análisis, se realizaron varias pruebas con esa consulta *SQL*; encontrando que en promedio demoraba entre 3 y 5 minutos. Una cantidad nada despreciable, si consideramos que se hacía en una máquina con carga productiva, con datos reales y masivos y que el programa original se tardaba de 10 a 15 veces más tiempo.

6.3. UNIX vs OS390

Sacamos a propósito esta comparación de plataformas, porque es justo hacerle justicia a las dos (*OS390* vs *Unix*), y porque la propuesta de *SIUX* es una especie de fusión entre ambas. Pero no en el sentido tecnológico, sino en las costumbres o patrones de uso que emplean los desarrolladores de cada plataforma. Este proyecto de migración fue una experiencia bastante interesante, en el sentido de que fue posible la peculiar situación de que los desarrolladores de ambas plataformas conocieran lo que se hacía del "otro lado". Para el autor de este trabajo no fueron del todo extraño las prácticas que se encontraron en los miles de líneas de código analizadas para *C-Unix*, sin embargo, los que sí solían sorprenderse al ser expuestos a estas situaciones, eran los programadores y diseñadores de *Cobol*. En general, podríamos decir que los desarrolladores de *Unix*, cuyos códigos se analizaron, tienen una especie de "ansiedad académica" por mostrar al mundo que aprendieron cantidad de cosas en la escuela, que conocen varias estructuras de datos, que saben tantos lenguajes de programación, etc. Juntando esto con las facilidades y flexibilidad que proporcionan

los sistemas operativos tipo *Unix*, nos topamos con una cierta promiscuidad barroca, es decir, que se mataron moscas a cañonazos en muchos casos ... se concibió un sistema mucho más genérico y complejo de lo necesario.

El haber leído todos esos programas, evidenció en un servidor ese tipo de vicios. Supongo que muchos de ellos se originan de un hecho que puede resultar decepcionante, para los que seleccionan el desarrollo de *SI* como área de trabajo: no son necesarias, de manera directa, gran parte de las cosas que uno aprende en las carreras de ciencias computacionales. El mero orgullo propio de saber hacer tal o cual cosa, no debería justificar la complejidad agregada a los sistemas por forzar esa aplicación de conocimientos. Si nos interesan los algoritmos y las estructuras de datos complicados y variados, probablemente los sistemas de información (en general), no sea el lugar para quedarnos. En ellos existe complejidad, sí, pero es de otro tipo al que uno espera viniendo de la academia (en especial de formación científica). De hecho, desarrollarse en esta área suele traducirse en ir abandonando paulatinamente las cuestiones técnicas y concentrarse en las reglas del negocio, así como en habilidades para manejo de personal, toma de decisiones, cuestiones administrativas, etc. No decimos que esa orientación sea mala (de hecho, implica más dinero y prestigio social), pero ¿acaso es el único camino a seguir? En la realidad empresarial de nuestro país, la respuesta parece ser afirmativa.

Retomando las diferencias entre los desarrolladores de ambas plataformas, se percibió una situación muy distinta con los desarrolladores de la plataforma *mainframe*. En ella, gran parte de los programadores no habían estudiado una carrera totalmente orientada a las ciencias computacionales, sino más bien ingenierías (y no necesariamente en computación). El resultado es que muchos de ellos aprendieron a programar fuera de la escuela y por lo tanto su preparación fue más bien empírica. Ello les dió un sentido muy práctico de las cosas, y aunado a las limitaciones del lenguaje *Cobol* (no es precisamente flexible, comparado con *C* o *Java*), daba por resultado soluciones a los problemas, radicalmente distintas a las encontradas en el lado *Unix*. De las principales ventajas, respecto a la plataforma *Unix*, se captó el evitar intermediarios entre la información de las tablas y los procesos, ya que no se usaban en ningún lugar estructuras de datos innecesariamente. Sin embargo, como lado negativo vemos la falta de costumbre para emplear expresiones (algunos cálculos recordaban código en ensamblador), y más grave aún, la escasez de modularidad (los subprogramas no eran muy famosos aquí). Un punto donde definitivamente ganaba esta plataforma sobre *Unix*, pero más por la gente que por la tecnología, era en los mecanismos de control que tenían que pasar los programas para subir a producción: el control de calidad se tomaba más en serio, y como consecuencia, los programadores tenían más "respeto" por las instrucciones *SQL* que usaban (nadie quería que le rechazaran su programa por violar los lineamientos ... aunque varios de ellos eran criticables).

Entonces, si miramos a *SIUX* (la parte del *backend*) desde este contexto, veremos que es un

intento por tomar cosas buenas de ambos mundos: por un lado, acepta que no necesitamos (en la mayoría de los casos) de otra estructura de datos que la misma *BD* y por otra, promueve la modularidad a través de sus diversos componentes y capas. Así que la recomendación para aplicar un esquema como *SIUX* en las primeras capas, sólo podría aplicar en un ambiente de trabajo donde las reglas del juego se pudieran discutir y evaluar abiertamente, es decir, que el principal factor de bloqueo para su aplicación sería el humano. Comenzando por la relativa novedad o popularidad de la tecnología, no resultaría poco probable que el desconocimiento se tradujera en desconfianza. También se necesitaría de flexibilidad para administrar las bases de datos de desarrollo, puesto que se requeriría de libertad para crear toda una gama de objetos en ellas (tablas, vistas, restricciones, procedimientos, etc.). No sería nada práctico tener que pedir la ayuda de soporte para estas cuestiones.

Pero aparte de estas cuestiones que implican meros cambios de hábitos (programadores que también sepan soporte, y jefes de área con mayor contexto tecnológico), también hay cuestiones técnicas que podrían dificultar la aplicación de nuestra propuesta prototipo. La relativa juventud del lenguaje *PSM* provoca que existan aún muy pocas herramientas con entornos de desarrollo y depuración, comparado con las opciones tradicionales (que incluso cuentan tanto con versiones comerciales, como de código abierto). La limitante del matrimonio con cierta marca de *SMBD* es otro factor que se ha mencionado en este documento; pero la experiencia dicta que en muchos casos, este matrimonio de todas formas se efectúa (aunque por razones que rebasan el ámbito técnico). Así que, si de todas formas nos vamos a pasar los próximos 10 o 20 años con un manejador, tratemos de aprovechar sus características estándar (ello podría inclusive facilitar una futura migración).

6.4. SQL:2003 y PostgreSQL

En capítulos previos se mencionó la aparición de la versión 2003 del estándar *SQL* (*SQL:2003*), lo cual evidencía una característica imposible de ignorar en el área de la computación: todo está en evolución contínua y es necesario mantenerse actualizado. Por lo tanto, a un año de haber comenzado este trabajo ya se vislumbran opciones para mejorarlo.

Como fue mencionado, la aparición de *SQL:2003* no le quita mérito a la propuesta, por el contrario, sugiere el aprovechamiento de funcionalidad que podría hacerla más viable. Si tuviésemos un *SMBD* que soportase la nueva versión del estándar, no dudaríamos mucho en hacer los siguientes cambios:

1. Las columnas numéricas que representan llaves primarias, se podrían soportar directa-

mente en lugar de crear por separado secuencias y manipularlas manualmente. Característica que por cierto, existe desde hace varios años en el *SMBD PostgreSQL* (y que no era usada, en muchos casos, por temor a incompatibilidad con el estándar).

2. Las columnas derivadas ahora podrían ser soportadas directamente por el manejador, en lugar de simular dicha funcionalidad con procedimientos y disparadores (*triggers*).

3. *SQL:2003* introduce un nuevo tipo de expresión llamada *retrospectivamente determinística*, y que puede aparecer en varios contextos (como la definición de restricciones). Gracias a ello, ahora se podrían usar directamente en las expresiones de las restricciones *CHECK*, invocaciones a funciones como *CURRENT_TIME*, *CURRENT_DATE* o *CURRENT_TIMESTAMP*. Gracias a esta facilidad, nos podríamos ahorrar la codificación de procedimientos para estas validaciones.

4. La incorporación oficial de *XML* al estándar *SQL*, nos tentaría nuevamente a dar el paso *cuántico* en esta propuesta: extenderla para manejar datos con estructura arbitraria. Aún cuando no usemos columnas con tipos estructurados (como aquí se ha propuesto), las entradas y salidas de los *WebServices* que ofrece el *backend* podrían ser extentidas para soportar tipos *XMLSchema* arbitrarios.

Aunque la propuesta *SIUX* maneja un modelo relacional de datos, donde los tipos para las columnas de las tablas no pueden ser estructurados; es común que se requieran *vistas* con estructura de la información. Piense usted por ejemplo, en el estado de cuenta que le llega de su banco: aunque impreso en papel puede ser modelado con una estructura de datos de profundidad considerable (datos del cliente, movimientos, saldos, etc). Sin embargo, es muy probable que esa estructura haya sido construida por programas, a partir de tablas con columnas de tipos simples.

Gracias a la incorporación de *SQLX* al estándar *SQL*, ahora tendríamos una forma de hacer esa transformación con unas pocas consultas (en algunos casos, inclusive con una sola); la reducción en los tiempos de desarrollo para este tipo de servicios sería muy considerable (aunque habría que tomar en cuenta factores de desempeño). La generación automática de interfaces de usuario para esos datos, podría ser no tan trivial; pero sigue siendo atractivo que por lo menos el *backend*, pueda comunicar este tipo de información a otros sistemas.

Por otro lado, la revisión de las nuevas características del estándar *SQL*, nos llevó a hojear otras funcionalidades presentes en el estándar desde versiones anteriores, y que podrían resultar

últiles para este trabajo. Estamos hablando, en primer término, de los *dominios* (*domain constraints*) que pueden ser pensados como un método para crear nuevos tipos. Sólo que la peculiaridad de los *dominios* radica en que no definen nueva estructura, ni funcionalidad para los datos; sino restricciones. A través de ellos, podríamos factorizar considerablemente el código *SQL* para crear las tablas, con columnas que requieran de validaciones similares. Esta facilidad aparece em *PostgreSQL* desde hace varias versiones, aunque no se encontró en la documentación de *Oracle9i*.

En segundo término queremos mencionar las *aserciones* (*assertion constraints*), que podemos pensar como las restricciones *CHECK* pero sin las limitaciones de éstas (pueden contener consultas) y sin asociación con alguna tabla en particular. Esta funcionalidad presente en el estándar, debo admitirlo, resultaba desconocida por no estar implementada en los manejadores empleados (*Oracle*, *PostgreSQL*, *DB2*). Sin embargo, su implementación sería de gran interés para la propuesta, ya que permitiría evitar gran parte de la codificación de los procedimientos de validación (necesarios por las limitantes de las claúsulas *CHECK*). A ello habría que agregar, que estaríamos promoviendo aún más el paradigma funcional y declarativo. No obstante, la falta de soporte para estas restricciones podría sugerir que su implementación no es trivial para los fabricantes de *SMBD*.

Pasando al tema del software libre, nos vemos obligados a mencionar nuevamente que *PostgreSQL* era la primera opción para este trabajo. El factor decisivo que llevó a preferir *Oracle* fue la falta de soporte para las excepciones en *PostgreSQL* (mismas que liberan al código de la segunda capa, de la tediosa verificación de muchos errores). Pero hay buenas noticias en este sentido, no hace mucho fue liberada la versión beta de *PostgreSQL 8*; misma que rellena este gran vacío.

Con este antecedente, podríamos replantearnos la migración hacia *PostgreSQL* como un evento, no del todo imposible ... aunque con algunos detalles. La principal limitante técnica que existía (excepciones) ya no estaría presente, pero seguiríamos teniendo incomodidades con el manejo de errores y los parámetros de salida. En *PostgreSQL* todavía no existe un control muy fino sobre el *SQLCODE* a regresar al cliente, así que, a pesar de poder usar los mensajes de error de nuestros catálogos internos, estaríamos desperdiciando los códigos numéricos.

Otro problema sería, el hecho de que en *PostgreSQL* no existen los procedimientos como tales, sino que únicamente tenemos funciones. Esto afectaría considerablemente la tarea de migración de *SIUX*, ya que habría que empaquetar los parámetros de salida de los procedimientos, en tipos estructurados que sirvieran como parámetros de regreso de las funciones. Adicionalmente, tendríamos la situación de que los parámetros de entrada/salida en *Oracle PL/SQL*, no tendrían un equivalente directo. De hecho, nos veríamos obligados a rediseñar todas aquellas partes que usaran procedimientos que modifican sus entradas, por versiones puramente funcionales. En teoría suena interesante el cambio (por aquello de promover el paradigma funcional), pero podría resultar poco práctico. Lenguajes puramente funcionales nos piden purificarnos de los efectos laterales

(asignaciones), a cambio de una serie de facilidades: inferencia automática de tipos (no declaraciones), tuplas y listas dinámicas, funciones como valores de primer orden, etc. Sin estas facilidades, sería un tanto tedioso usar únicamente funciones en un lenguaje imperativo como *PLPGSQL de PostgreSQL* ... aunque no imposible.

6.5. Unas palabras finales

El empleo de los estándares *XML* no necesita mucha ayuda en cuanto a su justificación. Basta dar un pequeño vistazo a la cantidad de trabajos, artículos, productos, y evangelización (promoción) de esta tecnología. Pero esto sería desde el punto de vista global; si nos restringimos a nuestro país veremos que las cosas cambian. Otra búsqueda, pero a la sección de empleos relacionados con tecnología de la información, nos dejará ver que nombres como *WebServices* o *XForms* aún son bastante escasos. Entonces, aquí la limitante para una aplicación de *SIUX* en las capas 3 y 4, sería que requerimos como precondición, personal que conozca estas tecnologías. Desgraciadamente, muchas personas están acostumbradas a preferir los productos terminados de código cerrado que, si bien ahorran parte del trabajo, suelen no respetar los estándares internacionales; limitando así la compatibilidad y flexibilidad de los sistemas. En los años de experiencia laboral que se tienen, la única ocasión que se escuchó mencionar la palabra *WebServices* en una empresa, fue porque un cliente extranjero requería de dicho estándar. Triste situación, pero aquí únicamente el tiempo hará justicia a los dialectos *XML* que propone *SIUX*. Inclusive, el caso de *XForms* podría ser todavía más dramático, dado que el estándar es tan nuevo que las implementaciones aún están en proceso de estabilización.

Hablando de las interfaces de usuario, la parte que más preocuparía es la decoración de las vistas (usando *CSS*). Aunque en este trabajo se hicieron varios experimentos al respecto, no basta con cuestiones técnicas para producir interfaces "agradables", sino que también se requiere cierta habilidad estética. Y éste sería el nicho natural de los diseñadores gráficos, que actualmente ya están familiarizados en la creación de páginas para internet. Sólo habrá que esperar un poco más para que las herramientas que manejan convivan mejor con el estántar *CSS* - el caso ideal sería alguien que conociera la parte técnica de *XHTML* y *CSS*, y además tuviera nociones de diseño; pero no creemos que ese tipo de combinación abunde en el mercado.

Con el comentario previo abrimos paso para una de las conclusiones finales: *SIUX* parece ser una propuesta un tanto adelantada a su tiempo. Se necesita que los estándares y la cultura que implica su uso, permeen en los distintos grupos que están involucrados en el desarrollo de sistemas. De hecho, por eso le dimos el adjetivo de "prototipo" a la propuesta, en lugar de presentarlo como un producto listo para consumirse en cualquier ámbito. Evidentemente se trata de

un experimento que requeriría de más seguimiento antes de dar una respuesta más contundente. Empero, cumplió su objetivo inmediato: saciar la curiosidad del autor de estas líneas, respecto a la factibilidad de la propuesta para implementar porciones de *un* sistema de la vida real. Y en ese sentido, podemos decir que *SIUX* pasó la prueba. De especial interés era que las reglas del negocio realmente pudieran expresarse en puro *SQL* (sean expresiones, sean procedimientos) y que los estándares tan nuevos pudieran acoplarse (aquí jugó un papel muy importante usar productos de código abierto). También fue gratificante observar que gran parte del sistema de la vida real, pudo expresarse en términos de los patrones que presenta un problema de mantenimiento de tabla (como el mostrado a lo largo de este documento). De hecho, se sacrificó incluir esos ejemplos en aras de no desviar la atención de la naturaleza tecnológica de la propuesta (las reglas del negocio podrían distraer de este objetivo, aparte de que no eran realmente casos generales que pudieran interesar a un amplio público).

Entonces, podemos decir que, asumiendo un contexto donde las políticas y personal de la empresa lo permita, un esquema como *SIUX* podría ser aplicado en un futuro no muy lejano, aunque si deseamos la inmediatez, lo más probable es que tuviéramos que aplicar sólo la propuesta a una o dos de las capas, lo cual tampoco suena del todo mal. Y no podríamos finalizar este trabajo sin hacer una observación que consideramos crucial: aunque la tecnología juegue un papel importante en el desarrollo de los *SI*, es necesario percatarse que sólo suele ser un aliciente para concentrarse en los detalles importantes, mas no un factor que garantice buenos sistemas.

Propuestas como *SIUX* con estándares y tecnología novedosos, únicamente prometen ahorrarle al desarrollador momentos de tedio, dado que las tareas mecánicas podrán ser evitadas en muchos casos. Pero con ello se pretende que nos concentremos más en las partes cruciales como el análisis o el diseño. No es raro, pues, querer tapar el sol con un dedo pensando que con los mismos desarrolladores y prácticas, la inclusión de cierta tecnología arreglará de manera automática los problemas. El hecho es que, haciendo a un lado los requerimientos de comunicación entre sistemas (que podrían reclamar ciertos estándares), únicamente con buenas bases se pueden desarrollar sistemas satisfactorios, inclusive con la tecnología de hace 20 años. Lo cual nos debería llevar, a señalar acusatoriamente deficiencias en áreas más básicas, aunque probablemente en ello peligre el orgullo de mucha gente ... pero la humildad debería ser parte de nuestro curriculum.

Referencias

[1] Software Engineering
Ian Sommerville, Adison-Wesley

[2] Software Architecture in Practice
Len Bass, Paul Clements, Rick Kazman, Adison-Wesley

[3] Why Functional Programming Matters
John Hughes, The Computer Journal, Volume 32 , Issue 2 (April 1989)

[4] Software Reuse: Architecture, Process and Organization for Business Success
Ivar Jacobson, Martin Griss, Patrik Jonsson, Adison-Wesley-Longman

[5] Information Modeling and Relational Databases
Terry Halpin, Morgan Kaufmann

[6] Data on the Web, from Relations to semistructured data and XML
Serge Abiteboul, Peter Buneman, Dan Suciu, Morgan Kaufmann publishers

[7] The Craft of Functional Programming
Simon Thompson, Adison-Wesley

[8] Database Language SQL – Part 2: Foundation (SQL/Foundation)
ANSI/ISO/IEC (9075-2), Sep 1999,
http://www.ncb.ernet.in/education/modules/dbms/SQL99/ansi-iso-9075-2-1999.pdf

[9] Database Language SQL – Part 4: Persistent Stored Modules (SQL/PSM)
ANSI/ISO/IEC (9075-4), Sep 1999,
http://www.ncb.ernet.in/education/modules/dbms/SQL99/ansi-iso-9075-4-1999.pdf

[10] Information technology – Database languages – SQL –
Part 14: XML-Related Specifications (SQL/XML) [SQLX]
WD ISO/IEC 9075-14:2007 (E), 29 Dic 2003,
http://www.sqlx.org/SQL-XML-documents/5WD-14-XML-2003-12.pdf

[11] SQL Bible (SQL99)
Alex Kriegel and Boris M. Trukhnov
Wiley Publishing Inc., 2003.

[12] Overview of SQL:2003
Krishna Kulkarni, IBM Corporation-Silicon Valley Laboratory, 06 Nov 2003,
http://www.wiscorp.com/sql/SQL2003Features.pdf

[13] SQL:2003 has been published
Andrew Eisenberg IBM, Westford, MA
Jim Melton Oracle Corp., Sandy, UT
Krishna Kulkarni IBM, San Jose, CA
Jan-Eike Michels IBM, San Jose, CA
Fred Zemke Oracle Corp., Redwood Shores, CA
ACM Press New York, NY, USA 2004

[14] Oracle 9i Database Administrator's Guide, Release 2 (9.2)
Oracle Corporation, Mar 2002, Part No. A96521-01
http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96521/toc.htm

[15] Oracle9i Application Developer's Guide - Fundamentals, Release 2 (9.2)
Oracle Corporation, Mar 2002, Part No. A96590-01
http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96590/toc.htm

[16] Oracle9i SQL Reference, Release 2 (9.2)
Oracle Corporation, Oct 2002, Part No. A96540-02
http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96540/toc.htm

[17] PL/SQL User's Guide and Reference, Release 2 (9.2)
Oracle Corporation, Mar 2002, Part No. A96624-01
http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96624/toc.htm

[18] Oracle9i JDBC Developer's Guide and Reference, Release 2 (9.2)
Oracle Corporation, Mar 2002, Part No. A96654-01
http://download-west.oracle.com/docs/cd/B10501_01/java.920/a96654/toc.htm

[19] Oracle9i XML Database Developer's Guide - Oracle XML DB
Oracle Corporation, Oct 2002, Part No. A96620-02
http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96620/toc.htm

[20] Oracle9i Application Developer's Guide - Object-Relational, Release 2 (9.2)
Oracle Corporation, Mar 2002, Part No. A96594-01
http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96594/toc.htm

[21] Extensible Markup Language (XML) 1.0 (Third Edition)
W3C Recommendation, 04 Feb 2004,
http://www.w3.org/TR/2004/REC-xml-20040204

[22] Namespaces in XML
World Wide Web Consortium, 14 Ene 1999,
http://www.w3.org/TR/1999/REC-xml-names-19990114

[23] URIs, URLs, and URNs: Clarifications and Recommendations 1.0
Report from the joint W3C/IETF URI Planning Interest Group
W3C Note, 21 Sep 2001,
http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921

[24] XML Schema Part 0: Primer
W3C Recommendation, 2 May 2001,
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

[25] XML Schema Part 1: Structures
W3C Recommendation, 2 May 2001,
http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[26] XML Schema Part 2: Datatypes
W3C Recommendation, 2 May 2001,
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[27] XML Path Language (XPath) Version 1.0
W3C Recommendation, 16 Nov 1999,
http://www.w3.org/TR/1999/REC-xpath-19991116

[28] XSL Transformations (XSLT) Version 1.0
W3C Recommendation, 16 Nov 1999,
http://www.w3.org/TR/1999/REC-xslt-19991116

[29] Web Services Architecture
W3C Working Group Note, 11 Feb 2004,
http://www.w3.org/TR/2004/NOTE-ws-arch-20040211

[30] SOAP Version 1.2 Part 0: Primer
W3C Recommendation, 24 Jun 2003,
http://www.w3.org/TR/2003/REC-soap12-part0-20030624

[31] SOAP Version 1.2 Part 1: Messaging Framework
W3C Recommendation, 24 Jun 2003,
http://www.w3.org/TR/2003/REC-soap12-part1-20030624

[32] SOAP Version 1.2 Part 2: Adjuncts
W3C Recommendation, 24 Jun 2003,
http://www.w3.org/TR/2003/REC-soap12-part2-20030624

[33] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language
W3C Working Draft, 03 Ago 2004,
http://www.w3.org/TR/2004/WD-wsdl20-20040803

[34] Web Services Description Language (WSDL) Version 2.0 Part 2: Predefined
W3C Working Draft, 3 Ago 2004,
http://www.w3.org/TR/2004/WD-wsdl20-extensions-20040803

[35] Web Services Description Language (WSDL) Version 2.0 Part 3: Bindings
W3C Working Draft, 3 Ago 2004,
http://www.w3.org/TR/2004/WD-wsdl20-bindings-20040803

[36] Axis User's Guide, Version 1.1
ASF, The Apache WebServices Project, The Apache Axis Project,
http://ws.apache.org/axis/java/user-guide.html

[37] HTML 4.01 Specification
W3C Recommendation, 24 Dic 1999,
http://www.w3.org/TR/1999/REC-html401-19991224

[38] Cascading Style Sheets, level 2 revision 1
CSS 2.1 Specification
W3C Candidate Recommendation, 25 February 2004,
http://www.w3.org/TR/2004/CR-CSS21-20040225

[39] XHTML(TM) 1.0 The Extensible HyperText Markup Language (Second Edition)
A Reformulation of HTML 4 in XML 1.0,
W3C Recommendation, 26 Ene 2000, revisión 1 Ago 2002,
http://www.w3.org/TR/2002/REC-xhtml1-20020801

[40] XForms 1.0
W3C Recommendation, 14 Oct 2003,
http://www.w3.org/TR/2003/REC-xforms-20031014

[41] XForms 1.1 Requirements
W3C Working Group Note, 31 Ago 2004,
http://www.w3.org/TR/2004/NOTE-xforms-11-req-20040831

[42] Chiba Cookbook
Joern Turner, joern.turner@chibacon.de, 07 Jun 2004,
http://chiba.sourceforge.net/ChibaCookBook.pdf