UNIVERSIDAD NACIONAL DE SANTIAGO DEL ESTERO

FACULTAD DE CIENCIAS EXACTAS Y TECNOLOGÍAS

LICENCIATURA EN SISTEMAS DE INFORMACIÓN

TRABAJO FINAL DE GRADUACIÓN

JESSLET: JESS COMO UN SERVICIO

Autor:

RICARDO GUSTAVO MIRANDA

Profesor Guía:

DRA. ROSANNA COSTAGUTA

Asesores:

PROF. ALDO ROLDÁN

DRA. FLORENCIA CORONEL

Universidad Nacional de Santiago del Estero
Facultad de Ciencias Exactas y Tecnologías

Trabajo Final de Graduación de la Licenciatura en Sistemas de Información

JESSLET: JESS COMO UN SERVICIO

Autor:

Ricardo Gustavo Miranda

Profesor Guía:

Dra. Rosanna Costaguta

Asesores:

PU Aldo Roldán

Dra. Florencia Coronel

Aprobado del día _____ del mes _____ del año 20___

por el Tribunal integrado por

Firma	Firma	Firma
Aclaración	Aclaración	Aclaración

Santiago del Estero – Argentina

A mi familia. Lo bastante tolerantes. Desmesuradamente insistentes.

Ricardo Gustavo Miranda

Agradecimientos

A mi directora del Trabajo Final, la ing. Rosanna, por haber dedicado gran parte de su tiempo y salud en el desarrollo.

A mis compañeros Germán, Ricardo, Pablo, Silvia y Mario, por su obstinación y ayuda brindada cuando la necesitaba.

Al profesor Aldo, por sus consejos, orientación, y persistencia. Siempre oportunos y necesarios en tiempos de incertidumbre.

A mi esposa e hijos, sencillamente por formar la esencia vital que lo lleva a uno a aguantar un minuto más. Sin ellos, la vida no sería tan caótica, impredecible e interesante.

Índice de contenido

Resumen

Este proyecto planteó el desarrollo de una **interfaz** orientada a servicios (o **servicio web**), para el *shell* ***Jess***, mediante la cual sea posible la creación, administración y ejecución de *sistemas expertos basados en reglas de inferencias*. El objetivo principal de este software es el de mejorar la forma en la que la funcionalidad de estos sistemas inteligentes puede ser implementada en otros sistemas y aplicaciones. A partir de este objetivo, fueron planteadas las hipótesis, las cuales enuncian que esta incorporación de un sistema experto creado sobre el *shell Jess*, a través del servicio web, repercute con una mejora en las métricas de **cohesión** y **acoplamiento**,

El desarrollo del servicio web, denominado **Jesslet**, se realizó siguiendo las recomendaciones propuestas por la metodología ágil *Programación Extrema*. Para poder probar el funcionamiento de este nuevo software, se desarrolló un sistema experto de ejemplo para la advertencia de aplicación de vacunas, el cual sirve para indicar qué vacunas deben aplicarse en una fecha determinada, basándose en la edad y el historial de inmunizaciones de una persona. A su vez, para demostrar el funcionamiento del Jesslet desde múltiples plataformas, se programaron dos aplicaciones clientes: una sobre *Python*, y la otra sobre *Javascript* como una extensión del navegador web *Google Chrome*.

La contrastación de las hipótesis planteadas se hizo mediante pruebas específicas para tal fin, obteniéndose mejoras sustanciales en la integración de las capacidades inteligentes de un sistema experto sobre otros sistemas. Puntualmente, se observó una notable reducción en el acoplamiento y en la dependencia de componentes necesarios para la incorporación de la funcionalidad, aunque, por otro lado, resultaron evidentes las limitaciones que el *Jesslet* impone a las capacidades del *shell Jess*.

Palabras Claves: Sistema Experto, Shell, Jess, Servicio Web

Capítulo I Formulación del Problema y Planteo de una Solución

Un ***shell*** es un sistema experto carente de su base de conocimientos, provisto de un *motor de inferencias*, interfaz de usuario, y un *trazador de explicaciones*. El objetivo de un *shell* es la creación de sistemas expertos, mediante la carga de conocimientos, con la capacidad de *"razonar"*. **Jess** es un *shell* y un *entorno de programación* escrito en lenguaje *Java*, para el desarrollo de sistemas expertos basados en reglas de inferencia [FRIEDMAN2003, FRIEDMAN2008].

Como consecuencia de estar escrito sobre Java, el *shell* Jess es susceptible de ser implementado y distribuido sobre una variedad de plataformas, desde computadoras personales hasta tecnología móvil. Justamente, una de las cualidades más valoradas que posee Java es su capacidad de ser ejecutado en diferentes sistemas operativos. Jess hereda esta característica, la cual lo hace idóneo para enfrentar la creciente demanda de sistemas expertos en los más diversos ámbitos. Encargados de sistemas eléctricos [JAIN2008], bioquímicos [ZHANG2000, NAKAI2004], y médicos [VELICER1999] han pasado a ser algunos de los nuevos operadores, y sus tablets, teléfonos, laptops, y computadoras corporales (los *smartwatch*, por ejemplo) los nuevos dispositivos de operación.

La implementación de Jess puede hacerse de dos maneras:

1. Como un desarrollo directo sobre el shell utilizando el lenguaje propio de Jess.
2. Como una biblioteca externa desde una aplicación Java.

Sin embargo, ambas aproximaciones presentan ciertos retos, los cuales pueden complicarse aún más dada la amplia variedad de dominios de aplicación en donde aparecen los sistemas expertos, como se mencionó previamente.

Realizar un desarrollo directo sobre Jess demanda, además de conocimiento sobre el *shell* y su lenguaje, contar con experiencia en Java y sus bibliotecas para poder crear una funcionalidad con cierto grado de complejidad. La mayor parte de las aplicaciones

actuales requieren acceso a base de datos, conexión a redes y la utilización de una interfaz gráfica de usuario para la interacción con sus operadores. En esta situación, se requeriría de un entorno de programación (IDE) con soporte al *shell* para un proceso de desarrollo óptimo, pero en la actualidad no existe ninguno. Pueden encontrarse diversos entornos que brindan soporte para la creación de todo tipo de aplicaciones sobre Java (por ejemplo Eclipse o Netbeans), aunque ninguno específico para Jess.

Por otro lado, ser incorporado como un componente externo en una aplicación Java sólo requeriría conocer su API, pero así el uso de Jess estaría limitado sólo a sistemas creados sobre esta tecnología. Esto podría ser un problema si lo que se necesita es implementar tecnología de sistemas expertos en un sistema ya existente escrito en otro lenguaje. Asimismo, y a pesar de su capacidad de ser ejecutado en múltiples ambientes, Java puede no estar disponible en el ámbito donde debe ejecutarse el sistema experto (por ejemplo, debido a políticas de seguridad). Además, puede ocurrir que el sistema necesite estar instalado en varios puntos. Con lo cual, cualquier modificación o actualización en el código deberá entonces ser replicada en todas las estaciones de operación; algo que puede llegar a ser costoso dependiendo del volumen de operaciones.

Este proyecto propone la creación de un servicio web (interfaz) entre el *shell* Jess y las aplicaciones y sistemas que requieran incorporar su funcionalidad. De esta forma, gracias a las características inherentes de los servicios web [ver capítulo 4], se tiene un canal de comunicación que abstrae y aísla la complejidad del *shell*. Esta interfaz (el **Jesslet**) hace que estas *capacidades expertas* sean accesibles y de incorporación sencilla como componentes en otros sistemas, ya que están asentadas sobre los estándares abiertos propios de los servicios web [BOUGUETTAYA2014, KALIN2013, CERAMI2002, SNELL2001], dando la posibilidad, asimismo, de un acceso concurrente a la funcionalidad.

I.1 Justificación

La abstracción de Jess en un servicio web hace que la implementación de las capacidades de los shells y los sistemas expertos dentro de otros sistemas se realice con

poco esfuerzo. Esto se debe, principalmente, a que se elimina la necesidad de instalar o contar con software especial, o el aprendizaje de un nuevo lenguaje o sintaxis. Los estándares sobre los que se asientan los servicios web (SOAP, XML, WSDL y HTTP) son abiertos, y los datos se transmiten en texto plano [BOUGUETTAYA2014, KALIN2013, CERAMI2002, SNELL2001]. Así, se consigue que los métodos para la administración y ejecución de los sistemas expertos sobre el shell, posean un alto grado de accesibilidad.

Además, tratándose de un servicio centralizado, toda actualización que sea realizada sobre éste, repercute de manera instantánea sobre los sistemas clientes. De esta manera, la depuración de errores e instalación de actualizaciones en el software solo es necesaria en un único punto.

Como una consecuencia de la simplificación del acceso a Jess, se incrementa notablemente el número de dispositivos para su operación. Los operadores pueden trabajar con interfaces de usuario más diversas y ricas.

El prototipo del Jesslet desarrollado en este trabajo:

- Puede ser incorporado o utilizado dentro de otro sistema de manera sencilla y sin la necesidad de contar con software o componentes especiales.
- Posibilita el trabajo con Jess y los sistemas expertos creados con éste desde diferentes plataformas, sistemas y dispositivos de manera concurrente.
- Es un desarrollo abierto para su estudio y posible mejora posterior por parte de estudiantes e investigadores.

La capacidad que presenta Jess de ser usado como una biblioteca dentro de otra aplicación Java es lo que lo hace adecuado para ser tomado como shell base para el servicio web. Por otro lado, también puede afirmarse que su API está adecuadamente documentada y es posible obtener una licencia para uso académico con sólo solicitarla.

I.2 Antecedentes

Los servicios web han sido ampliamente utilizados en diversos campos, y sirven, en principio, para interconectar dos o más sistemas o componentes de software.

Enumerar las implementaciones de estos servicios sería imposible por los numerosos que resultan en la actualidad, pero entre algunas de ellas se pueden mencionar las siguientes:

- Servicio web de la Administración Federal de Ingresos Públicos, que permite realizar consultas de información (http://www.afip.gob.ar/ws/).
- Servicios web del Sistema Integrado de Información Sanitaria Argentino (SIISA), desde donde puede hacerse consultas y cargar información referente a pacientes, hospitales y establecimientos de salud, profesionales de la salud, farmacias, etc. (https://sisa.msal.gov.ar/sisa/).
- Servicios web para la consulta y reserva de habitaciones en hoteles (https://www.hoteldo.com/).
- Servicios web para la reserva de vuelos (http://www.amadeus.com/).

A partir de la investigación bibliográfica realizada, se ha podido encontrar algunas implementaciones de los sistemas expertos como servicios web. Por ejemplo, en [CHANG2010] los autores proponen un sistema para el diagnóstico de problemas en redes inalámbricas En este caso, se incorporan los servicios web sobre sistemas expertos construidos para la detección de problemas en una LAN inalámbrica, convirtiendo estos sistemas expertos en componentes estructurados como servicios web.

Asimismo, existen implementaciones de sistemas expertos basados en web que utilizan a *Jess* como base. En [HO2005] se introduce un sistema experto basado en web denominado **planificador de programa de clases** (**CSP** por sus siglas en inglés), el cual tiene el objetivo de brindar asistencia y ayudar con la planificación de clases a estudiantes. Este desarrollo utiliza a **Jess** como herramienta de procesamiento de hechos para la obtención de calendarios de clases viables para los estudiantes. [KHADIR2009] presenta el diseño de un sistema para el diagnóstico de fallas y mantenimiento en turbinas de vapor, mientras que [MESTIZO2008] plantea el desarrollo de un sistema experto web para brindar soporte técnico en línea a los usuarios del Sistema de Educación Distribuida (EMINUS) de la Universidad Veracruzana. Ambos aplicativos utilizan a Jess para realizar las inferencias y hacer sus recomendaciones. Finalmente,

[TRAPPEY2009] desarrolla un Sistema Médico Inteligente Móvil (MIMS por sus siglas en inglés) que soporta aplicaciones médicas y de decisiones clínicas. En dicho proyecto, Jess interactúa con aplicaciones móviles con tecnología RFID para el monitoreo de instrumentos fisiológicos. Esta comunicación se realiza mediante un módulo de recolección de datos (hechos), sobre el cual se pueden emitir alertas y enviar mensajes de diagnóstico.

Sin embargo, en el presente proyecto se avanzó un paso más allá al trabajar no sólo con los sistemas expertos, sino con un shell en particular que podrá ser usado para la creación de los mismos. Las implementaciones de Jess encontradas se diferencian del Jesslet creado en que plantean el desarrollo de un sistema experto especialmente desarrollado para brindar una solución a un problema puntual, mientras que el Jesslet permite la creación, administración y ejecución de cualquier sistema experto cuyo conocimiento pueda ser representado con reglas de inferencia.

I.3 Objetivos

En base a la problemática arriba expuesta, se definió el objetivo general y los objetivos específicos para guiar el desarrollo de este proyecto.

I.3.1 Objetivo General

El objetivo general del trabajo es tender a facilitar el uso e implementación de las prestaciones de los sistemas expertos dentro de otros sistemas y/o plataformas. Es decir, esta orientado esencialmente a permitir que la integración de funcionalidades tales como la creación y operación de sistemas expertos se vea simplificada, independientemente del sistema operativo y del lenguaje de programación sobre el cual se esté trabajando.

I.3.2 Objetivos Específicos

Además se formularon dos objetivos específicos, a saber:

1. Eliminar el acoplamiento no deseado entre *Jess* y otros sistemas al ser utilizado como componente.

2. Incrementar la cohesión de *Jess* al integrarse dentro de otro sistema.

I.4 Resumen

En este capítulo se expuso la problemática detectada al momento de implementar Jess, la solución propuesta, el enfoque tomado para el desarrollo del Jesslet, su motivación, y porque resulta importante su abordaje. También se presentaron brevemente los antecedentes encontrados respecto al uso del *shell* Jess como un servicio web o dentro de otro sistema. Por último, se plantearon los objetivos definidos para la realización de este trabajo académico.

Capítulo II Sistemas Expertos

Las diversas aplicaciones de los programas de computación y los sistemas de información ha hecho que estos tomen contacto con un sin número de problemas presentes en la vida cotidiana humana. Inicialmente enfocadas en el procesamiento de grandes volúmenes de datos, las aplicaciones del software han ido mutando e introduciéndose en terrenos cada vez más blandos y complejos. La asistencia en la toma de decisiones estratégicas y la resolución de problemas en donde la experiencia humana es de vital importancia son ahora algunos de los problemas que deben ser atacados. Los sistemas expertos aparecen como una aproximación para enfrentar este tipo de situaciones.

II.1 ¿Qué son los sistemas expertos?

Los **sistemas expertos** son una aproximación a la solución de problemas mediante conceptos vinculados con la **inteligencia artificial**. [FEIGENBAUM1982] los define de la siguiente manera:

Un sistema experto es un programa de computación inteligente que usa el conocimiento y los procedimientos de inferencia para resolver problemas que son los suficientemente difíciles como para requerir significativa experiencia humana en su solución.

En otras palabras, es un sistema que emula la habilidad para tomar decisiones de un humano sobre un problema dado [GIARRATANO1998].

Al formar parte del campo de la inteligencia artificial, los sistemas expertos enfrentan dos dificultades en su implementación [GARCIA2004]:

1 Los seres humanos no saben realmente cómo realizan la mayoría de sus actividades intelectuales.

2 Las computadoras enfrentan las tareas de manera distinta que un ser humano. Esto se debe a que son programadas en lenguajes en los que sólo es posible expresar conceptos muy elementales.

II.1.1 Tipos de Conocimiento

Los conocimientos que manipula un sistema experto pueden ser clasificados de la siguiente manera [GARCIA2004]:

1 **Conocimiento público**. Es el conocimiento que ha sido publicado y se encuentra al alcance de toda la comunidad, por ejemplo en libros, revistas, documentación y especificaciones técnicas.

2 **Conocimiento privado**. Es el conocimiento propio del o los expertos. Es adquirido mediante el ejercicio de sus actividades, y se lo utiliza de manera implícita. Suele ser un tipo de conocimiento que el especialista puede verbalizar, ya que es consciente de su accionar.

3 **Metaconocimiento**. Es un tipo de conocimiento que tienen interiorizado los expertos, adquiridos mediante la realización de sus actividades, pero que no puede ser verbalizado ya que el experto no es consciente de poseerlo y utilizarlo..

En este punto, cabe hacer notar una distinción entre los sistemas expertos y los sistemas basados en conocimiento (SBC). Los primeros están basados en el conocimiento y la experiencia humana en un determinado dominio., es decir, su fuente de conocimientos es el experto. Mientras que los SBC utilizan conocimientos públicos. Sin embargo, esta distinción ha perdido fuerza desde que los sistemas expertos y SBC fueron desarrollados por primera vez en los años setentas, , volviéndose prácticamente sinónimos [GIARRATANO1998].

II.1.2 Ventajas de los Sistemas Expertos

Los sistemas expertos proporcionan **disponibilidad** de experiencia humana en cualquier lugar en donde sea posible su instalación. Asimismo, su implantación puede ser realizada a un **costo bajo**, en comparación con la capacitación y preparación de recursos humanos para las tareas. Además, algunas de esas tareas pueden ser demasiado peligrosas para una persona, por lo que un sistema experto permite **reducir el peligro**. También posibilita tener **experiencia permanente** dentro de la organización, ya que, a diferencia de los humanos, los sistemas expertos no pueden retirarse, renunciar o morir.

Otra gran ventaja es que los sistemas expertos posibilitan la implementación de **múltiples experiencias** de manera simultánea sobre un mismo problema.

Al tratarse de programas de computación pueden brindar una **explicación clara y concisa** sobre el razonamiento utilizado para llegar a una determinada conclusión. Por otro lado, darán **rápidas respuestas**, totalmente **basadas en cálculos y carente de emociones**, cualidad que puede ser determinante cuando un experto debe trabajar bajo mucha presión y fatiga [GIARRATANO1998].

II.2 Componentes de un Sistema Experto

Los componentes elementales que pueden encontrarse en un sistema experto son [GARCIA2004]:

1 La Base de Conocimientos

2 La Base de Datos

3 La Memoria de Trabajo

4 El Motor de Inferencia

5 El Trazador de Consultas

6 El Trazador de Explicaciones

7 El Manejador de Comunicaciones

La **base de conocimientos** contiene el conocimiento que el sistema experto maneja. Es una formulación simbólica manipulable del área de conocimiento sobre el cual el sistema es experto. Su construcción es un punto vital para el correcto funcionamiento del sistema resultante, ya que éste será tan bueno como su base de conocimientos. La función principal de esta base es proveer al motor de inferencia de toda lo necesario para realizar el procesamiento.

Una de las formas para representar el conocimiento en un sistema experto es mediante **reglas de inferencias** o **reglas de producción**. Estas reglas son atribuidas a Newell y Simon, y aparece en su trabajo *Human Problem Solving* en el año 1972. Uno de los aportes más importantes de estos autores fue la conclusión de que gran parte de la

solución humana de problemas puede expresarse mediante estas reglas de producción de la forma `SI ... ENTONCES ...`. Por ejemplo:

```
SI batería baja ENTONCES conectar cable de alimentación
```

Una regla corresponde a una pequeña colección modular de conocimiento llamada fragmento. Los fragmentos se organizan con nexos hacia otros fragmentos con los cuales presenten alguna relación. Newell y Simon popularizaron la representación del conocimiento humano mediante reglas y mostraron cómo es posible razonar con ellas. La idea de esto es que los estímulos recibidos por el cerebro mediante los sentidos disparan reglas en la **memoria de largo plazo**, produciéndose una respuesta apropiada. Esta memoria se compone de cientos de miles de reglas [GIARRATANO1998].

La **base de datos**, también conocida como **base de hechos**, está formada por distintos datos sobre el problema particular que el sistema experto intenta resolver. El propósito principal es proveer de información al motor de inferencia.

La **memoria de trabajo** es una base de datos temporal en la cual el motor de inferencia almacena información inferida a partir de la base de conocimiento, la base de datos, y la misma memoria de trabajo.

En el modelo de Newell y Simon, la **memoria a corto plazo**, o **memoria activa**, se utiliza para almacenar datos e información temporal durante la solución de un problema. Esta memoria, contrario a la de largo plazo, tienen una capacidad sorprendentemente pequeña: entre cuatro y siete fragmentos [GIARRATANO1998]. En Jess, al igual que en el trabajo de estos investigadores, la base de hechos y la memoria de trabajo son lo mismo.

El **motor de inferencia** es el encargado de activar las reglas en función de la información contenida en la base de datos y la memoria de trabajo. También se encarga de proporcionar al trazador de explicaciones las reglas que se activaron para producir una salida determinada.

El motor de inferencia puede trabajar bajo dos principios: *universo cerrado* o *universo abierto*. El principio de **universo cerrado** establece que toda la información

necesaria está definida en la base de datos, por lo que todo aquello que no puede demostrar como verdadero, lo supone falso. Bajo este principio, la base de hechos no puede ser vacía. El principio del **universo abierto** establece que la información necesaria no está contenida en el sistema, por lo que debe comunicarse con el usuario para obtenerla.

Para realizar las inferencias, el motor de inferencia puede utilizar dos estrategias: orientada por el objetivo (*backward chaining*), o bien orientada por los datos (*forwardchaining*). La **estrategia orientada por el objetivo** usa como origen de la inferencia un resultado posible, e intenta construir un árbol hacia los datos conocidos, estando las reglas asociadas a las ramas del mismo. La **estrategia orientada por los datos** parte de los datos, y a partir de éstos intenta construir un conjunto que contenga como elemento al objetivo, usando las reglas de la base de conocimiento.

En el contexto de la investigación de Newell y Simon, otro elemento importante es el **procesador cognitivo**, cuya función principal es determinar qué reglas serán activadas con el estímulo apropiado. Sin embargo, si en un determinado momento hay varias reglas que deben ser activadas de manera simultánea, el procesador cognitivo llevará a cabo una solución de conflicto para decidir cuál posee la prioridad más alta. El motor de inferencia de los sistemas expertos se corresponde con este procesador. El modelo de Newell y Simon para la solución humana de problemas desde la perspectiva de la memoria a largo plazo (base de conocimiento), memoria a corto plazo (base de datos y memoria de trabajo), y un procesador cognitivo (motor de inferencia) es la base de los sistemas expertos modernos basados en reglas de inferencia [GIARRATANO1998].

El **trazador de consultas**, en un sistema experto, organiza y presenta en una forma entendible por el usuario los requerimientos de información. El trazador de explicaciones interpreta los requerimientos del usuario acerca del porqué de ciertas preguntas por parte del sistema, justificando las mismas. Ambos trazadores sólo se utilizan con un motor de inferencia que funcione bajo el principio del universo abierto.

El **manejador de comunicaciones** tiene dos funciones principales: derivar la información inicial que suministra el usuario hacia la base de datos, e interpretar los

mensajes del usuario. Estos mensajes pueden ser respuestas del usuario a una pregunta formulada por el sistema, o bien una solicitud de una explicación a partir de una consulta del sistema.

II.3 Recursos para el Desarrollo de Sistemas Expertos

Una de las decisiones más importantes al momento de definir un problema es en qué profundidad debe ser modelado y el paradigma utilizado para hacerlo. En general, una guía para seleccionar un paradigma es considerar inicialmente el más tradicional, es decir, la programación convencional. Esto se debe a que la experiencia con esta aproximación es mayor, además de que se dispone de una variedad de paquetes comerciales para desarrollos. Si mediante la programación convencional no puede solucionar eficazmente un problema, entonces se debe considerar paradigmas alternativos, como la inteligencia artificial [GIARRATANO1998].

Un lenguaje para el desarrollo de sistemas expertos es un lenguaje de más alto nivel que otros, que permite realizar cosas con mayor facilidad, pero sólo ante un número reducido de problemas. Así, por su naturaleza especializada, un lenguaje de este tipo resulta idóneo para la creación de sistemas expertos, pero no para la programación general.

La principal diferencia entre un lenguaje especializado y uno convencional es el enfoque de la representación. Los lenguajes convencionales están enfocados en brindar técnicas flexibles para poder representar y obtener una adecuada **abstracción de datos**. Mientras que los lenguajes para los sistemas expertos proporcionan herramientas para, además de la representación y abstracción de datos, lograr la **abstracción de conocimiento** [GIARRATANO1998].

Con la explosión comercial de los sistemas expertos, se han multiplicado los recursos disponibles para desarrollarlos. Éstos pueden ser clasificados en lenguajes, herramientas, o shells. Los **lenguajes** son traductores de comandos escritos en una sintaxis bien definida. Un lenguaje para sistemas expertos también proporcionará un motor de inferencia para ejecutar las instrucciones del lenguaje. Una **herramienta** es un lenguaje adicionado con programas utilitarios para facilitar el desarrollo y la

depuración. Estos programas utilitarios pueden incluir editores de texto, depuradores e incluso generadores de código. También pueden proporcionarse ensambladores multiplataforma para obtener software portable. Los **shells**son herramientas con propósitos especiales, diseñadas para cierto tipo de aplicación en las que el usuario sólo debe rellenar la base de conocimiento [GIARRATANO1998].

II.4 Metodología de Desarrollo IDEAL

El desarrollo del **sistema experto** con el que se validará este trabajo, y que se describirá en detalle más adelante, se hizo siguiendo los lineamientos de la metodología **IDEAL** propuestos por [GARCIA2004]. A continuación se presentan cada una de las fases junto con las etapas que las componen:

1. Fase I. Identificación de la Tarea
    - Etapa I.1: Plan de Requisitos y adquisición de conocimientos
    - Etapa I.2: Evaluación y selección de la tarea
    - Etapa I.3: Definición de la característica de la tarea
2. Fase II. Desarrollo de Prototipos
    - Etapa II.1: Concepción de la solución (descomposición en subproblemas y determinación de analogías)
    - Etapa II.2: Adquisición y conceptualización de los conocimientos
    - Etapa II.3: Formalización de los conocimientos y definición de la arquitectura
    - Etapa II.4: Implementación
    - Etapa II.5: Validación y evaluación del prototipo
    - Etapa II.6: Definición de nuevos requisitos y diseño
3. Fase III. Ejecución de la construcción del sistema integrado
    - Etapa III.1: Requisitos y diseño de la integración
    - Etapa III.2: Implementación y evaluación del sistema integrado

- Etapa III.3: Aceptación del sistema por parte del cliente

4. Fase IV. Actuación para conseguir el mantenimiento perfectivo
    - Etapa IV.1: Definir el mantenimiento del sistema global
    - Etapa IV.2: Definir el mantenimiento de las bases de conocimientos
    - Etapa IV.3: Adquisición de nuevos conocimientos y actualización del sistema
5. Fase V. Lograr una adecuada transferencia tecnológica
    - Etapa V.1: Organizar la transferencia tecnológica
    - Etapa V.2: Completar la documentación del sistema experto construido

II.4.1 Identificación de la Tarea

El principal objetivo de esta fase es la **definición de los objetivos de la aplicación**, para así determinar si la tarea en cuestión es susceptible de ser abordada mediante la ingeniería de conocimientos. De ser afirmativo, se definen las características del problema y se especifican los requisitos que enmarcarán la solución del problema.

La primera etapa busca **identificar las necesidades del cliente** mediante la descripción de los objetivos del sistema. Se debe, además, determinar qué funcionalidades se le exigirán, la información que será suministrada, y que información debería ser devuelta.

La etapa de **evaluación y selección de tarea** es una suerte de "estudio de viabilidad", en el cual se evalúa la tarea desde la perspectiva de la ingeniería del conocimiento y se la cuantifica para determinar el grado de dificultad.

En la última etapa de esta fase se **establecen y definen las características más relevantes asociadas al desarrollo de la aplicación**. Para esto, se debe realizar una definición (lo más formal posible) del sistema, en donde se incluyen:

- Requisitos funcionales
- Requisitos operativos

- Interfaz de usuario y con otros sistemas
- Soporte de hardware y software
- Criterios de éxito
- Casos de prueba
- Recursos y herramientas para el desarrollo
- Análisis de costo/beneficio
- Plan y calendario

II.4.2 Desarrollo de Prototipos

Esta fase implica el desarrollo de los diferentes prototipos que permiten ir definiendo y refinando más rigurosamente las especificaciones del sistema, gradualmente hasta conseguir las especificaciones exactas de lo que se puede hacer y cómo realizarlo.

La etapa de **concepción de la solución** tiene por finalidad producir un diseño general del prototipo, en base a la documentación obtenida en la fase anterior.

La **adquisición de los conocimientos** en la segunda etapa puede hacerse en dos facetas: extracción de los **conocimientos públicos** (libros, documentos, manuales, etc.), y educción de los **conocimientos privados** de los expertos. La **conceptualización** consiste, esencialmente, en el entendimiento del dominio del problema, y la terminología y conceptos utilizados. Para tal fin, puede utilizarse mapas conceptuales y diccionarios con la terminología del dominio de la aplicación. La adquisición y la conceptualización puede alternarse cíclicamente para modelizar el comportamiento experto.

La etapa de **formalización** tiene como finalidad expresar los conocimientos relativos al problema y su resolución en la forma de estructuras que puedan ser usadas por una computadora. Esta etapa se divide en dos actividades. Por un lado, se deben seleccionar los formalismos para poder representar en una computadora la

conceptualización obtenida en la etapa anterior, y por el otro, se debe realizar un diseño detallado del sistema experto.

La **implementación** depende del formalismo elegido, junto con la herramienta de desarrollo. Si se dispone de herramienta potente, se puede ahorrar mucho del trabajo de programación. En otro caso, este tarea se realiza como en cualquier otro sistema.

La **validación y evaluación** es uno de los puntos más sensibles y críticos. Estos sistemas están construidos para contextos en donde, generalmente, las decisiones son discutibles. Es posible comparar las respuestas de los expertos con las del sistema para buscar discrepancias. Si las hay, se deberá refinar el sistema.

La última etapa aparece por el desarrollo incremental de cada prototipo, y es aquí donde se elicitan los requisitos del prototipo del ciclo siguiente.

II.4.3 Ejecución de la Construcción del Sistema Integrado

La tercera fase abarca el estudio y diseño de interfaces con otros sistemas de hardware y software, y el desarrollo e implementación de estas interacciones. Finalmente se lleva a cabo la prueba final de aceptación por parte de los expertos y usuarios finales, la cual debe satisfacer todas sus expectativas y exigencias, tanto en lo relativo a su fiabilidad como eficiencia.

II.4.4 Actuación para Conseguir el Mantenimiento Perfectivo

Esta fase trata sobre el mantenimiento del sistema. Debido a las características de un sistema experto, el mantenimiento perfectivo es esencial, ya que no solo puede incrementar las funcionalidades, sino que incorpora nuevos conocimientos. Para este caso, se deben establecer los protocolos para captarlos, registrarlos, e incorporarlos al sistema.

II.4.5 Lograr una Adecuada Transferencia Tecnológica

Esta fase aborda el problema de la transferencia de manejo para una correcta implementación y uso rutinario. Esto puede hacerse mediante la planeación y organización meticulosamente la transferencia mediante entrenamiento en sesiones de

tutoría entre diseñadores y usuarios finales. También debe completarse la documentación y los manuales de usuario para su correcto uso.

II.5 Resumen

A lo largo de este capítulo se ha realizado una presentación sucinta sobre los sistemas expertos, sus objetivos y propósitos. También se han visto los tipos de conocimiento que estos pueden representar, y las tareas que los expertos comúnmente desempeñan. Estas tareas son de interés por el hecho de que deben luego ser realizadas por el sistema de una manera análoga a la realizada por el humano. Se describieron los componentes esenciales de cualquier sistema experto, el rol que desempeña cada uno, y cómo interactúan. Finalmente, se realizó una presentación de la metodología IDEAL para el desarrollo de sistemas expertos, sus fases y qué actividades se realizan en ellas.

Capítulo III JESS

Jess es un lenguaje y motor de reglas de inferencias desarrollado sobre el lenguaje **Java**, y basado, a su vez, en otro motor llamado **CLIPS** (escrito en lenguaje *C*) [FRIEDMAN2003, FRIEDMAN2008]. Aunque está inspirado y es similar a este, Jess se diferencia esencialmente en su implementación. Su tecnología subyacente lo convierte en una buena opción cuando se requiere introducir o implementar las prestaciones de los sistemas expertos basados en reglas de inferencia en aplicaciones sobre Java. Esto se debe a que Jess es en sí un componente sobre esta plataforma, con lo cual puede acceder a un potente conjunto de bibliotecas que permiten la conexión a bases de datos, redes e interfaz gráfica de usuario. A lo largo de este capítulo se realizará una presentación del mundo de Jess y algunas opciones que se pueden tomar para implementarlo dentro de otros sistemas y aplicativos.

III.1 Ejecución de Jess

Para poner a funcionar Jess es necesario contar con el archivo *JAR* (por sus siglas en inglés, **J**ava **AR**chive) que lo contiene, y con la máquina virtual de Java instalada. El archivo de Jess puede ser descargado desde la dirección http://www.jessrules.com/jess/download.shtml.

En el fragmento que sigue se muestra un ejemplo de la ejecución de Jess mediante una consola.

```
:~$ java -cp /opt/jess/jess.jar jess.Main
Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p2 11/5/2008
Jess>
```

Para este ejemplo, se deben ingresar una a una las instrucciones en lenguaje Jess, para luego ser interpretadas y ejecutadas por el motor.

```
Jess> (** 2 8)
256.0
Jess> (printout t "Ground Control to Major Tom!" crlf)
Ground Control to Major Tom!
```

Sin embargo, el ingreso manual de todos los comandos y sentencias resulta poco práctico ante programas que deben realizar diferentes acciones y contienen varias reglas de inferencia. Por ello, Jess admite como parámetro opcional la ubicación un archivo con código Jess, cuyas instrucciones serán ejecutadas luego de la carga. Como muestra por ejemplo, el fragmento siguiente que se almacena en un archivo llamado *main.clp*.

```
; Contenido del archivo 'main.clp'
(bind ?var (* 2 3))
(printout t "This is Major Tom to Ground Control." crlf)
(printout t "I'm stepping through the door, ")
(printout t "and I'm floating in a most peculiar way." crlf)
(printout t "2x3 es " ?var crlf)
```

Para ejecutarlo, su ubicación es enviada como parámetro al invocar el shell.

```
:~$ java -cp /directorio/de/jess.jar jess.Main main.clp

Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p2 11/5/2008

This is Major Tom to Ground Control.
I'm stepping through the door, and I'm floating in a most
peculiar way.
2x3 es 6
```

III.2 El Lenguaje Jess

Jess, además de tratarse de un shell para la creación de sistemas expertos, define un lenguaje para este fin. Al igual que *CLIPS*, la sintaxis del lenguaje de Jess es similar a **LISP**, en donde las *listas* son la base para de programación.

III.2.1 Funciones

Como todo elemento en el lenguaje Jess, las llamadas a funciones también son listas, en donde el primer elemento (**cabecera**) es el nombre de la función a ser

invocada, y los restantes elementos son sus parámetros, tal como muestra el párrafo que continúa.

```
(** 2 16)
(+ 232 898 45 45 12 45 2 -89 -5623 654 854)
(str-cat "¡HOLA " "MUNDO!")
```

Jess viene con un conjunto de *funciones predefinidas* que son de utilidad para la creación de los programas (una lista completa de estas funciones puede encontrarse en http://www.jessrules.com/jess/docs/71/functions.html). Sin embargo, en muchos casos es necesaria la creación de funciones para propósitos específicos del sistema que se pretende desarrollar. La construcción `deffunction` permite justamente esto. Una vez que la función ha sido definida, es posible utilizarla en el resto del software como a cualquier otra función. El siguiente párrafo ejemplifica la creación de la función factorial utilizando `deffunction`.

```
(deffunction factorial (?n)
    "Calcula el factorial de un número"
    (if (> ?n 0) then
        (return (* ?n (factorial (- ?n 1))))
    else
        (return 1))
)
(factorial 5) ; 5! = 120
(factorial 10) ; 10! = 3628800
```

III.2.2 Hechos

Jess mantiene una colección de piezas de información denominadas *hechos*. Esta colección se conoce como **memoria de trabajo**, y los datos que esta contiene son utilizados por las reglas de inferencia durante la ejecución.

El shell ofrece un conjunto de funciones que permiten realizar las operaciones básicas de creación, eliminación, y actualización de hechos sobre la memoria de trabajo:

- `assert`: Agrega un hecho a la memoria de trabajo
- `clear`: Borra todos los hechos y reglas de inferencia

- `deffacts`: Establece los contenidos iniciales de la memoria de trabajo
- `facts`: Muestra los hechos actuales
- `reset`: Inicializa la memoria de trabajo
- `retract`: Borra un hecho puntual
- `watch facts`: Imprime mensajes de diagnóstico ante ciertos eventos

Los hechos definidos en Jess poseen atributos que se conocen como **ranuras**, y cada tipo de hecho tiene un número fijo de estas. Estos tipos de hechos, junto con las ranuras que admiten, se corresponden con **plantillas**, las cuales se definen mediante la función `deftemplate`. Si bien pueden ser de diferentes tipos, como la mayoría de las estructuras de Jess, los hechos son almacenados como *listas*.

III.2.3 Reglas de Inferencia

La **base de conocimientos** está formada por la colección de reglas de inferencia, las cuales ejecutan acciones dependiendo de los hechos que se hallen en la base de hechos o memoria de trabajo. Una regla de inferencia se asemeja a una sentencia `SI ... ENTONCES ..` (`if ... then`) en un lenguaje de programación, pero en un sentido no procedimental. Mientras que, en contexto de programación, estas construcciones se ejecutan en el orden determinado por el programador, en Jess se disparan en el momento en el que son activadas.

Así como existen funciones y construcciones que permiten administrar los hechos, el shell posee otras que lo hacen sobre las reglas:

- `defrule`: Define o actualiza una regla
- `undefrule`: Elimina una regla de inferencia
- `ppdefrule`: Imprime una regla en una forma amigable para el usuario
- `run`: Ejecuta el sistema
- `watch rules`: Muestra mensajes de diagnóstico cuando se dispara una regla

- `watch activations`: Muestra mensajes de diagnóstico cuando se activa una regla

En el anexo A puede encontrarse una especificación más detallada del *lenguaje Jess*, su sintaxis, estructuras de control y las construcciones para la administración de hechos y reglas de inferencia.

III.3 Funcionamiento de Jess

El propósito de Jess, en breve, es la aplicación continua de instrucciones del tipo `SI ... ENTONCES ...`, sobre el conjunto de datos que forman la memoria de trabajo. Supóngase una regla como se muestra en el párrafo que sigue.

```
SI
    usuario admin ESTA ACTIVO
ENTONCES
    IMPRIMIR "El usuario está activo"
    ENVIAR CORREO ELECTRÓNICO admin
```

Esta regla se *activa* si existe en la memoria de trabajo un usuario *admin* en un estado *activo*. En lenguaje Jess, esta se escribiría como aparece en el siguiente párrafo.

```
(
    defrule usuario-activo
    "Regla que imprime un mensaje cuando se halla un usuario
activo"
    (usuario (nombre-usuario admin) (estado ACTIVO))
    =>
    (printout t "El usuario está activo" crlf)
    (enviar-email admin)
)
```

Las acciones que una regla ejecuta (el consecuente) pueden ser: - **Funciones predeterminadas** de Jess - **Funciones definidas por el usuario** mediante `deffunction` - **Funciones definidas en Java**

De esta manera, el principal problema que Jess debe resolver es emparejar algunos de los hechos presentes en la memoria de trabajo con los antecedentes de las reglas de

inferencias especificadas. Con esta finalidad, se debe ejecutar perpetuamente un ciclo compuesto por las siguientes instrucciones:

1. Hallar todas las reglas satisfechas por el conjunto de hechos presentes en la memoria de trabajo.
2. Formar *registros de activación* para estas asociaciones regla/hecho.
3. Optar por uno de estos registros para su ejecución.

Para que la búsqueda de estas activaciones de las reglas de inferencia sea eficiente, Jess implementa el algoritmo **Rete**.

III.3.1 El Algoritmo Rete

Este algoritmo es implementado mediante la construcción de una red interconectada de **nodos**, en donde cada uno de ellos representa una o más pruebas encontradas en los antecedentes de las reglas. Cada nodo posee una o dos entradas, y un número variable de salidas. Los hechos que se van agregando o quitando de la memoria de trabajo son procesados por esta red. Los nodos de entrada se encuentran en la parte superior de la red, mientras que los de salida se ubican en la inferior. En conjunto, forman la **red Rete**, la cual es la implementación en sí de la memoria de trabajo en Jess.

En la parte superior de la red, los nodos de entrada separan los hechos en categorías dependiendo de su cabecera (`usuario` en el ejemplo anterior), y a medida que se desciende, se aplican discriminaciones más finas hasta llegar a los nodos inferiores, los cuales representan las reglas en sí. Cuando un conjunto de hechos atraviesa la red, significa que pasa todos los filtros definidos en el antecedente de una regla particular, y debe convertirse en un *registro de activación*.

III.4 Jess como un Componente

A continuación se presentan dos ejemplos de interacción de Jess con otros sistemas. En el primer ejemplo será utilizado desde un programa escrito en lenguaje **Java**, mientras que en segundo ejemplo será utilizado como un componente desde un programa en **Python**.

III.4.1 Jess en Java

Para utilizar Jess **como un componente dentro de una aplicación Java** se debe agregar el archivo *JAR* en el directorio de clases (*classpath*). Los módulos y clases necesarios se encuentran dentro del *paquete*1 `jess`. El fragmento siguiente es un programa sencillo en Java que muestra cómo hacer esto.

```
public class Main {
    public static void main(String[] args) {
        try {
            jess.Rete motor = new jess.Rete();
            motor.eval("(printout t \"HOLA MUNDO\" crlf)");
        } catch (Exception ex) {
            System.out.println("ERROR: " + ex.getMessage());
        }
    }
}
```

En el ejemplo anterior, se crea un objeto del tipo `Rete` (el motor de inferencias en sí) sobre el cual se va a ejecutar la función `printout` de Jess para mostrar `"HOLA MUNDO"` en la consola. Para ello se hace uso de la función `eval` de la clase `Rete`, la cual interpreta y ejecuta instrucciones en lenguaje Jess.

Otra forma de conseguir el mismo resultado de una manera más precisa, es emplear la clase `Funcall` como aparece en el párrafo a continuación.

```
public class Main {
    public static void main(String[] args) {
        try {
            jess.Rete motor = new jess.Rete();
            jess.Context context = motor.getGlobalContext();
            jess.Funcall f = new jess.Funcall("printout",
motor);
            f.arg("t");
            f.arg("HOLA MUNDO");
            f.arg("crlf");
            f.execute(context);
         } catch (Exception ex) {
            System.out.println("ERROR: " + ex.getMessage());
        }
    }
```

```
}
```

Para ambos casos, esto se compila y ejecuta con las siguientes instrucciones:

```
$~ javac -cp /directorio/de/jess.jar Main.java -d .
$~ java -cp .:/directorio/de/jess.jar Main
HOLA MUNDO
```

III.4.2 Jess en Otras Plataformas

Para poder utilizar Jess sobre otra plataforma que no se trate de Java se puede optar entre dos aproximaciones:

1. Recurrir a una **interfaz de consola de comandos** (**CLI**, por sus siglas en inglés) desde el entorno o plataforma de trabajo
2. Emplear bibliotecas o componentes que permitan acceder a las clases y módulos presentes en el archivo *JAR*

La primera solución encuentra como desventaja el hecho de que el *JAR* que contiene la biblioteca de Jess puede ejecutar código solamente **desde un archivo**; luego, todo código *no Java* que requiera realizar una o más acciones sobre el motor de inferencias, debe primero crear y escribir un archivo con las líneas en *lenguaje Jess* correspondientes.

Un ejemplo del `HOLA MUNDO` anterior en lenguaje *Python* se vería como se muestra a continuación:

```
from subprocess import call
from tempfile import NamedTemporaryFile
import os
with NamedTemporaryFile() as archivo:
    archivo.write('(printout t "HOLA MUNDO" crlf)')
    archivo.flush()
    call(["java", "-cp", "/opt/jess/lib/jess.jar",
"jess.Main", archivo.name])
```

Para esta situación, se utiliza un *archivo temporal* del sistema para almacenar las instrucciones correspondientes. La ejecución de estas devuelve un mensaje como el siguiente:

```
Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p2 11/5/2008

This copy of Jess will expire in 1336 day(s).
HOLA MUNDO
```

Sin embargo, la generación de un archivo cada vez que se necesite realizar algo sobre Jess, indefectiblemente, repercutirá en el desempeño general de la aplicación debido a la carga de operación de E/S.

Una solución quizás un tanto más eficiente (y elegante) es la segunda alternativa, la cual implica hacer uso de herramientas (propias de la plataforma en donde se quiere implementar Jess) que permitan acceder a un archivo *JAR*, y consecuentemente, a todas las clases y paquetes de Jess. En Python existe (entre otras) la biblioteca **JPype** (https://pypi.python.org/pypi/JPype1) que posibilita esto. Usando esta biblioteca el ejemplo mostrado antes quedaría de la siguiente forma:

```
from jpype import *
startJVM(getDefaultJVMPath(), '-
Djava.class.path=/directorio/de/jess.jar')
motor = JPackage('jess').Rete()
motor.eval('(printout t "HOLA MUNDO" crlf)')
shutdownJVM()
```

El resultado de este código es un tanto más escueto que el anterior:

```
HOLA MUNDO
```

Cabe agregar que, utilizando *JPype*, es posible tener un acceso más profundo a la funcionalidad de la biblioteca de Jess que mediante la *CLI*.

III.4.3 Dificultades

El trabajo con Jess **directo desde la consola** encuentra como desafío inicial el conocer con cierta profundidad el lenguaje de Jess. Y el tener que entender un nuevo lenguaje cuando los planes y tiempos de desarrollo apremian puede resultar un gran escollo. A su vez, otro gran reto puede encontrarse al momento de la separación de las salidas y resultados. Debido a que todo se envía al buffer de salida estándar, se deben establecer los mecanismos necesarios para determinar si un resultado recibido es una recomendación, un mensaje de error, o un mensaje de diagnóstico. Este inconveniente resulta particularmente evidente al correr Jess en una *interfaz de consola* desde otras plataformas, en donde además, al tratarse de un proceso separado, el control que se tiene sobre este es prácticamente inexistente. Por estos motivos, esta forma de trabajo con Jess no suele ser una alternativa viable o plausible para un sistema en producción. El trabajo con la consola de comandos puede, no obstante, resultar aceptable en un ambiente de pruebas y experimentación.

Al acceder a Jess como un componente se obtiene una gran flexibilidad en cuanto a las acciones que se pueden realizar sobre el shell. El manejo de las clases internas posibilita la creación de configuraciones más precisas y efectivas de reglas de inferencia, condiciones y acciones, y además, obtener más detalles en lo relativo a la ejecución del programa. No obstante, esta flexibilidad trae consigo otro tipo de complejidad en el código resultante. Si bien por un lado se pierde la carga del trabajo sobre el lenguaje Jess, por el otro aparece la necesidad de entender el funcionamiento de la API de Jess y el uso correcto de sus componentes. Esto, indudablemente, incrementa la cantidad de componentes y módulos que se ven involucrados, lo cual repercute en el acoplamiento del código

La situación puede agravarse aún más si se trabaja desde una plataforma *no Java*, debido a la implementación de herramientas y paquetes externos (*JPype* en el ejemplo anterior), añadiendo así, más dependencias al sistema.

III.5 Resumen

En este capítulo se ha realizado una introducción al shell Jess y sus componentes. Asimismo, se ha referido a las formas en las que este puede ejecutarse, como así también un breve análisis de las ventajas, desventajas y desafíos que pueden encontrarse en ellas.

Capítulo IV Servicios Web

Desde hace ya algún tiempo, las organizaciones advirtieron los beneficios que devenían con la migración de sus negocios hacia la red. Tener un sitio web fue, tal vez, el sello de una empresa tecnológicamente pujante en los 90. La nube es otro paso en la evolución de la computación en las empresas y organizaciones, y los **servicios web** son una de las tantas herramientas que la componen. Este capítulo introducirá los conceptos básicos para el entendimiento de esta tecnología, y los principios sobre los cuáles se fundamenta.

IV.1 ¿Qué son los servicios web?

Un **servicio web** es un componente de aplicación que puede ser accedido por otras aplicaciones a través de una red mediante una combinación de protocolos de comunicación abiertos (como HTTP, XML, SMTP o Jabber) [W3SCHOOLSWS]. Se trata de una unidad lógica autónoma, la cual puede evolucionar aisladamente, manteniendo, a su vez, cierto de grado de estandarización para la comunicación [EARL2005].

Las Figuras 4.1 y 4.2 presentan, en diagramas, la forma en la que trabaja un servicio web. En su esencia, un servicio web funciona como una interfaz, construida utilizando tecnologías estandarizadas de Internet y accesible a través de una red, ubicada entre cierta **funcionalidad** y los **clientes** de esta. Su rol principal es el de actuar como una capa de abstracción respecto a la plataforma sobre la cual se ejecuta dicha funcionalidad. Los servicios web en sí no representan nada revolucionario, sino que son más bien el resultado de la evolución de los conceptos y principios que han gobernado Internet desde sus comienzos [SNELL2001].

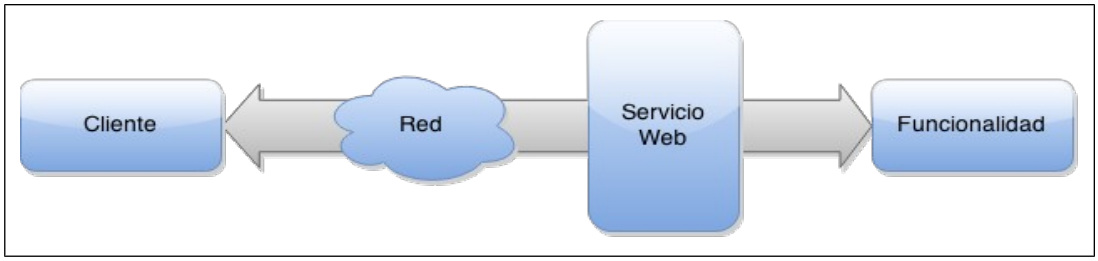


Figura 4.1. Acceso a funcionalidad mediante un servicio web.

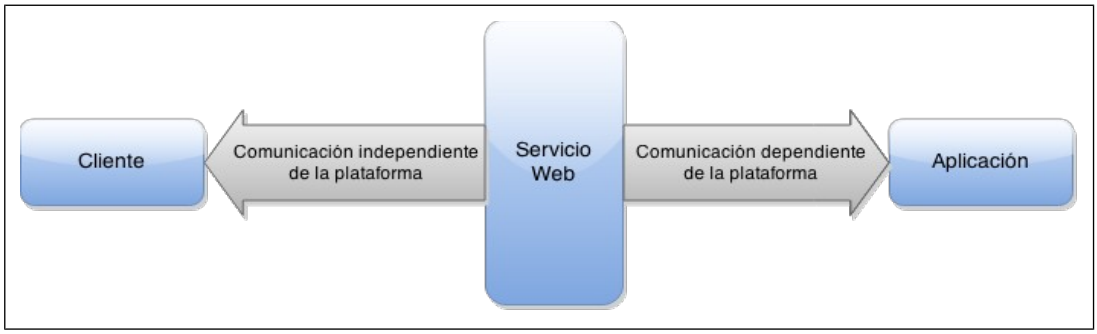


Figura 4.2. Los servicios web como un capa para abstraer la comunicación entre la aplicación y los clientes.

La Figura 4.3 presenta los componentes generales de los **servicios web**. La **aplicación** es quien contiene toda la lógica de negocios. El **demonio de servicio** entiende los protocolos de transporte de Internet (HTTP, SOAP, XML, etc.), y se encarga de escuchar y recibir los pedidos entrantes. El **proxy de servicio** se encarga de codificar y decodificar los pedidos y las respuestas de la aplicación para el demonio. Los **clientes** son los **usuarios** que necesitan acceder a la funcionalidad de la aplicación. Para este caso puntual, el concepto de *usuario* es utilizado en un sentido amplio, abarcando no solo a una persona, sino también a otras aplicaciones y sistemas [SNELL2001].

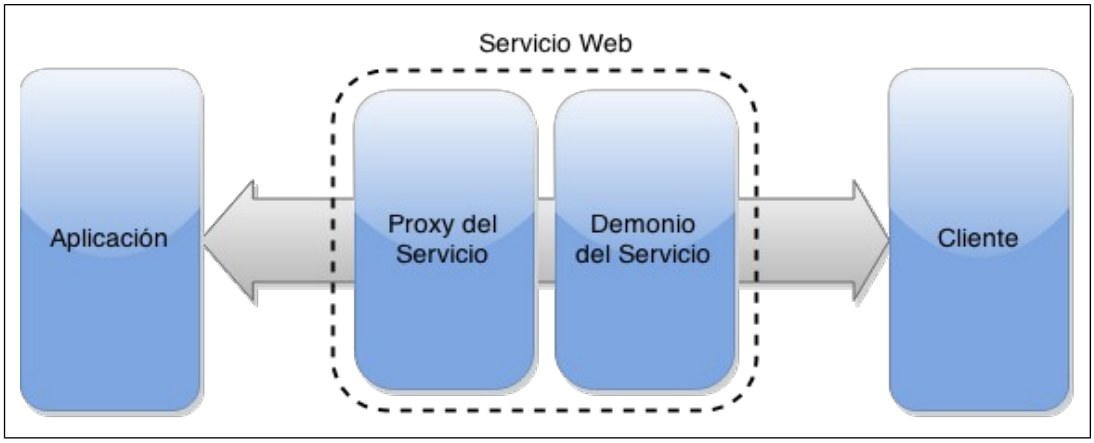


Figura 4.3. Componentes clave en un servicio web.

La relación entre un servicio web y el mundo exterior se realiza en base a una **descripción** de este, la cual contiene, mínimamente, el nombre del servicio, los datos que se esperan recibir y los que se deberían devolver. Como resultado del uso de estas descripciones, el acceso a los servicios web resulta en una relación denominada **débilmente acoplada** [EARL2005]. La herramienta utilizada especificarla es **WSDL** (*Web Services Description Language*) [W3CWSDL], el cual establece cómo y dónde acceder al servicio.

El intercambio de información, en una relación de estas características, se lleva a cabo mediante una plataforma de **mensajería**, la cual gestiona los mensajes que emite y recibe un servicio web independientemente de este y de otros mensajes [EARL2005]. El estándar **SOAP** (*Simple Object Access Protocol*) [W3CSOAP] cumple con estos requisitos, y además posee una amplia difusión en diferentes plataformas como protocolo de mensajes (para más información sobre WSDL y SOAP, ver el anexo B).

IV.1.1 Características de los Servicios Web

Los servicios web presentan ciertas particularidades, algunas de las cuales han resultado vitales para poder conseguir la abstracción de los pormenores del *shell Jess*. En ese sentido, se pueden mencionar las siguientes como las más sobresalientes:

- El **acoplamiento débil** hace que los servicios web se relacionen minimizando las dependencias.

- Un **contrato de servicio** se desprende de las descripciones de los servicios, el cual establece cómo se comunicará.
- Son **autónomos** en el sentido que controlan la lógica que abstraen.
- **Abstraen** dicha lógica y otros pormenores del mundo exterior.
- La abstracción de la funcionalidad permite la **reutilización**.
- Diferentes servicios pueden ser coordinados y ensamblados para crear otros servicios **compuestos**.
- **Carecen de estado** al minimizar la información específica de una actividad.
- Son susceptibles **de ser descubiertos** mediante mecanismos específicos para estos fines.

IV.2 Beneficios de los Servicios Web

El mayor beneficio que los servicios web aportan es el de la **interoperabilidad**. La abstracción provista por las interfaces basadas en estos hace que sea indistinto si la aplicación que brinda el servicio está escrita en Python y su cliente en Javascript, o si una o la otra se ejecuta sobre Solaris, Linux, Windows Phone o Android. Esta tecnología posibilita una forma de comunicación en donde la plataforma se vuelve irrelevante. Por ejemplo, una implementación de mucha utilidad puede encontrarse en los sistemas administradores de bases de datos, sobre los cuales se trabaja con servicios web para que funcionen como fuentes de datos con un alto grado de accesibilidad.

Esta interoperabilidad promueve un diseño netamente **reutilizable**. El hecho de que los servicios sean accesibles hace que estos sean utilizados como partes componentes de otros sistemas y servicios.

La aceptación y dispersión de las tecnologías de los servicios web ha generado un beneficio para los **sistemas heredados**, dando la posibilidad de que estos puedan ser incorporados a nuevas arquitecturas y sistemas de información. Un servicio web, siendo una capa de comunicación, le permite a sistemas en ambientes o plataformas aisladas, concretar el intercambio de información e interoperación [EARL2005].

IV.3 Desafíos en la Implementación de los Servicios Web

La mayoría de los servicios web, al estar basados en estándares abiertos de Internet, realizan el envío y la recepción de mensajes utilizando solo texto. Esto puede generar sobrecargas de procesamiento, y, eventualmente, perjudicar el desempeño general. En numerosos casos se deben transmitir datos complejos, como por ejemplo estructuras y arreglos, y su codificación y decodificación puede resultar muy costosa [KALIN2013].

En sus inicios, generalmente los servicios responden adecuadamente. Sin embargo, a medida que se agrega más funcionalidad, también se ve incrementada la comunicación basada en mensajes. En esta instancia suelen presentarse los problemas de latencia y demoras debido a la sobrecarga de procesamiento. Para prever este tipo de problemas se recomienda realizar pruebas de estrés sobre los ambientes donde se han de ejecutar los servicios. Por otro lado, también debería probarse la capacidad de la plataforma de mensajería, y explorar alternativas para la optimización de su desempeño [EARL2005].

IV.4 Servicios Web en Java

JAX-WS es una API presente en Java, que tiene el propósito de simplificar la creación y el consumo de servicios web. Para tal fin, se define un conjunto de **anotaciones** (ver capítulo IV) que permiten establecer qué es un servicio web, cuáles son sus métodos, y generar el código de máquina correspondiente.

- `@WebService` marca una clase como un servicio web.
- `@WebMethod` marca un método público como un método de servicio dentro de una clase marcada con la anotación anterior.

Cabe notar que la anotación `@WebMethod` es opcional, pero está recomendada. Por defecto, en toda clase anotada con `@WebService`, sus métodos públicos son

considerados de servicio, aún cuando no se encuentran anotados como tales [KALIN2013].

La publicación un servicio web puede hacerse de dos maneras:

1. Mediante un servidor web Java estándar (por ejemplo, Apache Tomcat).
2. Utilizado la clase `Endpoint` para crear publicaciones *ad-hoc*.

A continuación se presenta un ejemplo de una clase definida como un servicio, la cual posee un único método para recuperar el *tiempo UNIX* en base a la fecha y hora actuales.

```
package testws;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class DateTimeService {
    @WebMethod
    public long Timestamp() {
        return System.currentTimeMillis() / 1000;
    }
}
```

La clase `Main`, que publicará el servicio, se define como:

```
package testws;

import javax.xml.ws.Endpoint;
public class Main {
    public static void main(String[] args) {
        try {
            String url = "http://localhost:8888/tsws";
            System.out.println("Iniciando el servicio...");
            Endpoint end = Endpoint.publish(url, new
DateTimeService());
            System.out.println("Servicio Iniciado...");
            System.out.println("Presione ENTER para
cerrar...");
            System.in.read();
            System.out.println("Cerrando el servicio...");
            end.stop();
```

```
        } catch (Exception ex) {
            System.out.println("ERROR: " + ex.getMessage());
        }
    }
}
```

Esto se compila y ejecuta como se muestra en el extracto que sigue.

```
:~$ javac testws/Main.java -d .
:~$ java testws.Main
Iniciando el servicio...
Servicio Iniciado...
Presione ENTER para cerrar...
```

En este punto, el servicio se encuentra en ejecución, con lo que si se realiza un pedido a la URL `http://localhost:8888/tsws?wsdl` mediante **cURL**[1] o un navegador web, se obtiene el documento **WSDL** del servicio generado (para mayor información sobre WSDL y SOAP, consultar el anexo B):

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net.
RI's version is JAX-WS RI 2.2.4-b01. -->
<!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net.
RI's version is JAX-WS RI 2.2.4-b01. -->
<definitions xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://testws/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://testws/"
name="DateTimeServiceService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://testws/"
schemaLocation="http://localhost:8888/tsws?xsd=1"/>
    </xsd:schema>
```

1. cURL es una herramienta y biblioteca basada en línea de comandos para la transferencia de datos con URLs.

```
  </types>
  <message name="Timestamp">
    <part name="parameters" element="tns:Timestamp"/>
  </message>
  <message name="TimestampResponse">
    <part name="parameters" element="tns:TimestampResponse"/>
  </message>
  <portType name="DateTimeService">
    <operation name="Timestamp">
      <input
wsam:Action="http://testws/DateTimeService/TimestampRequest"
message="tns:Timestamp"/>
      <output
wsam:Action="http://testws/DateTimeService/TimestampResponse"
message="tns:TimestampResponse"/>
    </operation>
  </portType>
  <binding name="DateTimeServicePortBinding"
type="tns:DateTimeService">
    <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>
    <operation name="Timestamp">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="DateTimeServiceService">
    <port name="DateTimeServicePort"
binding="tns:DateTimeServicePortBinding">
      <soap:address location="http://localhost:8888/tsws"/>
    </port>
  </service>
</definitions>
```

Para una prueba sencilla, se genera el siguiente mensaje *SOAP* a los fines de ser enviado a la ubicación donde fue publicado el servicio.

```
<?xml version="1.0"?>
<soap:Envelope
```

```
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-
encoding"
    xmlns:ns1="http://testws/">
    <soap:Body>
        <ns1:Timestamp></ns1:Timestamp>
    </soap:Body>
</soap:Envelope>
```

Al efectuar un pedido utilizando el verbo *POST* del protocolo *HTTP*, pasando como datos el contenido del mensaje previamente generado, se obtiene la siguiente respuesta desde el servicio.

```
<?xml version="1.0"?>
<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:TimestampResponse xmlns:ns2="http://testws/">
      <return>1488722277</return>
    </ns2:TimestampResponse>
  </S:Body>
</S:Envelope>
```

Cabe notar que, al tratarse de un estándar abierto y ampliamente utilizado, la mayoría de las plataformas y entornos de desarrollo manejan herramientas y bibliotecas que permiten generar automáticamente estos mensajes SOAP. En *Python* por ejemplo, se encuentra la biblioteca *PySimpleSOAP*, la cual permite utilizar el servicio desarrollado en Java desde este lenguaje. La simplicidad de su uso queda evidenciada en el siguiente ejemplo.

```
from pysimplesoap.client import SoapClient
c = pysimplesoap.client.SoapClient(wsdl =
'http://localhost:8888/tsws?wsdl')
print(c.Timestamp())
```

IV.5 Resumen

En esta etapa se ha realizado una introducción a los servicios web y sus tecnologías fundamentales. A su vez, se ha presentado brevemente la forma en la que

estos pueden ser implementados y publicados en la plataforma Java, y consumidos desde otras sistemas y plataformas.

Capítulo V Programación Extrema

El desarrollo de software y aplicaciones ha cambiado radicalmente desde su concepción inicial hasta nuestros días. En un principio, existía un orden claramente establecido en las etapas del proceso: elicitación de requisitos, desarrollo del programa, pruebas, implementación, y mantenimiento. Sin embargo, a medida que cambió la naturaleza de los problemas con los que se enfrentaban los sistemas, los requerimientos se tornaron cada vez más variables. En muchos casos, arribada la etapa de implementación del software, los requisitos habían sufrido cierta transformación, con lo cual el sistema resultante no brindaba la solución para la cual había sido originalmente concebido.

Las **metodologías ágiles**, como la **Programación Extrema**, cambian la forma en la que se aborda el desarrollo de un programa de computadora, considerando al cambio en los requerimientos no solamente como algo totalmente inevitable, sino incluso como algo deseable [SHORE2008].

V.1 ¿Qué es la Programación Extrema?

La Programación Extrema (XP) es un proceso de desarrollo ágil que ha probado ser exitoso en diferentes tipos de proyectos. La clave de este éxito radica en que la XP hace hincapié en la satisfacción de los clientes más que en la fechas límites. Esto se logra mediante la entrega de software funcional de gran valor para la organización a medida que se lo necesita, en lugar de entregar todo lo que el cliente posiblemente podría llegar a querer en una fecha determinada en el futuro. Además de este, existen otros aspectos y enfoques que también influyen en el éxito de esta metodología [SHORE2008] a saber:

- Prioriza a los individuos y a las interacciones sobre los procesos y las herramientas.
- Valora el software funcional sobre la documentación comprensible.
- Enfatiza la colaboración con los clientes más que la negociación del contrato.

- Busca y toma ventaja de los cambios más que usar un plan preestablecido.

V.2 Principios de la XP

Como se dijo, la prioridad más alta es la satisfacción del cliente; y esta se consigue mediante entregas de software rápidas y de alto valor. Uno de los objetivos es generar software funcional de manera frecuente, desde un par de semanas a un par de meses, aunque con preferencia al período más corto. Estas entregas periódicas representan la mejor medida del progreso del proyecto.

Asimismo, incluso en etapas tardías del desarrollo, los cambios son bienvenidos. La XP no descuida los cambios, sino que los persigue como oportunidades, ya que estos pueden ser cruciales para que el cliente logre una ventaja competitiva. Estos cambios no pueden surgir de los programadores, sino que deben ser propuestos y definidos por los clientes. Es por esto, que las personas vinculadas al área de negocios deben trabajar conjuntamente a los desarrolladores diariamente para garantizar que las características con las que cuenta el sistema tienen valor para la organización.

Para mejorar la agilidad, el equipo debe prestar atención continua a la excelencia técnica y el buen diseño. La simplicidad, considerada como *el arte de maximizar el trabajo no realizado*, resulta igualmente esencial en la XP. Con las entregas periódicas el equipo muestra cómo ser más efectivo para luego ajustar su comportamiento oportunamente [SHORE2008].

V.3 El Ciclo de Vida de la XP

La Programación Extrema resulta novedosa y, hasta cierto punto, sacrílega para las personas vinculadas a la planificación de proyectos de desarrollo. Esto se debe a que en los enfoques de desarrollo tradicionales, las etapas y las actividades que las componen están claramente identificadas, planificadas y estructuradas, y se tiene una clara separación entre las etapas de análisis, diseño, programación y pruebas, implementación y mantenimiento. En la XP, en cambio, todo el proceso de desarrollo se fragmenta en **iteraciones**, cada una de las cuales tiene el objetivo de producir una versión funcional del sistema, motivada principalmente por el hecho de que en la XP se

admite que los clientes no son capaces de especificar sus requerimientos al comienzo de un proyecto.

En cada iteración (Figura 5.1), todas las fases se hacen de manera simultánea, produciendo entregas de software periódicas y de gran valor para el negocio.

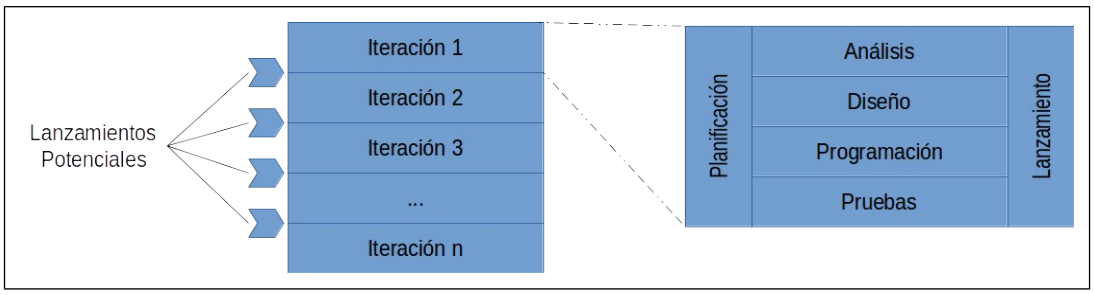


Figura 5.1. Ciclo de Vida de la XP

Este enfoque no pretende ser el más productivo, sino que trata de obtener realimentación rápidamente de parte del cliente. La principal motivación de ello es que permite al equipo adecuar los planes de manera más rápida, y al hecho de que un cliente puede refinar una idea futura si este posee ya un prototipo funcional [SHORE2008].

En la Figura 5.2 se presenta más detalladamente el ciclo de vida de un proyecto XP, el cual como cualquier otro proceso de desarrollo pretende determinar lo que el cliente necesita, para luego proceder a crear una solución que lo satisfaga, pero en este caso de una manera diferente.

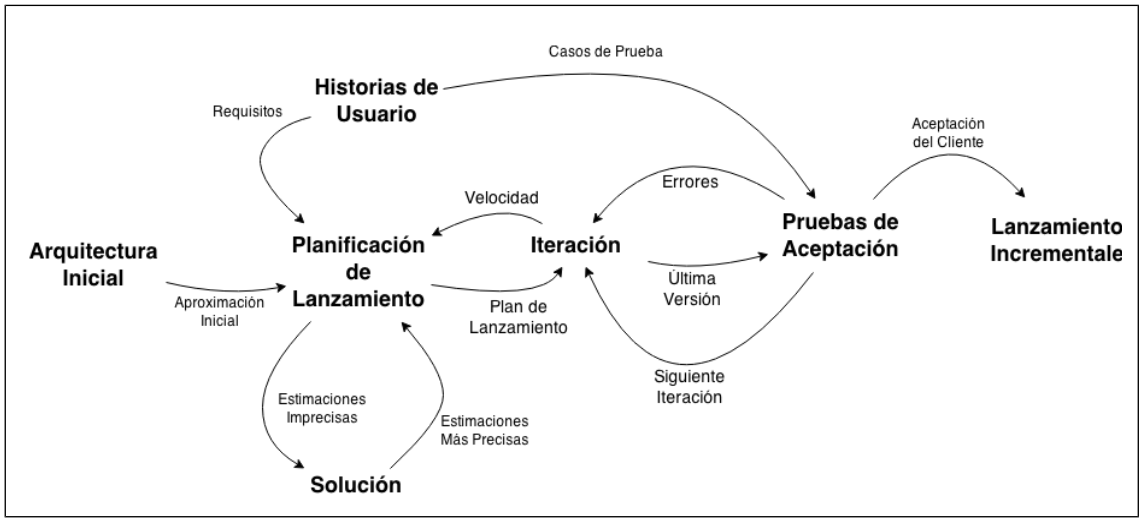


Figura 5.2. Ciclo de Vida de un Proyecto XP

En particular, en la XP se comienza con una **fase de exploración**, en la cual se define el alcance general del proyecto. Para ello, el cliente define lo que necesita mediante la redacción de sencillas **historias de usuarios** 1, y a partir de las cuales, los programadores, estiman de manera aproximada los tiempos de desarrollo, y a su vez tratan de establecer una arquitectura inicial sobre la cual comenzar a trabajar.

Establecida la visión general del sistema y el tiempo de desarrollo estimado, se procede a la fase de **planificación**. En ella, el cliente, los gerentes y el grupo de desarrolladores definen el **plan de lanzamiento**, el cual establece el orden en el cual deberán implementarse las historias de usuario, y, asociadas a éstas, las entregas. Durante esta etapa se tratan de priorizar aquellas historias de usuario que le produzcan mayores beneficios para el cliente, así como también aquellas que resulten más críticas desde el punto de vista del desarrollo.

La **fase de iteraciones**, es la fase principal en el ciclo de desarrollo de la XP. Cada una de las iteraciones tiene por objetivo producir una aplicación funcional libre de errores que implemente las historias de usuario asignadas a la iteración. Para ello se realiza un ciclo completo de análisis, diseño, desarrollo y pruebas, pero utilizando un conjunto de reglas y prácticas que caracterizan a la XP, tal cual se muestran en la Figura 5.3.

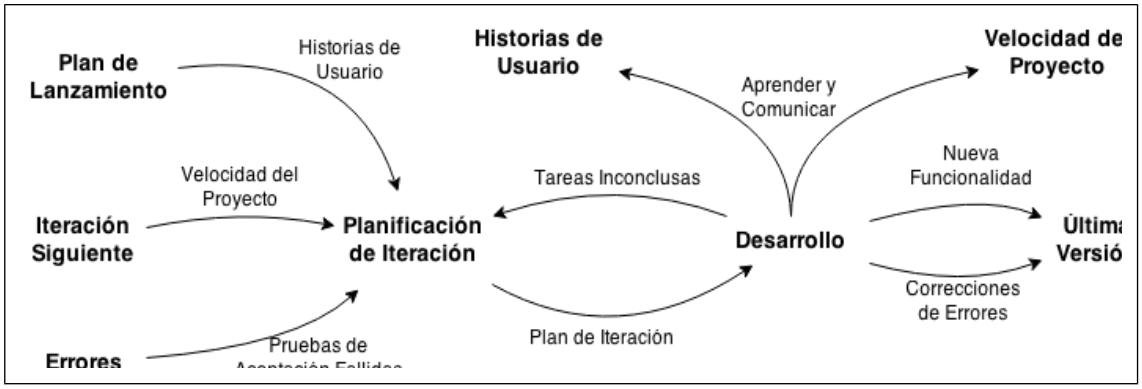


Figura 5.3. Ciclo de una Iteración

Como las historias de usuario existentes no tienen suficiente detalle, al principio de cada iteración se realizan las tareas necesarias de análisis junto al cliente para obtener toda la información que resulte necesaria. Este debe participar de manera activa durante

esta fase del ciclo a los fines de que la funcionalidad obtenida sea de alto valor para la organización. Las iteraciones sirven como una medida del avance del progreso del proyecto.

V.4 Reglas y Prácticas de la XP

Si bien el ciclo de vida de un proyecto XP es muy dinámico y su aplicación ha demostrado ser exitosa, resulta importante conocer aun más acerca de las reglas y prácticas que la gobiernan, a fin de poder aplicarlo de manera satisfactoria.

V.4.1 Historias de Usuario

Las **historias de usuario** son descripciones de trabajo de una o dos líneas, que el equipo debe producir. No son requisitos o casos de uso; son más simples que estos. El objetivo de estas historias es el de planear el trabajo en piezas pequeñas centradas en el cliente. Una historia es un marcador de posición para una discusión detallada sobre los requerimientos. Los clientes son los responsables de proveer estos requisitos cuando el resto del equipo se los solicite.

Las historias de usuario poseen las siguientes características:

- Representan valor para los clientes y están escritas en una terminología comprensible por estos.
- Poseen criterio de completitud claro. Los clientes pueden describir pruebas objetivas que permiten los programadores discernir si una historia ha sido o no implementada exitosamente.

Un resultado directo de tener historias centradas en los clientes es que no aparecerán historias para cuestiones técnicas. Estas deben incluirse en las estimaciones. A su vez, es recomendable que las historias sean escritas en fichas para facilitar la planificación. Las fichas físicas son más efectivas y simples que las herramientas computarizadas [SHORE2008, WELLS2009].

Una buena manera de manera de garantizar que las historias de usuario cumplen con las características arriba mencionadas es que estas sean escritas por los clientes.

Las historias pueden ser de cualquier tamaño, pero si son demasiado grandes o demasiado pequeñas, su estimación puede ser algo problemática. Para que estas tengan un tamaño óptimo, es posible combinarlas y dividirlas. Este proceso debe hacerse teniendo en consideración que se recomienda completar entre cuatro y diez historias de usuario por iteración [SHORE2008].

V.4.2 Iteraciones

Una iteración es un ciclo completo de diseño, programación, pruebas e implementación, y su duración, en la mayoría de los casos, va des una a tres semanas. Cada iteración comienza con los clientes indicando cuáles historias de usuario deberían ser implementadas, y termina con el equipo instalando software que el cliente puede usar y probar. El inicio de cada iteración es el punto en el cual los clientes pueden modificar el rumbo general del proyecto.

V.4.3 Timeboxing (Límites Temporales)

Este concepto significa especificar un bloque de tiempo perfectamente definido para la investigación o discusión, y detenerse cuando se acaba el tiempo, sin importar el progreso final. El principal desafío de esta práctica es el hecho de que resulta difícil detenerse cuando la solución podría estar a segundos, sin embargo es muy importante para aprender a manejar el tiempo, y no perderlo en discusiones improductivas.

V.4.4 Reuniones de Pie

Las **reuniones de pie** tienen el propósito de dar a conocer lo que se hace al resto del equipo, y el motivo de que sean de pie es para mantener una reunión breve. En estas reuniones, que se realizan todas las mañanas, los miembros se mantienen de pie en un círculo, y, uno a la vez, describen nueva información relevante para el resto. Esta información puede abarcar problemas, soluciones o nuevos enfoques, siendo obligatoria la mención de los siguientes puntos:

1. Qué se realizó el día de ayer.
2. Qué se intentará realizar hoy.
3. Qué problemas están causando demoras.

En una reunión de proyecto típica, los asistentes por lo general no participan, limitándose a escuchar aquello que se desea comunicar. Las reuniones de pie en cambio, sirven para comunicar sólo aquello que necesita ser comunicado, sin incurrir en largas discusiones, ya que su clave está en su brevedad. Sólo se busca transmitir una idea somera, no un reporte pormenorizado. Los pies les recuerdan a los participantes que deben ser breves; treinta segundos para cada uno debería ser más que suficiente. Algo que cabe destacar es que no se debería esperar por una de estas reuniones para comunicar un problema importante [SHORE2008].

V.4.5 Programación de a Pares

Todo el código que pasa a producción debe ser creado por dos personas trabajando al mismo tiempo sobre una misma computadora. Contra todo razonamiento lógico, dos personas trabajando sobre la misma máquina agregará tanta funcionalidad como trabajando por separado, pero con la diferencia de que con esta última manera, la calidad del código será inferior [WELLS2009].

La mejor manera de realizar la programación de a pares es sentarse de a dos frente al mismo monitor. La persona que se encarga de escribir el código se denomina **conductor**, y quien se encuentra sentado atrás, **navegador**. Mientras que el conductor se enfoca en escribir código de manera correcta, el navegador puede hacer consideraciones estratégicas y de diseño.

Con esta forma de trabajo en conjunto se puede crear código de una calidad superior que la que se tendría trabajando por separado. Esta práctica incrementa el poder cerebral al momento de atacar un problema, permite tener una mayor concentración, además de reforzar los buenos hábitos de programación como pruebas continuas y refinamiento de diseño [SHORE2008].

Para una correcta aplicación de la programación de a pares pueden seguirse algunas de las siguientes recomendaciones [SHORE2008], [WELLS2009]:

- Los pares deben formarse naturalmente.
- Los pares deben variar durante la jornada.

- Se deberían formar pares siempre que se necesite completar trabajos de mantenimiento.
- Cambiar de pareja cuando se requiera una perspectiva fresca o un nuevo enfoque sobre un problema.
- Evitar emparejarse con la misma persona más de una vez al día.
- Colaborar, no criticar.
- Intercambiar los roles de navegador y conductor frecuentemente.

Por otro lado, también posee ciertos retos tales como compatibilizar las capacidades diferentes de los programadores, su experiencia, los diferentes estilos de comunicación, solo por mencionar algunos.

V.4.6 Estándares de Codificación

Un **estándar de codificación** es un conjunto de recomendaciones a las cuales los programadores acuerdan apegarse cuando programan. Un enfoque común para llegar a esto es partir de un estándar previamente definido para el lenguaje sobre el cual se va a trabajar. De esta forma, el consenso puede enfocarse en otras cuestiones relativas al diseño, como:

- Prácticas de desarrollo
- Herramientas e IDEs
- Estructura de directorios
- Convenciones de compilación
- Manejo de errores y pruebas
- Eventos y registro
- Convenciones de diseño

Al momento de convenir un estándar, se recomienda aplicar los siguientes lineamientos [SHORE2008]:

1. Crear un conjunto de estándares mínimos aceptables.
2. Enfocarse en la consistencia y trabajar sobre la perfección.

V.4.7 "Hecha Hecha"

Una historia de usuario está *"hecha hecha"* cuando se encuentra completamente lista para entrar en producción; es decir, un conjunto de código integrado, probado y listo para ser implementado. Las historias no "hechas hechas" (parcialmente completas) poseen costos ocultos y sorpresivos, siendo la meta de esta práctica evitar demoras a causa de esto. Como lineamiento, una historia de usuario puede considerarse "hecha hecha" si cumple con los siguientes requisitos:

- Probada
- Codificada
- Diseñada
- Integrada
- Compila correctamente
- Se instala correctamente
- Realiza las migraciones de datos de manera correcta
- Revisada
- Depurada
- Aceptada

El diseño guiado por las pruebas puede ser útil para unificar las pruebas, la codificación y el diseño. Los clientes deben probar el software constantemente para evitar cambios de último momento. También se debe trabajar con historias de usuario lo suficientemente pequeñas para que todo el trabajo pueda ser finalizado en una semana.

V.4.8 Sistema de Control de Versiones

Un **sistema de control de versiones** (*SCV*) provee un repositorio central que ayuda a coordinar los cambios en los archivos, además de mantener un historial de los cambios que se realicen. Cuando un equipo trabaja con un SCV, los programadores pueden recuperar la última versión desde el servidor, realizar los cambios necesarios, ejecutar las pruebas, y subirlos nuevamente. El manejo del historial hace que sea posible volver a un código antiguo y revisarlo en busca de errores.

No es recomendable versionar el código generado (por ejemplo, en una compilación), pero sí los requisitos de los usuarios. Como regla general, sólo debería subirse código que compila correctamente y pasa satisfactoriamente todas las pruebas. Existen diferentes niveles de completitud del código fuente:

1. No finalizado: sólo debe estar en el repositorio local.
2. Compila y pasa las pruebas: todas las revisiones en el repositorio central deben poseer este nivel.
3. Listo para los actores: se encuentra lista para que los actores la prueben.
4. Listo para los clientes: se encuentra listo para entrar en producción.

Al manejarse un SCV, una situación bastante común son los conflictos. Un conflicto aparece cuando dos o más programadores intentan modificar un mismo archivo al mismo tiempo. Aunque la mayor parte de estos sistemas de control poseen herramientas para afrontar esta situación.

V.4.9 Desarrollo Guiado por las Pruebas

El **desarrollo guiado por las pruebas** (*TDD*, por sus siglas en inglés) es un ciclo rápido de prueba, codificación y refactorización. Cuando los programadores deben agregar una nueva característica a un sistema repiten un gran número de estos ciclos, implementando y refinando código fuente progresivamente, en pequeños pasos. Las investigaciones muestran que el TDD reduce radicalmente la incidencia de los errores [JANZEN2005]. Usado apropiadamente, ayuda a mejorar el diseño, a documentar las interfaces, y a servir de vigía ante futuros errores [SHORE2008, WELLS2009].

El TDD comienza con las pruebas. Estas se escriben desde la perspectiva de la interfaz pública de una clase, y se enfocan en su comportamiento, no en su implementación. Al ser escritas antes que el código de producción, lo que se consigue es que los programadores generen interfaces que sean fáciles de utilizar más que de implementar.

Por otro lado, estas pruebas, al ser versionadas junto con el código, actúan como un documento viviente. Son reejecutadas luego de cada modificación, garantizando que el código funciona como se pretendía originalmente.

Sin embargo, el TDD no es perfecto. Uno de los inconvenientes que posee es que suele ser difícil de utilizar en fragmentos de código heredados. Inclusive en proyectos nuevos, lleva meses de uso intensivo superar su curva de aprendizaje.

Cada ciclo del TDD se puede ver como un ciclo de **rojo**, **verde**, **refactorización**. La Figura 5.4 ilustra estos pasos.

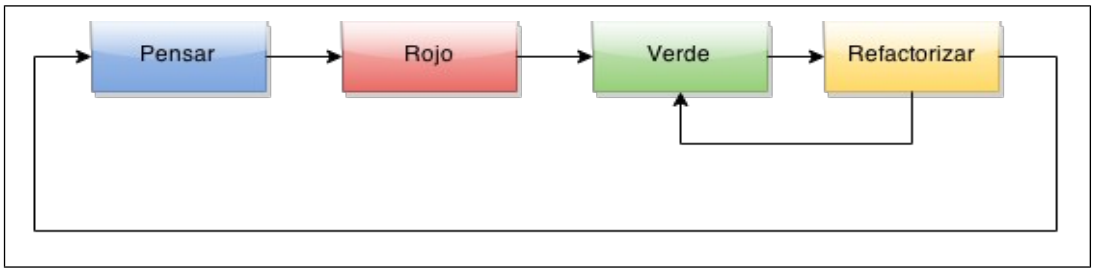


Figura 5.4. Ciclo Rojo, Verde y Refactorización

1. **Pensar**. El TDD usa pequeñas pruebas para forzar a los programadores a escribir el código. Sólo se escribe código suficiente para pasar las pruebas. Inicialmente se debe plantear o pensar el comportamiento que se quiere posea el código, y luego pensar en un pequeño incremento que requiera menos de cinco lineas de código. A continuación, pensar en una prueba que falle a menos que ese comportamiento esté presente. Esta resulta ser la mayor dificultad del TDD. Ir de las pruebas al código parece marcha atrás, y puede resultar muy complicado pensar en pequeños incrementos.
2. **Rojo**. Se escribe la prueba, y esto debe hacerse en términos del comportamiento de la clase y su interfaz pública. Inicialmente se intentará utilizar métodos o clases inexistentes. Luego de esta prueba, la ejecución debería fallar. Si esto no ocurre o falla de una manera inesperada, entonces algo no salió bien. No sólo es importante analizar los éxitos inesperados, sino también los fallos inesperados.
3. **Verde**. Se escribe solamente el código necesario para pasar la prueba. No hay necesidad de preocuparse en este punto por el diseño o la arquitectura. Si la prueba falla y no se sabe porqué, es mejor volver el código anterior a las modificaciones. Esta es una segunda oportunidad para comparar las intenciones del programador con la realidad del sistema.

4. **Refactorizar**. Si las pruebas pasan, se debe revisar el código para encontrar posibles mejoras en el mismo. Para cada problema que se encuentre, se debe refactorizar el código para solucionarlo. Se debería trabajar en series de pequeñas refactorizaciones, y todas las pruebas deben pasar siempre. Se puede mejorar el código todo lo que se crea necesario, pero siempre limitando el alcance al comportamiento ya existente en el código.
5. **Repetir**. Cuando se quiere agregar nuevo comportamiento, el ciclo debe volver a comenzar. La clave para el TDD, nuevamente, son los pequeños incrementos.

V.4.9.1 Pruebas de Unidades

Las **pruebas de unidades** se enfocan en una clase o método, y siempre son ejecutadas en la memoria. Puntualmente, una prueba de unidad no [SHORE2008]:

- Interviene en la base de datos.
- Se comunica sobre la red.
- Interviene en el sistema de archivos.
- Necesita configuraciones especiales para su ejecución.

Para ello se utilizan objetos *mock* , los cuales son un artificio popular para aislar las clases en las pruebas de unidad, ya que se los utilizan para evitar las operaciones arriba mencionadas.

V.4.9.2 Pruebas de Integración

Toda prueba que requiera conexión a la base de datos, la red o el sistema de archivos es una **prueba de integración**, y están enfocadas en la interacción con el mundo exterior. Como principio se debe indicar, que cada prueba debe estar aislada de las restantes [SHORE2008].

Uno de los retos de estas pruebas radica en la preparación del recurso externo, debiendo la misma establecer el entorno que necesita y luego restablecerlo.

No deberían necesitarse demasiadas pruebas de integración. Su cantidad debería ser proporcional al número de interacciones externas. La necesidad de muchas pruebas de integración es una clara señal de problemas.

V.4.10 Refactorización

La refactorización es *el proceso de cambiar el diseño del código sin modificar su comportamiento*. Esta actividad lleva a un diseño reflexivo, mediante el cual se puede analizar el diseño del código existente y mejorarlo. El diseño reflexivo ayuda a entender qué cambiar; mientras que la refactorización permite llevar a cabo esos cambios. Debe notarse que refactorizar no significa reescribir. Los cambios se realizan en pequeños pasos controlados.

V.4.10.1 Code Smells

Una de las mejores maneras de mejorar el código son los ***code smells*** [SHORE2008]. Un *code smell* no necesariamente significa que existe un problema, sino sencillamente algo que merece ser revisado. A continuación se enuncian los *code smells* más comunes.

- El **cambio divergente** ocurre cuando cambios no relacionados afectan a la misma clase. En este caso se recomienda dividir la clase.
- La **cirugía de escopeta** se da cuando deben modificarse muchas clases para una misma idea.
- La **obsesión primitiva** es la representación de conceptos de diseño de alto nivel mediante tipos de datos primitivos.
- Los **grupos de datos** aparecen cuando varias primitivas representan un concepto como un grupo. Esto puede verse al encontrar conjuntos de variables pasadas a lo largo del código.
- Las **clases de datos** son clases que poseen sólo variables y métodos de acceso.
- Las **clases pseudoestáticas** son clases que poseen métodos, pero ningún estado de objeto significativo.
- Las **nulificaciones** pueden provocar errores inesperados.
- Las **dependencias temporales** ocurren cuando los métodos de una clase deben ser invocados en un orden específico. Estas suelen indicar problemas de encapsulamiento.

- Los **objetos a medio cocinar** son un tipo especial de dependencia temporal: primero deben ser construidos, luego inicializados con un método, para recién ser usados.

V.4.11 Soluciones *Spike*

Una **solución *spike*** es, en esencia, un experimento aislado para recavar información sobre una tecnología o campo desconocido. En otras palabras, es una investigación técnica para encontrar la respuesta a un problema.

Debido a que la XP prefiere los datos concretos sobre la especulación, ante cualquier duda, la mejor alternativa es realizar el experimento. Estos experimentos no están pensados para ser implementados directamente en el código en producción. Puede ser desechado o guardado como documentación [SHORE2008, WELLS2009].

Los *spikes* pueden incorporarse en las estimaciones de las historias de usuario, o bien pueden convertirse en historias en sí mismas.

V.4.12 Deuda Técnica

Es el total de decisiones de diseño e implementación de baja calidad realizadas en el proyecto, como por ejemplo soluciones rápidas que tienen el fin de hacer funcionar algo de manera casi instantánea, y decisiones de diseño que no son aplicables debido a cambios en los negocios. También abarca porciones de código pobremente probadas. La factura de esta deuda es visible en la forma de un alto costo de mantenimiento. De no ser pagada, la deuda puede crecer hasta ahogar los proyectos. La XP utiliza una aproximación fundamentalista de aversión para con la deuda técnica.

V.4.13 Tiempo Extra (*Slack*)

El **tiempo extra** (*slack* en inglés) de un proyecto es tiempo que se introduce en las estimaciones de las historias de usuario, que puede o no ser utilizado, para hacer frente a sorpresas o imprevistos. Este tiempo no depende del número de problemas que enfrenta el proyecto, sino de la aleatoriedad de los mismos.

Una de las maneras de introducir este tiempo extra en las iteraciones es no agendar trabajo alguno en el último día de trabajo. Este enfoque es simple, pero genera pérdida de tiempo. Una mejor aproximación es agendar trabajo útil y no dependiente del tiempo; es decir, trabajo susceptible de ser pospuesto en caso de una emergencia. El pago de la deuda técnica encaja perfectamente en esta categoría. No importa que tan disciplinado sea el equipo, siempre existirán fragmentos de códigos descuidados y la deuda técnica tenderá a subir [SHORE2008]. Por cada iteración, se recomienda invertir el 10% del tiempo de desarrollo a pagarla.

La investigación es otra actividad útil que permite introducir laxitud en el proyecto. Es posible destinar mediodía para cada programador para realizar actividades de autoaprendizaje en tópicos de su preferencia.

El uso excesivo de este tiempo extra en un proyecto puede ser un síntoma de problemas mucho mayores a nivel sistémico. Utilizar constantemente el slack es un indicio de un compromiso mayor que el equipo puede realizar.

V.4.14 Velocidad

La velocidad es una forma de calendarizar las estimaciones, y se calcula sumando el total de las estimaciones para las historias de usuario finalizadas en cada iteración. Este concepto resulta vital para mejorar la precisión de las estimaciones realizadas por los programadores y la planificación general del proyecto. Al inicio de cada **reunión de iteración**, a los clientes se les permite escoger un número de tareas que equivalgan a la velocidad de la iteración anterior.

Este sencillo cálculo hace que las estimaciones se tornen más precisas a medida que avanza el proyecto. En las primeras iteraciones, normalmente la velocidad varía, pero tiende a ser constante a medida que el equipo se adecúa a la forma de trabajo.

V.4.15 Retrospectivas

Las retrospectivas son herramientas que permiten mejorar de manera continua el proceso y los hábitos de trabajo, ayudando a su vez a afrontar los cambios constantes. Pueden encontrarse cuatro tipos de retrospectivas:

1. Retrospectivas de Iteración: ocurren al final de cada iteración.
2. Retrospectivas de lanzamiento: se realizan luego de una implementación.
3. Retrospectivas de proyecto: ocurren al concluir el proyecto.
4. Retrospectivas sorpresa: se conducen cuando un evento inesperado modifica la situación general.

Cualquier persona puede conducir una retrospectiva si el equipo funciona adecuadamente. Todos los miembros del equipo deberían participar, y, a los fines de que todos puedan hablar abiertamente, ningún no miembro debería estar presente.

Nunca debe utilizarse una retrospectiva para culpar o atacar individuos. Es recomendable que todos los participantes estén de acuerdo con esta directriz y la cumplan durante el desarrollo de la reunión. Algunos coaches optan por explicitar esto antes de comenzar de la siguiente manera:

Más allá de lo que se descubra hoy, entendemos y realmente creemos que todos hicieron su trabajo lo mejor que pudieron, dado lo que conocían en su momento, sus habilidades, capacidades, los recursos disponibles, y la situación encontrada.

Esta directiva se escribe en un lugar central y visible, y todos los presentes deben estar de acuerdo. El desarrollo de una retrospectiva no está estandarizado ni posee una serie de pasos a seguir. Una buena idea es realizar una tormenta de ideas sobre los problemas, dificultades, eventos e ideas en general que los miembros consideren debe ser tratados para su análisis y mejora.

La realización frecuente de retrospectivas tiene dos ventajas [SHORE2008]:

1. Compartir ideas cohesiona al equipo.
2. Proponer soluciones permite mejorarlo.

V.5 Resumen

En este capítulo se ha realizado un resumen de la filosofía de la **Programación Extrema** y su forma de encarar el desarrollo de sistemas de información. Se han presentado sus *principios* y las etapas de su *ciclo de vida*.

También fueron introducidos los *conceptos* más importantes que aparecen en un desarrollo mediante XP, los cuales son necesarios para poder comprender muchas de las *prácticas* del proceso. Entre estas prácticas pueden distinguirse la *programación de a pares*, las *historias de usuario*, el *desarrollo guiado por las pruebas*, las *reuniones de pie*, y la *refactorización* como algunas de las más influyentes y características de la metodología. Por ultimo, cabe destacar que las prácticas son *recomendaciones* que moldean la Programación Extrema, y, como tales, su aplicación no resulta indispensable en la totalidad para el proceso.

Capítulo VI Métricas para la Cohesión y el Acoplamiento

Las dos hipótesis que marcan el rumbo de este proyecto son las siguientes:

1. *Utilizar el Jesslet reduce el **acoplamiento** entre Jess y otros sistemas al ser implementado como un componente*
2. *Utilizar el Jesslet aumenta la **cohesión** de Jess al integrarse dentro de otro sistema*

Para poder corroborar o refutar estas hipótesis se hizo uso de las métricas de **cohesión** y **acoplamiento**, las cuales se presentan en el presente capítulo, así también como la forma en que se realizarán las mediciones.

VI.1 La Cohesión y el Acoplamiento

A lo largo de la historia de la Ingeniería de Software, ha ido evolucionando un conjunto de conceptos fundamentales acerca del diseño tales como la abstracción, la arquitectura, los patrones, la modularidad, la independencia funcional, etc., los cuales deben tenerse en consideración si se pretende conseguir software de alta calidad; sin importar el modelo de proceso que se elija, los métodos de diseño que se apliquen o el lenguaje de programación que se utilice.

El término **acoplamiento** fue utilizado por primera vez dentro de la Ingeniería de Software cuando la programación estructurada era la norma, y fue definida como "*la medida de fuerza de la asociación establecida por una forma de conexión de un módulo con otro*" [STEVENS1974, ISO24765]. En el contexto del diseño orientado a objetos, el acoplamiento se ve cómo el grado de interconexión de una clase con otra, es decir que el acoplamiento indica la dependencia de una respecto de otra. Una alta dependencia puede decrementar la reutilización de la clase e incrementar el esfuerzo de mantenimiento, ya que los cambios en una clase pueden afectar a las clases dependientes y, en consecuencia, producirse un efecto en cascada. Así, cuanto más

fuerte es el acoplamiento entre los componentes de un sistema, más difícil resultará su comprensión, modificación y corrección y, por lo tanto, más costoso su mantenimiento.

Para reducir el acoplamiento, resulta necesario definir módulos con una alta **cohesión**, la cual se la define "*como un indicador cualitativo del grado en el que un módulo se centra en hacer una sola cosa*" [PRESSMAN2010, ISO24765]. De idéntica manera, el acoplamiento entre clases se puede reducir promoviendo la cohesividad de las mismas. Una clase es cohesiva si la asociación de los elementos declarados en la clase se centran en llevar a cabo una única tarea.

VI.2 Métricas

Las métricas de software se han convertido en un elemento esencial al medir la calidad de los proyectos de software. En especial, las métricas de acoplamiento y de cohesión juegan un papel importante en la evaluación cuantitativa y la mejora de los atributos de calidad internos de los productos de software.

A lo largo de la literatura se han introducido diferente conjuntos de métricas tanto para la cohesión y el acoplamiento, siendo **LCOM4** y **CBO** las métricas seleccionadas para medirlos en el presente trabajo. La importancia de la selección de estas métricas queda evidenciada en [BRIAND2000], en donde se analiza la relación entre estas y la **propensión al fallo** en las clases de un programa. Puntualmente, se encuentra evidencia de que ante el **aumento del acoplamiento**, o la **disminución de la cohesión** en una clase, el número de fallos o errores encontrados en esta tiende a aumentar. De esta forma, resulta patente que cualquier mejora que pudiera hacerse en este sentido, repercutirá positivamente en la calidad de los sistemas resultantes.

VI.2.1 Métrica para la Cohesión de Clases

Para la medición de la cohesión de los componentes y clases su utilizará la métrica **LCOM4** (*Lack of Cohesion of Methods*, *Carencia de Métodos Cohesivos*), la cual es definida por [HITZ1995], tomando como base la métrica LCOM previamente propuesta por [CHIDAMBER1991].

Esta métrica calcula el número de **componentes conectados** en una clase, entendiéndose un componente conectado como un *conjunto de métodos y variables de clase relacionados*. Como regla general, debería haber uno y sólo un componente por clase. De haber dos o más, entonces la clase debería ser dividida en tantas subclases como componentes hayan sido detectados.

Dos métodos, `A` y `B`, están relacionados si se cumple algunas de las siguientes condiciones:

1. Ambos acceden o utilizan la misma variable de clase
2. `A` invoca a `B`, o `B` invoca a `A`

De esta manera, el valor `LCOM4` es igual a la cantidad de grupos de métodos conectados.

- Si `LCOM4 = 1`, entonces la clase es cohesiva.
- Si `LCOM4 >= 2`, entonces indica un problema de que la clase tiene más de una responsabilidad y debería ser dividida.
- Si `LCOM4 = 0`, indica que la clase no contiene método alguno (no tiene responsabilidad) y se trata de un contenedor de datos.

En la Figura 6.1 que aparece a continuación se presenta un ejemplo de un caso con `LCOM4 = 1`, y otro con `LCOM4 = 2`. En esta Figura, los **métodos** se representa con **rectángulos verdes**, y las **variables de la clase** mediante **círculos azules**.

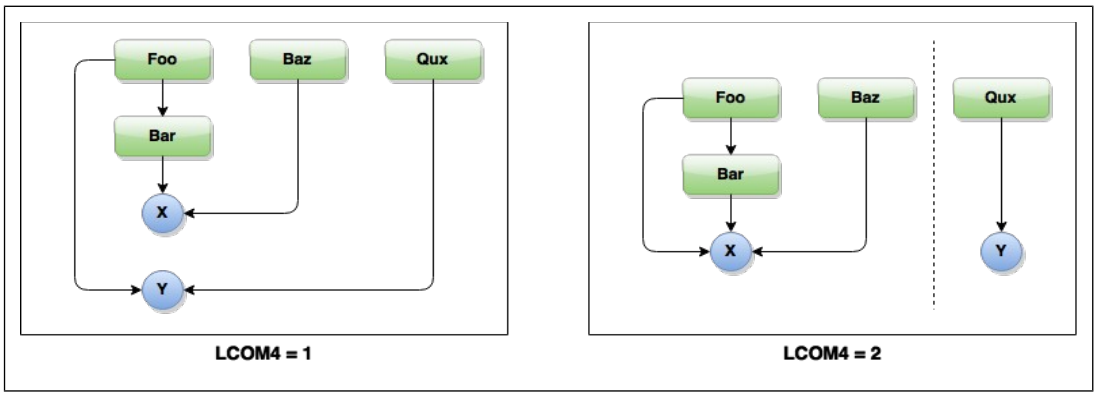


Figura 6.1. Cálculo de LCOM4.

Cabe notar que la presencia de un valor `LCOM4 >= 2` no necesariamente requiere de una refactorización de la clase. Este indicador alerta a los programadores de un posible problema en la clase. Queda a su criterio determinar si se trata o no de error de diseño, y si merece o no ser resuelto.

VI.2.2 Métrica para el Acoplamiento entre Clases

La evaluación del acoplamiento entre clases de realizará mediante la métrica **CBO** (*Coupling Between Object Classes*, *Acoplamiento Entre Objetos de Clases*) [CHIDAMBER1991], la cual se obtiene mediante el conteo de las clases que una clase utiliza o accede, sumado a la cantidad de clases que la referencian a esta. Si las clases se referencian mutuamente, solo se cuenta una sola vez. Como ejemplo, se presenta el diagrama de la Figura 6.2.

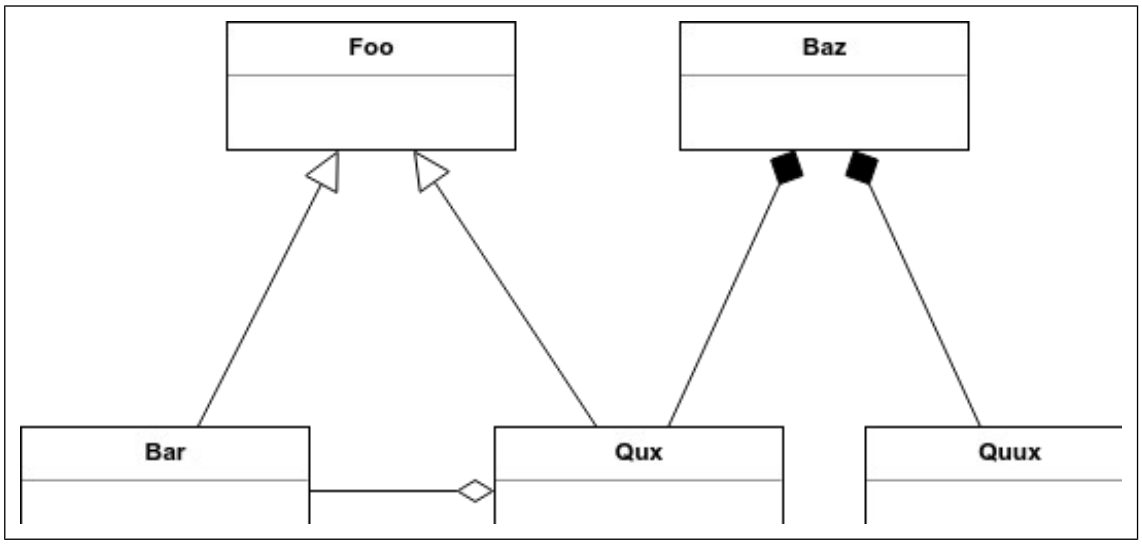


Figura 6.2. Ejemplo para el Cálculo de CBO.

La Tabla 6.1 que aparece a continuación lista los valores de *CBO* obtenidos con dicho cálculo.

Clase	CBO
`Foo`	2
`Bar`	2
`Baz`	2
`Qux`	3
`Quux`	1

Tabla 6.1. Valores CBO obtenidos para cada clase en el ejemplo.

Un número bajo es una buena señal, mientras que un número alto un indicador de peligro. El acoplamiento entre clases ha probado ser efectivo al momento de predecir los fallos en los sistemas, y algunos estudios han demostrado que un valor menor o igual a **9** representa el límite más eficiente para el *CBO*.

VI.3 Cálculo de las Métricas

Las métricas arriba presentadas están pensadas para ser aplicadas sobre una clase puntual, por lo cual, en este proyecto, serán calculadas sobre las clases utilizadas como *interfaces* para **Jess** y el **Jesslet** (ver capítulo VII). La Figura 6.3 ilustra la situación descrita.

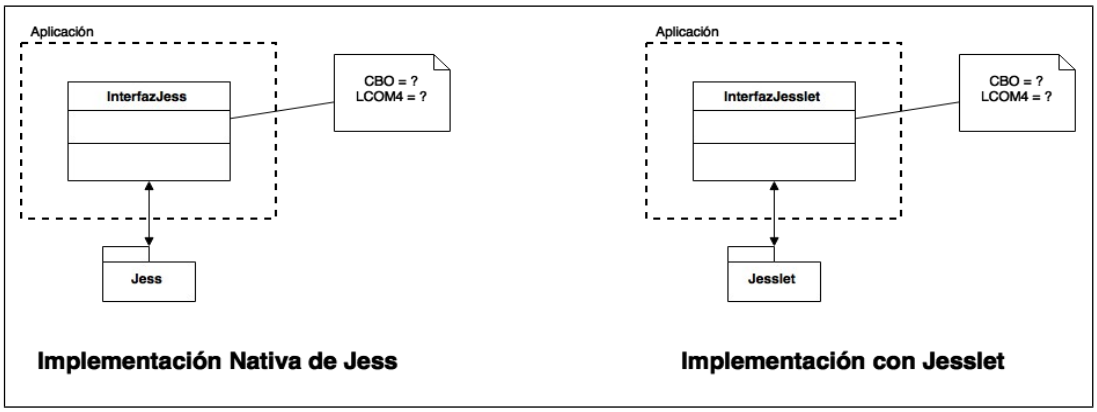


Figura 6.3. Cálculo de las Métricas en el Proyecto.

Una vez programadas las clases y superadas satisfactoriamente todas las pruebas unitarias automatizadas, estos cálculos serán realizados de manera **manual**. El motivo principal por el cual estos valores serán obtenidos de esta manera es que no se ha podido

encontrar una herramienta automática que arroje resultados concluyentes o acordes a lo esperado.

En el caso de *LCOM4*, con estas herramientas, se obtiene un valor cero (`0`) debido a que los métodos implementados mediante una anotación `@Override` son simplemente ignorados. Con esto, la aplicación de un cálculo automatizado de *LCOM4*, para este caso, resulta inviable.

Por otro lado, de acuerdo a la investigación realizada, la métrica *CBO* ha sido "*interpretada*" de varias maneras desde su propuesta original por [CHIDAMBER1991]. Por esta razón, muchas herramientas automáticas, habiendo aplicado una interpretación arbitraria, arrojan valores distintos para la misma clase.

VI.4 Resumen

En esta parte se han presentado las métricas utilizadas para evaluar el resultado de la incorporación de las capacidades de un *sistema experto* dentro de otro sistema mediante la implementación del *Jesslet*. Tanto **LCOM4** (métrica para la *cohesión*) como **CBO** (métrica para el *acoplamiento*) fueron presentadas mediante ejemplos sencillos, y luego se introdujo la forma en la que estas serían luego utilizadas para analizar el impacto del *Jesslet*. La importancia de estas mediciones está en que permiten obtener un resultado firme y cuantitativo del impacto real del uso del servicio web en el diseño y construcción de otro sistema.

Capítulo VII Jesslet

En este capítulo se documenta el proceso de desarrollo del servicio web **Jesslet**, durante el cual se especifica y programa un servicio web sobre Jess con métodos bien definidos, que permiten gestionar las reglas de inferencia y los hechos de un sistema experto, y ejecutarlo para recuperar sus inferencias. Para dicho desarrollo se utiliza el enfoque propuesto por las prácticas de la **Programación Extrema**, adaptado y ajustado al contexto de **un único programador** (ver capítulo V).

VII.1 Fase de Exploración

En esta primera fase, se procedió a establecer el alcance general del proyecto, las herramientas, pautas, restricciones, etc. que se utilizaron a lo largo del proceso, así también como las primeras historias de usuario necesarias para la siguiente fase.

VII.1.1 Alcance

Para este trabajo el alcance del prototipo del Jesslet fue definido como sigue:

- Podrá ser incorporado o utilizado dentro de otro sistema de manera sencilla y sin la necesidad de contar con software o componentes especiales.
- Posibilitará el trabajo con Jess y los sistemas expertos creados con éste desde diferentes plataformas, sistemas y dispositivos de manera concurrente.
- Será un **desarrollo abierto** para su estudio y posible mejora posterior por parte de estudiantes e investigadores.

Más específicamente, los métodos que componen el contrato del Jesslet permitirán crear y ejecutar un sistema experto. A su vez, proveerá de funciones que darán la posibilidad de agregar, actualizar, eliminar y recuperar las reglas de inferencia y los datos de estos sistemas. Este servicio web será programado utilizando el lenguaje Java, e incorporará como un componente a la librería de *Jess*.

Debido al funcionamiento inherente de los servicios web, se ha limitado el conjunto de sistemas que pueden ser representados con el Jesslet a **sistemas expertos de universo cerrado** (ver capítulo II).

VII.1.2 Herramientas

El Jesslet fue desarrollado utilizando el **lenguaje de Programación Java**, el cual presenta un gran soporte para el desarrollo de servicios web.

Se utilizó como base el *shell Jess*, debido la capacidad que presenta de poder ser utilizado como una librería dentro de otra aplicación Java, lo cual lo hace adecuado para ser tomado como shell base para el servicio web. Además, y no menos importante, su API está adecuadamente documentada y es posible obtener una licencia para uso académico con sólo solicitarla.

Cada historia de usuario fue desarrollada mediante el enfoque del **desarrollo guiado por las pruebas** (ver capítulo V), utilizando la herramienta **JUnit**1 para realizar pruebas de unidad de manera automatizada sobre Java. Algunas historias de usuario fueron trabajadas mediante mapas mentales, gráficos y diagramas UML. Si bien la utilización de estas herramientas parece contradictoria con el principio ágil de la poca o nula documentación, se debe considerar que la XP no hace hincapié en ningún tipo de modelo o representación formal de los sistemas o parte de ellos. Entre sus lineamientos sólo enuncia que los programadores y actores del proyecto deben utilizar aquello que les resulte conveniente o consideren mejor para facilitar la comunicación entre sus miembros. En el contexto del desarrollo del *Jesslet*, la representación de ciertos patrones de diseño orientados a objetos mediante estos diagramas fueron de gran utilidad para poder analizar el funcionamiento del sistema previo a su programación. Puntualmente, través de los diagramas de secuencia se pudieron trazar de manera precoz algunos de los métodos e interfaces de las clases, cuyas interconexiones fueron determinantes en el diseño del sistema final.

A lo largo del desarrollo del Jesslet (y de este documento) se utilizó **Mercurial** para el control de versiones del código fuente y del contenido, utilizando como repositorio central el servicio gratuito provisto por **BitBucket**. Este repositorio puede

encontrarse en https://bitbucket.org/rgmiranda/jesslet, en el cual pueden apreciarse todos los cambios por los que ha ido pasando el sistema.

Por último, para la escritura del código se utilizó como recomendación el **estándar de codificación** propuesto por **Sun Microsystems** para el lenguaje Java [SUN1997].

VII.2 Fase de Planificación y Plan de Lanzamiento

Los requisitos fueron planteados mediante la definición de **historias de usuarios**. Cada historia de usuario fue escrita en tarjetas o fichas haciendo uso de la aplicación gratuita Trello[1]. Una vez completada la lista de historias, se procedió a la creación de un plan de trabajo mediante su priorización, la cual se lista a continuación:

1. Servicio web contenedor de los métodos
2. Método para agregar una regla de inferencias
3. Método para recuperar una regla de inferencias
4. Método para recuperar todas las reglas de inferencias
5. Método para eliminar una regla de inferencia
6. Método para agregar un hecho (**Fact**)
7. Método para eliminar un hecho
8. Método para recuperar un hecho
9. Método para recuperar todos los hechos
10. Método para restablecer los hechos (**reset**)
11. Método para ejecutar el sistema (**run**)
12. Soportar múltiples sistemas (proyectos)
13. Manejo de un archivo de configuración central

1. Trello es una aplicación para la administración de proyectos. El proyecto del Jesslet puede encontrarse en la dirección https://trello.com/b/sTFkSF16.

VII.3 Fase de Iteraciones

A continuación se presenta la especificación de las historias de usuarios, los ciclos de desarrollo y las historias abarcadas en cada uno de ellos. Cabe notar que algunas historias impactaron o modificaron el funcionamiento de otras. Lo que se describe a continuación son las historias terminadas en su totalidad; no se muestran los cambios que se les pudieran haber realizado a medida que eran desarrolladas las restantes historias. La ejecución casi constante de las pruebas automatizadas ayudó a mantener la consistencia en las interfaces de las clases y sus valores esperados.

VII.3.1 Primer Ciclo

El primer ciclo abarcó la **creación del servicio web** inicial sobre el cual se ejecutan todos los métodos y procedimientos que se invoquen sobre el *Jesslet*. La estimación del tiempo que insumiría la creación de este componente provocó que nacieran tres historias de usuario a partir de la original:

1. Definición del contrato del servicio.
2. Implementación del contrato
3. Publicación del servicio.

VII.3.1.1 Historia de Usuario: *Definición del Contrato del Servicio*

El primer paso en la creación del servicio web es la definición de la **interfaz** (`Interface`), la cual establece la firma de cada uno de los métodos que integran el contrato del servicio web. Esta interfaz es utilizada por las bibliotecas de la plataforma Java para generar el documento `WSDL` (ver capítulo IV) necesario para ser consumido por los clientes.

La siguiente Figura presenta la interfaz y su relación con el servicio web.

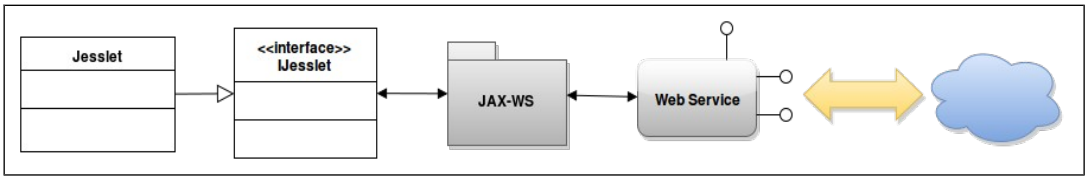


Figura 7.1. La interfaz del servicio

Esta historia de usuario está centrada exclusivamente en la parte de la interfaz `<<Interface>> IJesslet`. En este punto, aún no se tenían firmemente establecidos los tipos de parámetros que cada uno de ellos recibirían, ni los valores que deberían devolver. A medida que se avanzó sobre el desarrollo de las restantes historias de usuario, y se obtuvo un mayor conocimiento de las mismas, las firmas de los métodos fueron actualizadas oportunamente. En el fragmento de código que se muestra a continuación se encuentra una aproximación inicial de los métodos disponibles.

```
public interface IJesslet {
    void AddRule();
    void RemoveRule();
    void GetRule();
    void GetAllRules();
    void AssertFact();
    void RemoveFact();
    void GetFact();
    void GetAllFacts();
    void Reset();
    void Run();
}
```

VII.3.1.2 Historia de Usuario *Implementación del Contrato*

La implementación del contrato o interfaz implicó la creación de la clase `Jesslet` como una implementación de la interfaz `IJesslet`. Cabe mencionar que ninguno de los métodos definidos en la interfaz, en este punto, poseía implementación alguna. Inicialmente, todas estas funciones sólo eran marcadores de posición para las implementaciones que se realizarían al trabajar con las otras historias de usuario en las iteraciones subsecuentes.

Este fragmento muestra parte de la clase `Jesslet` en su estado incipiente.

```
public class Jesslet implements IJesslet {
    @Override
    public void AddRule() {
        throw new UnsupportedOperationException();
    }

    @Override
```

```
    public void RemoveRule() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void GetRule() {
        throw new UnsupportedOperationException();
    }

    // ...
}
```

VII.3.1.3 Historia de Usuario: *Publicación del Servicio*

La publicación del servicio implicó la creación de un **demonio** para escuchar las invocaciones mediante SOAP que se realicen sobre los métodos de la interfaz. Los cambios introducidos por esta historia, como se ve en el diagrama de la Figura 7.1, implicó la utilización del paquete `JAX-WS`, y la creación del servicio mediante este. Esta biblioteca utiliza los métodos públicos definidos en la interfaz para generar automáticamente el documento WSDL con los métodos SOAP disponibles, sus parámetros, y los valores devueltos. Para conseguir esto se establecen una serie de **anotaciones**[1] en la interfaz `IJesslet` como aparece a continuación.

```
@WebService(name = "JessletService", targetNamespace =
JessletConstants.WS_TARGETNAMESPACE, portName =
"JessletServicePort")
@SOAPBinding(style = Style.RPC, use = Use.LITERAL,
parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
public interface IJesslet {

    @WebMethod
    void AddRule();

    @WebMethod
    void RemoveRule();

    @WebMethod
    void GetRule();
```

1. Una anotación de Java es una forma de añadir metadatos, al código fuente, que están disponibles para la aplicación en tiempo de ejecución. Es una forma de programación declarativa dentro de Java.

```
    // ...
}
```

Existen dos maneras de publicar un servicio web en Java:

1. Utilizar un servidor de aplicaciones (por ejemplo *Tomcat* o *Glassfish*) para que escuchar las solicitudes HTTP y SOAP.
2. Crear un servidor *ad-hoc* mediante *JAX-WS*.

Por razones de simplicidad y facilidad de puesta a punto, se optó por el segundo camino, obteniéndose una clase principal `Main` en donde se crea la instancia del servicio y se define la URL sobre la cual escuchará los pedidos SOAP (`http://127.0.0.1/jesslet/`).

```
public class Main {

    public static void main(String[] args) {
        try {
            System.out.println("Iniciando el servicio...");
            Endpoint end =
Endpoint.publish("http://127.0.0.1/jesslet", new Jesslet());
            System.out.println("Servicio Iniciado...");
            System.out.println("Presione ENTER para
cerrar...");
            System.in.read();
            System.out.println("Cerrando el servicio...");
            end.stop();
        } catch (IOException e) {
            System.out.println("ERROR: " + e.getMessage());
        } catch (JessletFault ex) {

Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
}
```

VII.3.2 Segundo Ciclo

Este ciclo incluyó la implementación del **método para la adición de una nueva regla de inferencia** en la base de conocimientos, lo cual requirió de la creación de una

clase `Rule` a modo de contenedor de todos los datos relativos a una regla de inferencia. El método `AddRule` recibe como parámetro una instancia de dicha clase, la cual representa la regla de inferencia a ser creada, y devuelve un valor `boolean` indicando si la operación ha resultado exitosa o no. En la nueva firma del método se establece que, eventualmente, este podría lanzar una excepción del tipo `ServerFault`. La cual sirve para retornar errores (`SOAP Faults`) hacia el cliente del servicio.

El fragmento siguiente presenta la **firma** de este método de la interfaz `IJesslet`, en la cual se especifica que este método tiene la posibilidad de lanzar una excepción del tipo `ServerFault`.

```
boolean AddRule(Rule rule) throws ServerFault;
```

Debido a la complejidad del desarrollo que se debía afrontar en esta historia, tuvo que se fraccionada en dos:

1. Representación de las Reglas de Inferencia
2. Creación de una Regla de Inferencia en el Shell

VII.3.2.1 Historia de Usuario: *Representación de las Reglas de Inferencia*

La clase `Rule` representa una regla de inferencia en el *Jess*, y, al igual que su par en el *shell*, está compuesta por un **nombre**, una **descripción**, el **antecedente**, (condiciones o restricciones de activación), y el **consecuente** (acciones a realizarse cuando esta sea disparada) (ver capítulo II).

```
(nombre-regla "Descripción de la regla"
    ; Condiciones antecedentes...
    =>
    ; Acciones consecuentes...
)
```

De esta forma, `Rule` fue creada con cuatro atributos: `name`, `description`, `conditions` y `actions`. El campo `name` se corresponde con `nombre-regla` y `description` con la documentación de la regla que aparece entre comillas. Ambos

campos son del tipo *cadena de caracteres* (`String`), mientras que los campos `conditions` y `actions` son arreglos que representan los conjuntos de antecedentes y consecuentes respectivamente.

VII.3.2.1.1 Condiciones de una Regla

Cada **condición** en una regla es una **restricción** basada en la existencia (o no) de un hecho con determinadas características. Para el desarrollo del Jesslet, se tuvo que restringir la sintaxis de estas condiciones a un formato que pudiese, por un lado, ser lo suficientemente **flexible** para permitir el trabajo adecuado con las reglas de inferencia, y por otro, lo suficientemente **abstracto** para ser representado mediante una clase serializable a *XML* para su envío a través de un mensaje *SOAP*.

Debido a estas limitaciones, se optó por soportar sólo condiciones con **hechos ordenados con un único valor**, por ejemplo:

```
(encendido TRUE)
(semaforo "VERDE")
(tension 220)
```

Los hechos no ordenados y ordenados con múltiples valores habrían complejizado demasiado la interfaz del servicio, y se consideró que no era lo ideal para una primera aproximación.

El atributo `conditions` en la clase `Rule` es un arreglo de objetos del tipo `FactCondition`. Cada elemento de este arreglo se corresponde con una condición para la regla, y posee cuatro atributos:

1. `factName`: nombre del hecho.
2. `factAlias`: nombre con el que será llamada la variable asociada al valor del hecho, y al hecho en sí mismo siempre que no esté negado (`negated = false`).
3. `negated`: valor booleano que indica si el hecho está negado (`not`) o no.
4. `filter`: objeto del tipo `Expression` que establece una condición sobre el valor del hecho.

La clase `Expression` ha sido diseñada para representar una expresión en Jess mediante una **estructura de árbol**. Por ejemplo `(* (+ ?x 34) (- ?y -40))` se traduciría en el árbol que aparece en el Figura 7.2.

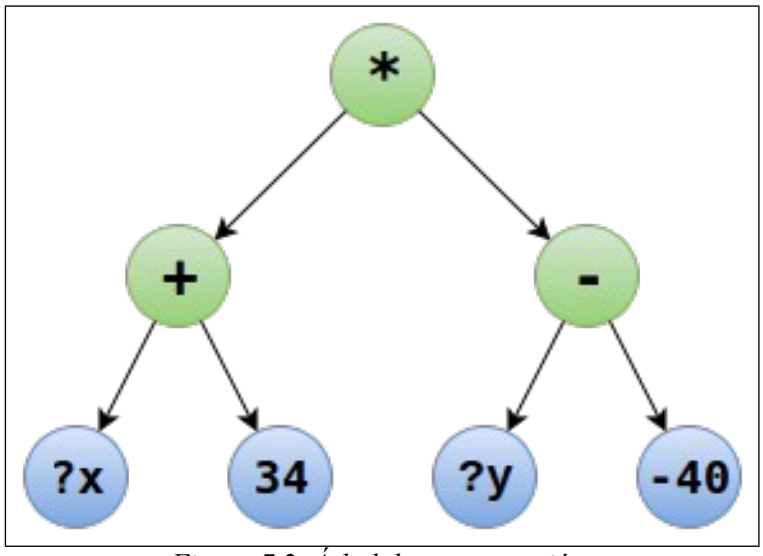


Figura 7.2. Árbol de una expresión.

`Expression` es una **clase abstracta** dos clases:

- `LiteralExpression`, la cual se usa para expresiones literales, por ejemplo `"una cadena"`, `256`, `3.1459`.
- `OperationExpression`, la cual representa expresiones basadas en una función u operación, por ejemplo `(= 90 90.0)`, `(and (<= 0 ?var) (<= ?var 35535))`.

Una **expresión literal** se utiliza para presentar un valor literal final, como una cadena de caracteres o un número. `LiteralExpression` tiene dos campos para tal fin: `value` y `valueType`. Este último determina el tipo de valor con el cual será cargado `value`. Por ejemplo, si se tiene una expresión literal con un valor `-128`, y el tipo de valor fuese `STRING`, el valor asignado sería `"-128"`. Por otro lado, si fuese `INTEGER`, entonces se obtendría sencillamente `-128`. Para poder hacer referencia al valor del hecho en los antecedentes de la regla de inferencia, hay que utilizar el tipo de valor `REFERENCE`, y el valor debe iniciar con el carácter `@`. Cuando la condición se haga

efectiva en el motor de inferencias, este valor será modificado cambiando el `@` por `?`, y agregando el sufijo `-fact`. Por ejemplo, `@alias` será convertido en `?alias-fact`.

Una `OperationExpression`, o **expresión de operación**, se encuentra compuesta, a si mismo, por un conjunto de objetos del tipo `Expression` (los *operandos*), utilizando de esta manera, un **patrón Composite**.

VII.3.2.1.2 Acciones en una Regla

Se entiende por **acciones** al conjunto de sentencias que aparecen en el consecuente de una regla de inferencia, y en `Rule`, están representadas por el arreglo `actions`. Cada elemento de este arreglo es una instancia de una clase que extiende a la clase *abstracta* `Action`.

Un inconveniente encontrado a la hora de representar las acciones fue que en esta parte de la regla es posible escribir prácticamente cualquier función o código fuente en lenguaje Jess. A los fines de evitar el envío de código fuente a través del servicio web, se tuvo que reducir las acciones que se pueden realizar en el consecuente de una regla de inferencia a tres tipos: **creación de un hecho**, **eliminación de un hecho**, **recomendación al operador**.

Se entiende por **acciones** al conjunto de sentencias que aparecen en el consecuente de una regla de inferencia, y en `Rule`, están representadas por el arreglo `actions`. Cada elemento de este arreglo es una instancia de una clase que extiende a la clase *abstracta* `Action`.

Un inconveniente encontrado a la hora de representar las acciones fue que en esta parte de la regla es posible escribir prácticamente cualquier función o código fuente en lenguaje Jess. A los fines de evitar el envío de código fuente a través del servicio web, se tuvo que reducir las acciones que se pueden realizar en el consecuente de una regla de inferencia a tres tipos: **creación de un hecho**, **eliminación de un hecho**, **recomendación al operador**.

A partir de `Action`, heredan tres clases que se corresponden con estas tres acciones:

- `AssertAction`: para la **creación de un hecho** en la memoria de trabajo.
- `RemoveAction`: para la **eliminación de un hecho** de la memoria de trabajo.
- `RecommendAction`: para la presentación de un **mensaje para el operador**.

`AssertAction` posee un único atributo `fact` del tipo `SimpleFact`, el cual simboliza el hecho a ser agregado.

`SimpleFact` está compuesta por el identificador del hecho en la memoria de trabajo (`id`), el nombre del hecho (`name`), una breve descripción (`description`), y el valor del hecho a agregarse (`value`). Este valor es una instancia de una clase que hereda de `Expression`.

La acción `RemoveAction` sólo se compone del atributo `factId`, el cual guarda el identificador del hecho a ser eliminado de la memoria de trabajo. Este identificador es una cadena de caracteres, y debe ser un identificador válido para un alias de un hecho en la regla. Este identificador debe cumplir con las reglas antes detalladas para un alias en una expresión literal, es decir, debe iniciar con el carácter `@`, el cual será reemplazado por `?` y se la agregará el sufijo `-fact`.

Una acción `RecommendAction` genera el código que imprime un mensaje al operador del sistema. La única propiedad que tiene esta clase es `message`, y se trata de cadena de caracteres a ser presentada. Por ejemplo:

```
(printout t "Esta es la cadena de caracteres en 'message'."
crlf)
```

El parámetro `t`, que aparece primero en la llamada a la función `printout`, es el buffer de salida sobre el cual se va a imprimir el mensaje, y `t` es el buffer por defecto de Jess. En el contexto del Jesslet, este buffer cambia por otro desde el cual se pueden recuperar los mensajes para ser devueltos como resultado de una ejecución a través de SOAP.

VII.3.2.2 Historia de Usuario: *Creación de una Regla de Inferencia en el Shell*

La creación de la regla de inferencia se hace ejecutando un código en lenguaje Jess, el cual hace uso de la función `defrule` para conseguir esto. La clase `Rule` ha sido dispuesta con un método `getSource`, el cual genera el código fuente en lenguaje Jess correspondiente a la definición de la regla de inferencia requerido por `defrule` para crearla en el motor. La Figura 7.3 presenta un diagrama UML con las clases junto con sus atributos y métodos que intervienen en la generación del código fuente para la definición la regla.

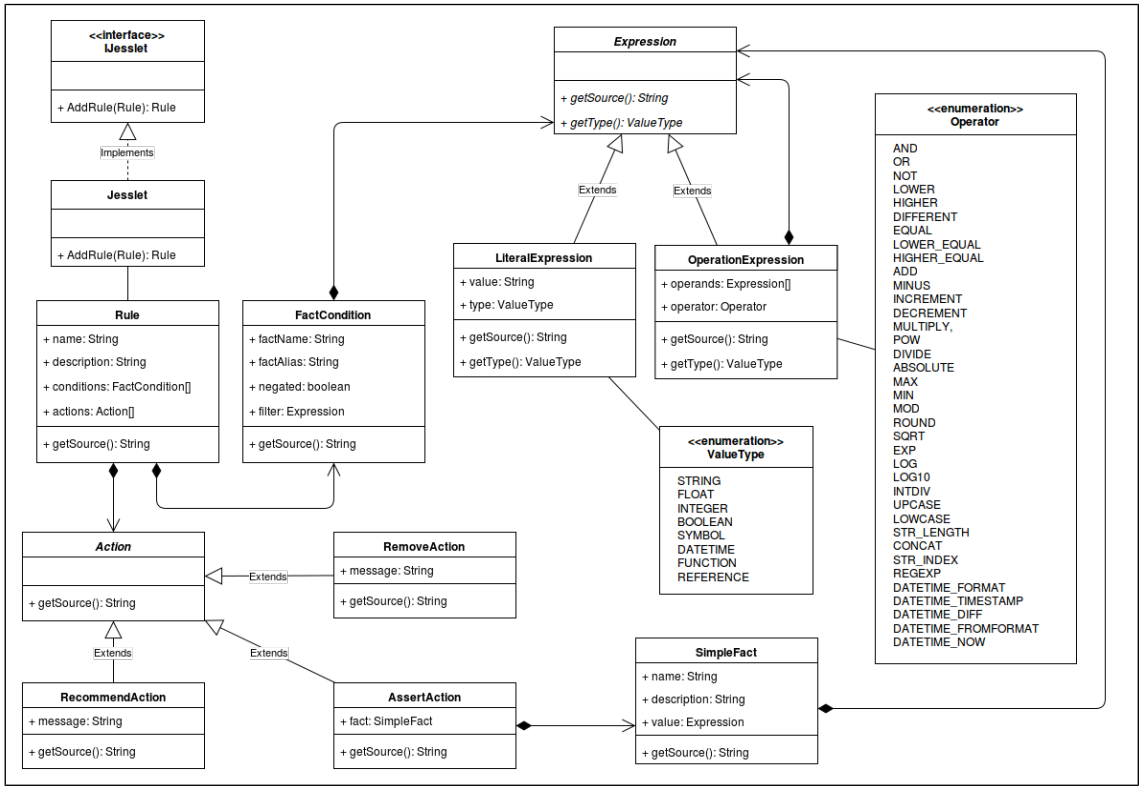


Figura 7.3. Clases del método para agregar una regla de inferencia

Cuando el método `getSource` de `Rule` es invocado, este se encarga de generar el código haciendo uso de los `getSource` de las clases que la componen.

En la Figura 7.4 aparece un diagrama de secuencias de UML en donde se presenta conjuntamente el trabajo de la clase `Rule` para la obtención del código fuente con

`getSource`, desde que se invoca al método `AddRule` en el Jesslet, hasta su implementación en el motor (`Rete`) mediante la función `eval`.

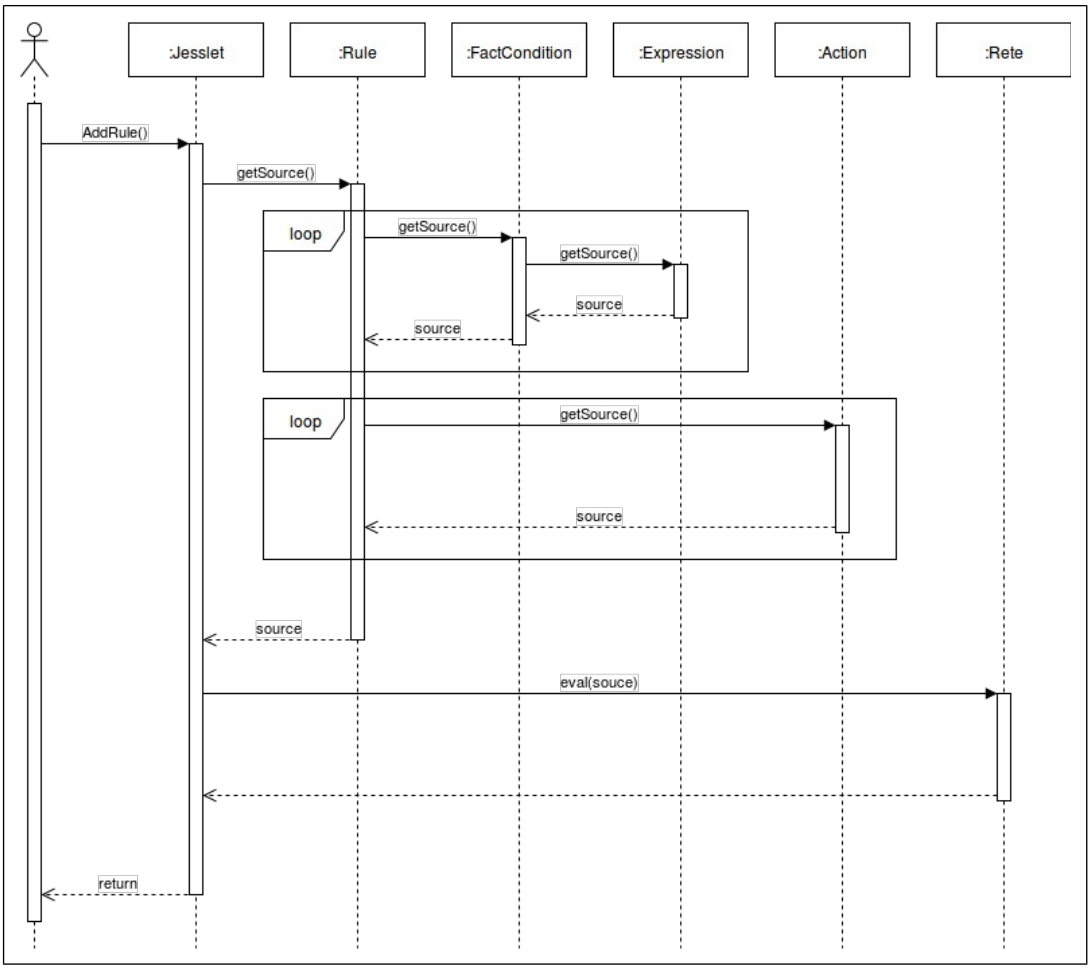


Figura 7.4. Diagrama de Secuencias para Agregar una Regla de Inferencia.

Para las condiciones de la regla, se recorre el arreglo de objetos `FactCondition`, generando una a una las condiciones del antecedente. El código mediante el método `getSource` de esta clase posee el siguiente formato:

```
?<factAlias>-fact <- (<factName> ?<factAlias>-
value&:<filter.getSource()>)
```

Por ejemplo, si se tiene un objeto con los atributos:

```
{
    factName: "hecho",
    factAlias: "h",
```

```
    negated: false,
    filter: null
}
```

entonces se generaría una condición de la forma:

```
?h-fact <- (hecho ?h-value)
```

Si `negated`, por el contrario, tuviese un valor positivo (`true`), entonces el hecho no se dejaría en variable alguna, y el resultado de la función `getSource` cambiaría de esta forma:

```
(not (hecho ?h-value))
```

Análogamente, las acciones en el consecuente también son generadas una a una con el recorrido de su arreglo contenedor. Cada clase hija de `Action` implementa el método abstracto `getSource` utilizado por Rule para generar el código completo de la regla. Para las acciones de **creación de hechos** (`AssertAction`), el código generado es estructurado de esta manera:

```
(assert <fact.getSource()>)
```

Por ejemplo:

```
(assert (presion 101325))
(assert (mensaje "..."))
(assert (ensamblado TRUE))
(assert (total (* 2 (+ ?x-value ?y-value))))
```

Para una acción de **eliminación de un hecho**, se hace uso de `factId` para obtener el código Jess. Por ejemplo, un valor `@alias-del-hecho` resultaría de esta forma:

```
(retract ?alias-del-hecho-fact)
```

La principal ventaja de estructurar las acciones de esta manera es que si en un futuro se requiere una nueva acción, su adición no representará una gran dificultad.

Solamente debe crearse una nueva clase que herede de `Action` e implemente oportunamente el método `getSource`.

VII.3.3 Tercer Ciclo

Durante esta iteración se trabajó sobre los **métodos para recuperar las reglas de inferencia** del shell Jess, y con el **método para eliminar una regla** de la base de conocimientos. El reto principal en este ciclo fue el de traducir los objetos internos de Jess que representan el antecedente y el consecuente hacia objetos del tipo `Rule`.

VII.3.3.1 Historia de Usuario: *Método para Recuperar una Regla de Inferencia*

El método `GetRule` tiene el propósito de recuperar una regla de inferencia, y para tal fin, su firma fue actualizada para recibir como único parámetro el nombre de la regla de inferencia a ser buscada, y devolver un objeto del tipo `Rule` en caso de que la regla haya sido encontrada. En caso contrario, se lanza una excepción `ServerFault`.

```
Rule GetRule(String ruleName) throws ServerFault;
```

Para este método, también se tuvo que modificar la clase `Rule` agregándole un constructor que recibe como parámetro objeto `Defrule`, el cual representa una regla de inferencia en el contexto de Jess. Este objeto es extraído desde la instancia de `Rete`, que representa el motor de Jess en sí. Cuando este constructor es ejecutado, se recuperan todos los objetos `CondicionalElement` (las **condiciones**) y `Funcall` (las **acciones**). Utilizando estos objetos, se crean las instancias de `FactCondition` para las condiciones, y las instancias de `Action` para las acciones en la clase Rule.

Para la creación de cada uno de los objetos `FactCondition` mediante un `ConditionalElement`, se programó un constructor que recibe como parámetro una instancia de esta clase. Este se encarga de determinar qué tipo de condición es, y de la creación del filtro (`Expression`) correspondiente.

Cada `ConditionalElement` está asociado a un conjunto de pruebas (`Test1`), las cuales son los filtros asociados a la condición. Cada objeto `Test1` tiene un valor

asignado (`Value`) con el contenido de la prueba. La clase abstracta `Expression` también fue modificada para dar soporte a esta construcción con un método estático `loadExpression`. Esta función construye una instancia de una de las dos clases que heredan de `Expression` (`LiteralExpression` y `OperationExpression`) usando el parámetro del tipo `Value` asociado a la instancia de `Test1`.

De manera semejante, para poder crear las acciones `Action` para la regla, esta clase fue adicionada con un método estático `loadAction`, el cual utiliza un objeto `Funcall` (cada una las acciones en la regla `Defrule` en Jess) para determinar qué tipo de acción es (agregar, eliminar o recomendar), y construir la instancia correspondiente (`AssertAction`, `RemoveAction` o `RecommendAction`). Para el caso de la creación de un hecho, también se requiere recuperar desde Jess el hecho a ser creado. Este se corresponde con el atributo `fact` de la clase `AssertAction`. Para dar soporte a esta situación, se programó un constructor sobre `SimpleFact` que recibe como parámetro un objeto del tipo `Fact`.

El diagrama en la Figura 6,5 muestra las clases, junto con sus relaciones, intervinientes en el proceso descrito.

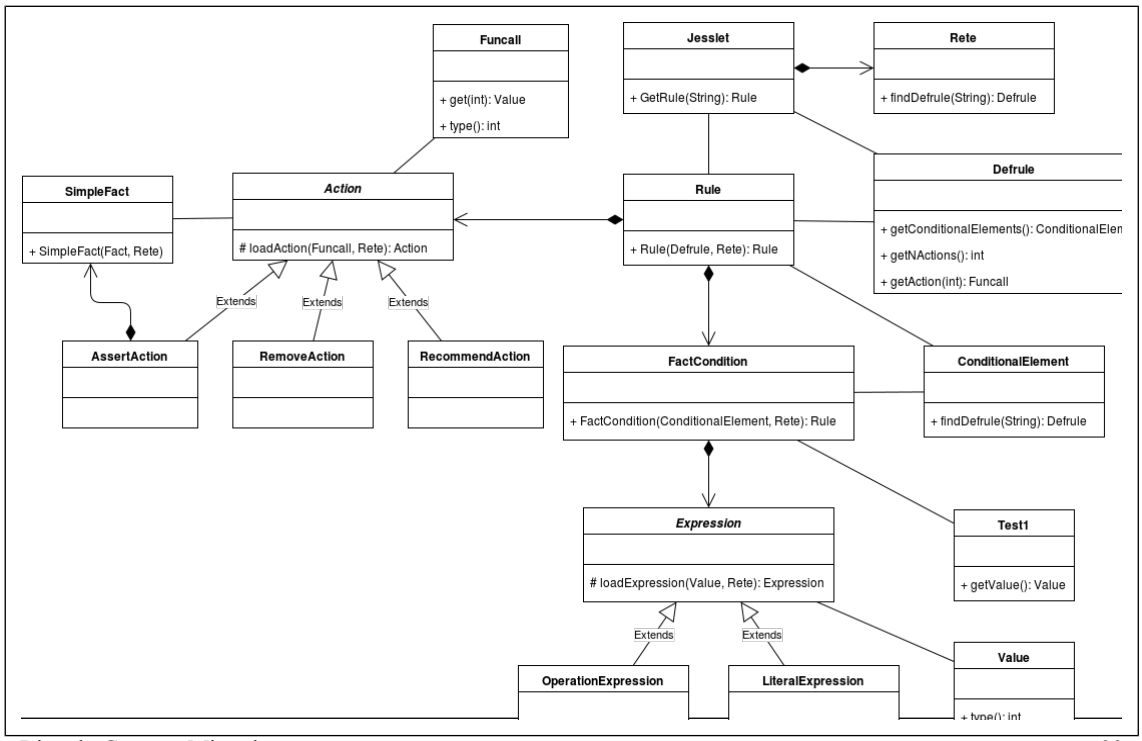


Figura 7.5. Diagrama de Clases para Recuperar una Regla de Inferencia.

Las Figuras 7.6, 7.7 y 7.8 de abajo presentan el diagrama de secuencias de la creación de una clase `Rule`. La primera se enfoca en el constructor de esta clase, mientras que las otras dos ahondan en la creación de las condiciones (`FactCondition`) y las acciones (`Action`).

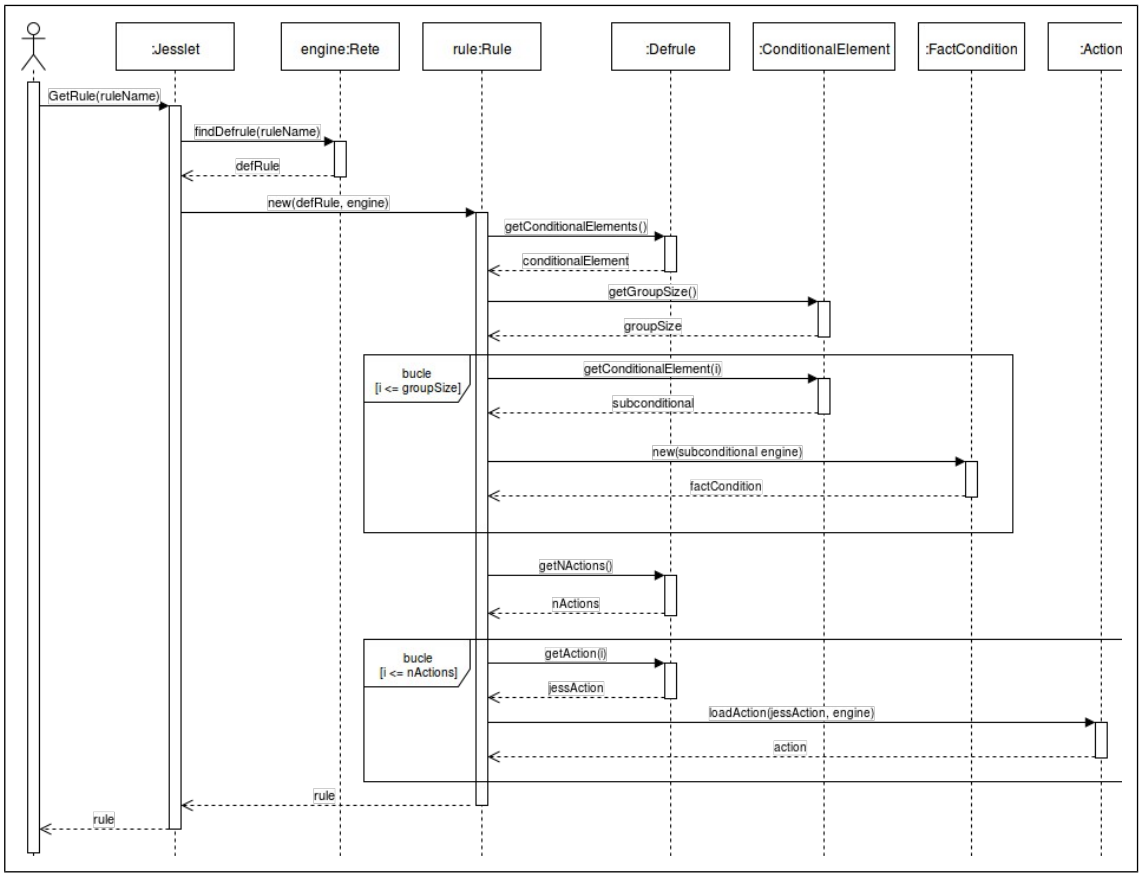


Figura 7.6. Diagrama de Secuencias para Recuperar una Regla de Inferencia.

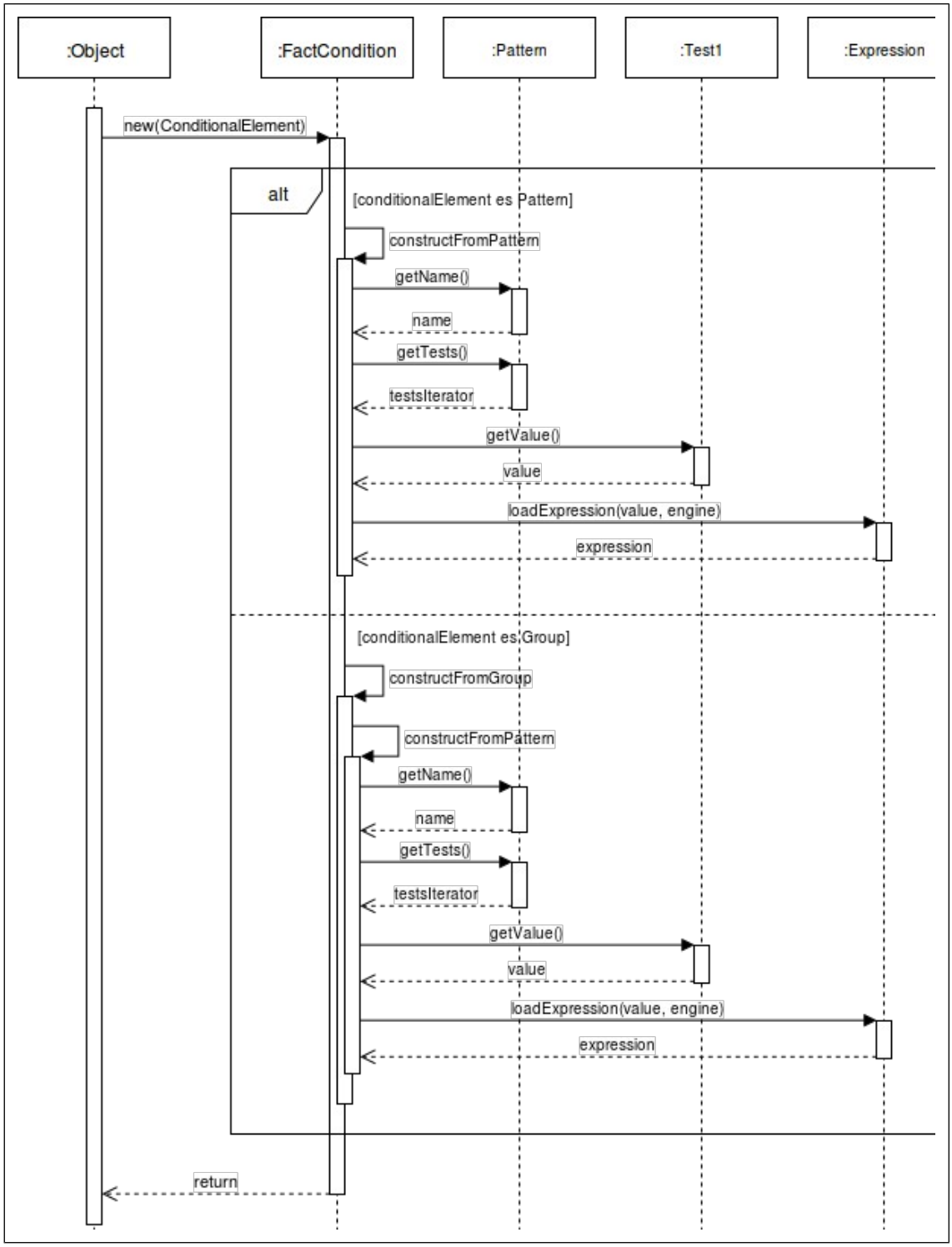


Figura 7.7. Diagrama de Secuencias para Construir una Condición.

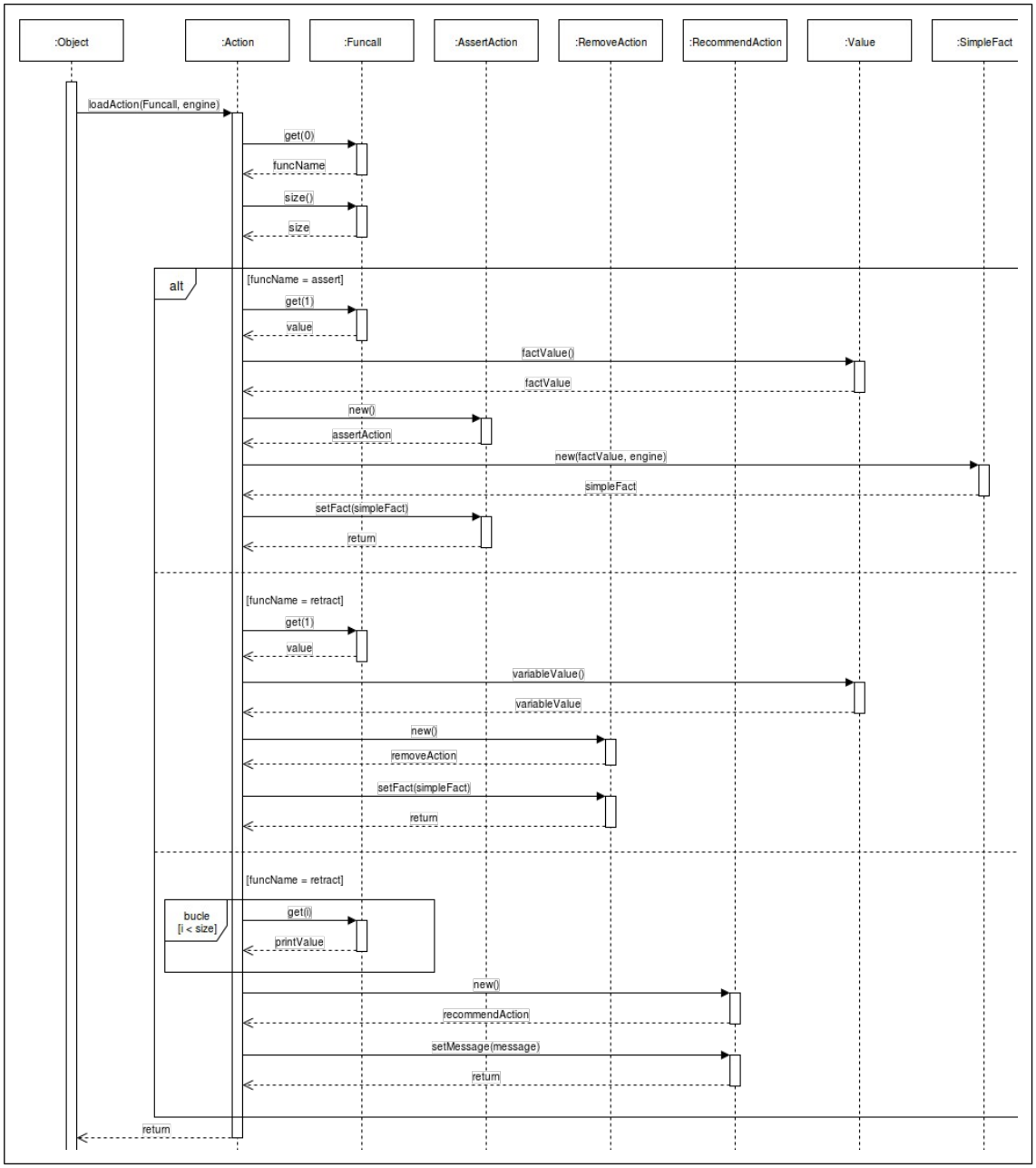


Figura 7.8. Diagrama de Secuencias para Construir una Acción.

VII.3.3.2 Historia de Usuario: *Método para Recuperar Todas las Reglas del Sistema*

Este método funciona de manera similar al desarrollado en la historia anterior, con la diferencia de que en vez de retornar una regla, devuelve todas las reglas encontradas en el sistema como un arreglo de objetos `Rule`.

```
public Rule[] GetAllRules() throws ServerFault;
```

Internamente, este procedimiento itera sobre todas las reglas presentes (`Defrule`), y generar para cada una su correspondiente representación (`Rule`) para ser enviada al cliente. En este caso, se reutiliza el constructor de `Rule` desarrollado en la función anterior. El diagrama de secuencia en la Figura 7.9 presenta brevemente este funcionamiento.

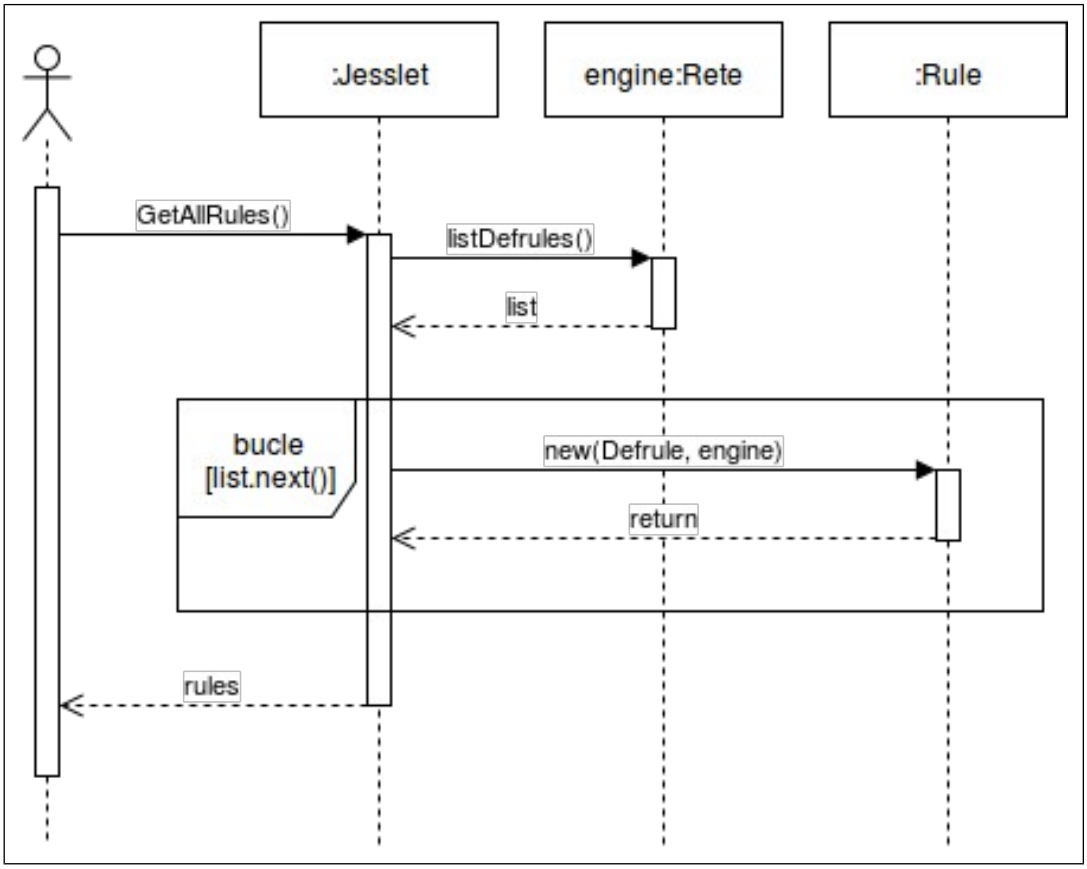


Figura 7.9. Diagrama de Secuencias para Recuperar Todas las Reglas.

VII.3.3.3 Historia de usuario: *Método para Eliminar Una Regla de Inferencia*

Este método recibe el nombre de regla que se quiere borrar como único parámetro, y devuelve un valor booleano: `true` si todo ha resultado correctamente, y `false` en cualquier otro caso.

```
public boolean RemoveRule(String ruleName) throws ServerFault;
```

El funcionamiento de este procedimiento resulta bastante sencillo. Haciendo uso del nombre de la regla, se genera un fragmento de código en lenguaje Jess de la siguiente forma `(undefrule <ruleName>)`, por ejemplo

```
(undefrule regla-1)
(undefrule betelgeuse)
(undefrule aracalacana)
```

Nuevamente, este fragmento se ejecuta sobre el `Rete` con la función `eval`.

VII.3.4 Cuarto Ciclo

Las historias de usuario que fueron abarcadas por este ciclo son las que se corresponden con los métodos para **agregar** y **eliminar** hechos de la memoria de trabajo de Jess.

VII.3.4.1 Historia de Usuario: *Método para Agregar un Hecho (`Fact`)*

El fin de esta función es agregar un hecho a la memoria de trabajo del sistema. La definición en la interfaz `IJesslet` fue alterada para recibir como parámetro un objeto del tipo `SimpleFact`, y, en base a este, crear una instancia de la clase `Fact` de Jess. Ya se trabajó anteriormente con esta clase `SimpleFact` en la historia de usuario "*Representación de las Reglas de Inferencia*" en la primera iteración.

```
public int AssertFact(SimpleFact simpleFact) throws
ServerFault;
```

El diagrama de secuencias que aparece a continuación (Figura 7.10) resume el funcionamiento del método.

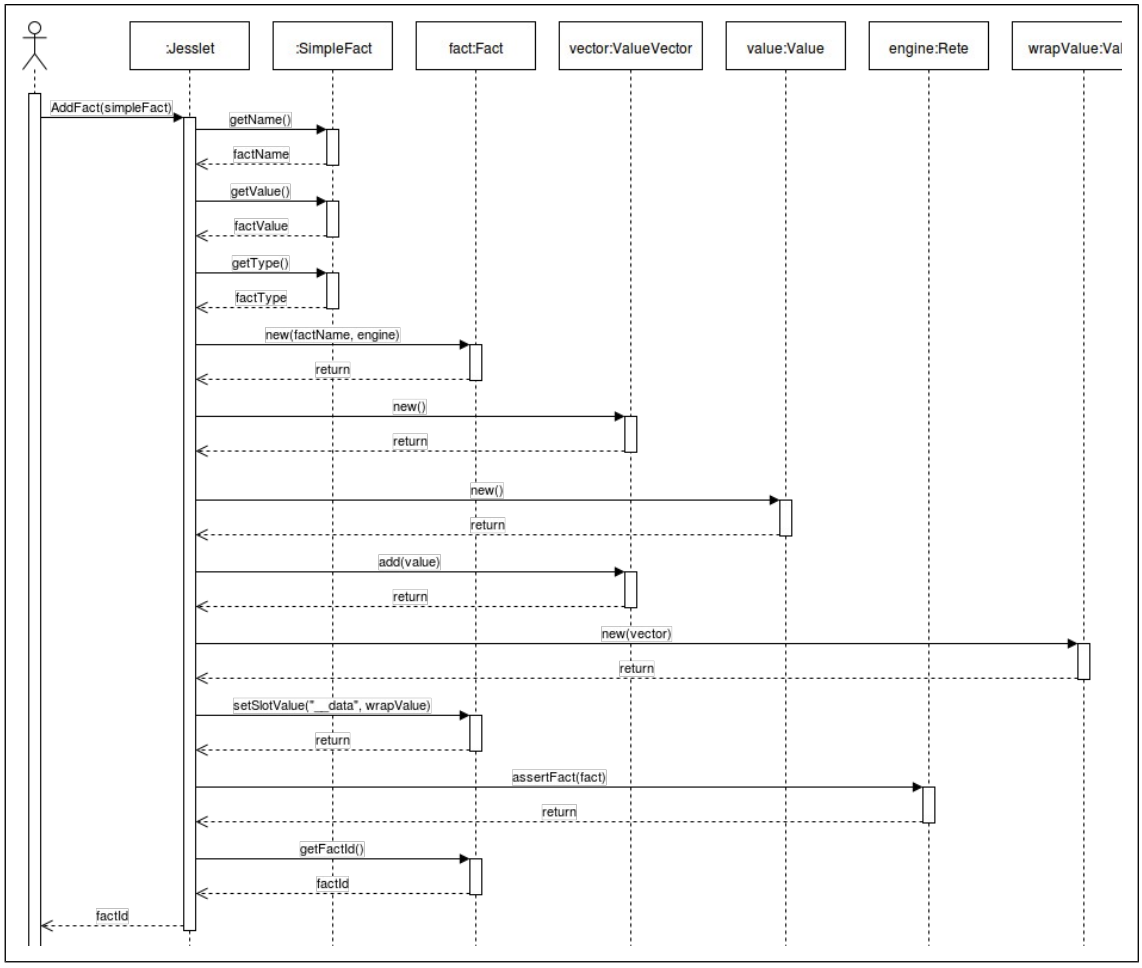


Figura 7.10. Diagrama de Secuencia para Agregar un Hecho.

El abordaje de este requisito se hizo, en un principio, de una forma semejante a la utilizada para agregar una regla de inferencia. El método invocaría a la función `getSource` del parámetro `SimpleFact` recibido, para luego llamar a `eval` en el motor `Rete`. La ventaja de este enfoque es que es posible una abstracción con respecto a los detalles internos del manejo de los hechos en el shell. No obstante, el problema que se descubrió con esta aproximación es que resulta imposible determinar con que *identificador* fue agregado el hecho a la memoria de trabajo. Podría hacerse la presunción de que el nuevo identificador podría ser el mayor de todos, pero debido a que Jess estaría ejecutándose en calidad de un servicio, no hay forma de garantizar esta premisa.

La solución finalmente propuesta es la creación de los hechos nativos (`Fact`) en base a los atributos del parámetro `SimpleFact`. Estos hechos se agregan a la memoria de trabajo asociada al motor usando la función `assert` en la clase `jess.Rete`.

VII.3.4.2 Historia de Usuario: *Método para Eliminar un Hecho*

Este método borra un hecho de la memoria de trabajo de Jess, y devuelve `true` si ha sido satisfactorio, y `false` en otro caso. La firma resultante del método es la siguiente:

```
public boolean RemoveFact(int factId) throws ServerFault;
```

El procedimiento hace uso del método `findFactByID` del motor `jess.Rete` para recuperar el método, y luego invoca a `retract` en caso de que el hecho haya sido encontrado.

VII.3.5 Quinto Ciclo

En esta iteración fueron desarrolladas las historias de usuario correspondientes a los métodos para **recuperar hechos** de la memoria de trabajo, y para **restablecerla** (`reset`).

VII.3.5.1 Historia de Usuario: *Método para Recuperar un Hecho*

Este método recibe un identificador de un hecho, y devuelve un objeto `SimpleFact` correspondiente al hecho. Si el hecho no es encontrado, se lanza una **SoapFault** del tipo `ServerFault`.

```
public SimpleFact GetFact(int factId) throws ServerFault;
```

Este procedimiento reutiliza el constructor programado con anterioridad para la clase `SimpleFact`, en el cual el objeto se construye utilizando una instancia de un hecho `Fact` de Jess.

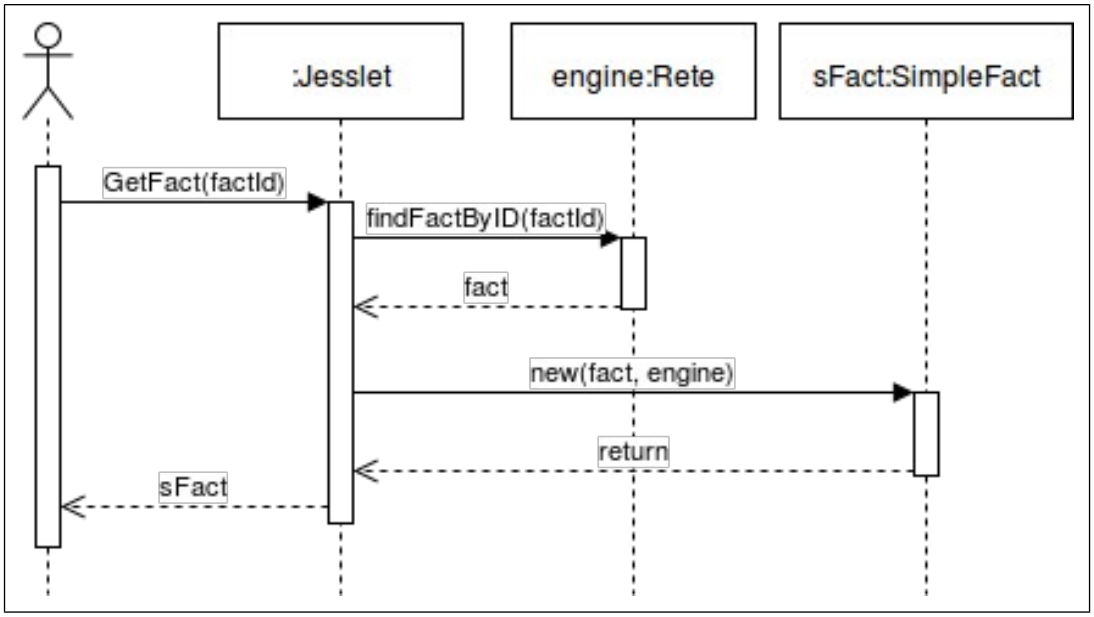


Figura 7.11. Diagrama de Secuencia para Recuperar un Hecho.

VII.3.5.2 Historia de Usuario: *Método para Recuperar Todos los Hechos*

Esta función trabaja de una forma análoga a la anterior, con la diferencia de que retorna un arreglo de objetos `SimpleFact` en lugar de uno solo.

```
public SimpleFact[] GetAllFacts() throws ServerFault;
```

Recupera todos los hechos desde el motor `Rete`, y construye el arreglo usando estos objetos `Fact` encontrados.

VII.3.5.3 Historia de Usuario: *Método para Restablecer los Hechos*

Este método es una suerte de *alias* para el método `reset` de Jess, y tiene el objetivo de eliminar de la memoria de trabajo todos los hechos y crear el hecho inicial `initial-fact` (ver capítulo III). Recordemos que este hecho resulta ser muy importante, debido a que permite la activación de reglas de inferencia cuyas primeras condiciones utilizar la cláusula de no existencia `not`.

En el Jesslet, `Reset` invoca al método `reset` de la clase `Rete`, y devuelve un valor booleano que indica si la operación ha sido exitosa o no.

```
public boolean Reset() throws ServerFault;
```

VII.3.6 Sexto Ciclo

En esta etapa se programó el método del servicio que permite ejecutar el sistema experto con sus reglas de inferencias y hechos en la memoria de trabajo. Cabe recalcar que la ejecución debe hacerse luego de *restablecer* el sistema y cargar los hechos necesarios.

VII.3.6.1 Historia de Usuario: *Método para Ejecutar el Sistema*

El método `Run` permite ejecutar el sistema y devuelve una instancia de la clase `Response`, la cual resume el resultado de la operación.

```
public Response Run() throws ServerFault;
```

Este objeto `Response` se compone de los siguientes atributos:

- `httpCode`: código de resultado de la operación
- `text`: texto con mensajes de resultado
- `recommendations`: listado de recomendaciones emitidas por el sistema
- `ruleStackTrace`: listado de las activaciones de las reglas de inferencia

El `httpCode` es el código de resultado, e indica el resultado de la ejecución del sistema. El valor de esta propiedad se corresponde a uno de los **códigos de estado de HTTP** [W3CHTTP11]. `text` mantiene un mensaje de resultado de la operación, a través del cual se puede obtener más información sobre el resultado final de la operación. Todo mensaje de error puede ser recuperado con esta propiedad. El campo `recommendations` es un arreglo de cadenas de caracteres en el cual cada una de estas es un mensaje de recomendación emitido por el motor de inferencias a medida que es ejecutado. Estas recomendaciones se generan mediante las acciones `RecommendAction` en una regla, las cuales se traducen a una llamada a la función `printout` en Jess. La propiedad `ruleStackTrace` permite recuperar el listado de activaciones por de las reglas de inferencias por las cuales ha ido pasando el sistema a lo largo de su ejecución.

Para poder obtener las recomendaciones y la traza de las activaciones y hechos generados, se deben configurar streams de salida en el motor `Rete` de Jess, e indicarle que los utilice para dicho propósito. Para crear estos streams de salida se programó una clase especial llamada `StringArrayWriter`, la cual se encarga de escribir todo buffer de entrada como un nuevo elemento en un arreglo de cadenas de caracteres. Esta clase hereda de la clase abstracta nativa de Java `Writer`. Con el método `addOutputRouter` de `Rete` se agrega un objeto de este tipo para recuperar todas las salidas. Cuando la ejecución finaliza, se usa la función `getStrings`, el cual devuelve el arreglo de cadenas necesario.

La Figura 7.12 presenta un diagrama de secuencias en donde puede apreciarse con mayor detalle la interacción de las diferentes clases involucradas en este método.

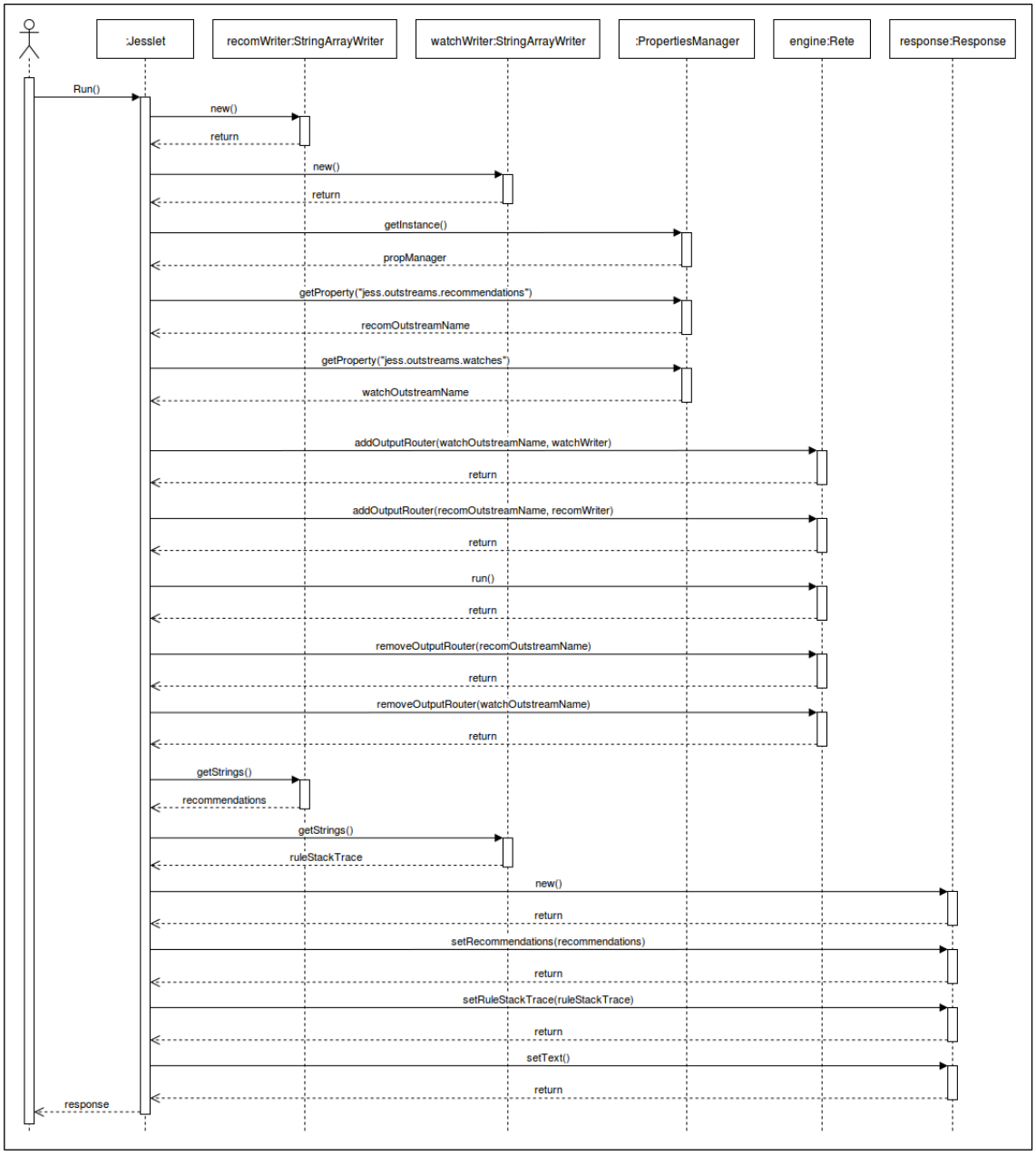


Figura 7.12. Diagrama de Secuencias para la Ejecución de un Sistema.

Cabe aclarar que en este diagrama aparece una clase, aún no mencionada, llamada `PropertiesManager`. Esta clase será explicada más adelante al tratar el requisito del **manejo de un archivo de configuración central**.

Debido al funcionamiento inherente de los servicios web, no es posible que el sistema experto que se ejecuta sobre el Jesslet solicite al usuario información para sumarla a su base de hechos. Una vez que la ejecución es iniciada, esta no puede ser

detenida. Es por ello que, para este prototipo inicial, se ha limitado el conjunto de sistemas que pueden ser representados con el Jesslet a los **sistemas expertos de universo cerrado** (ver capítulo II).

VII.3.7 Séptimo Ciclo

El séptimo y último ciclo abarcó dos características que permitieron mejorar el funcionamiento del Jesslet: el soporte a **múltiples proyectos**, y el manejo de un **archivo de configuración** central. La primera historia de usuario apareció con el objetivo de permitir que el Jesslet pueda manejar más de un sistema experto simultáneamente. Esta característica le agrega otra porción de versatilidad en el contexto de los *servicios web*. Por otro lado, el tener un archivo de configuración central y externo al software, lo vuelve más flexible ante variaciones en su entorno de ejecución, permitiendo ajustar parámetros sin la necesidad de recompilar todo el sistema.

VII.3.7.1 Historia de Usuario: *Soportar Múltiples Proyectos (Sistemas)*

La idea principal detrás de esta historia de usuario es el poder administrar y ejecutar diferentes sistemas dentro del servicio del Jesslet. Todo los métodos y requerimientos descritos hasta este punto permiten agregar y actualizar reglas de inferencias y hechos sobre un único sistema experto contenido en el Jesslet. Si se necesitase crear un nuevo sistema experto, se debería configurar una nueva instancia del servicio (con todo lo que ello implica).

De esta manera, se ideó una forma de permitirles a las aplicaciones clientes especificar sobre qué sistema desean aplicar los métodos que invocan sobre el servicio. El servicio debería **cargar el estado** de ese sistema o proyecto, ejecutar el método invocado, y **guardar el estado** nuevamente para su uso futuro.

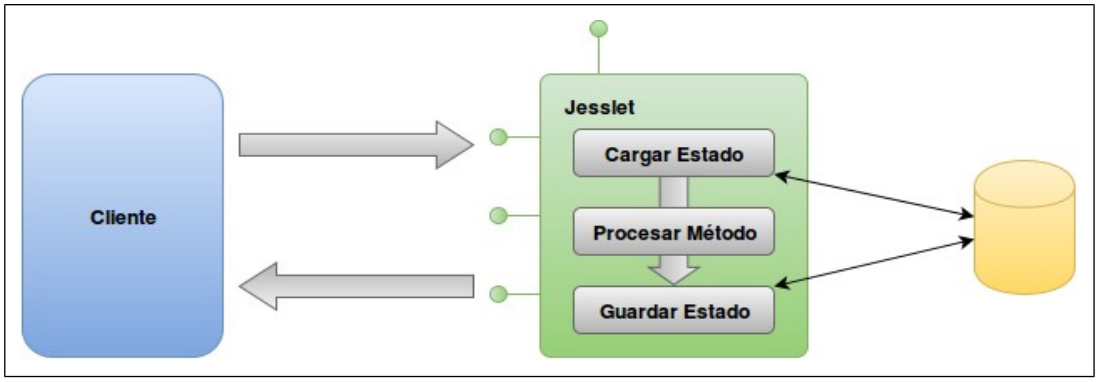


Figura 7.13. Carga y Almacenamiento de Sistemas.

Para poder almacenar el estado de un sistema en Jess, en primera instancia, tuvo que encontrarse una manera de representarlo. Afortunadamente, la clase `Rete` cuenta con dos funciones para guardar y recuperar los contenidos de un sistema: `bsave` y `bload`. Este último recibe un parámetro del tipo `InputStream`, y sirve para cargar el contexto (estado) mediante este stream. Por otro lado, el método `bsave` recibe un parámetro `OutputStream`, y tiene el objetivo de guardar dicho contexto sobre este.

La especificación del proyecto de trabajo por parte del cliente se hace mediante un parámetro **PID** (identificador de proyecto), el cual debe ser enviado en cada mensaje SOAP en el elemento `<Header>` (ver capítulo IV).

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"
  xmlns:ns="http://fce.unse.edu.ar/jesslet">

  <soap:Header>
    <ns:PID>FOO</ns:PID>
  </soap:Header>
  <!-- ... -->
</soap:Envelope>
```

Para la recepción de la cabecera `<PID>` se hace mediante un parámetro especial agregado en la firma de cada uno de los métodos de la interfaz `IJesslet`. Este parámetro se recibirá desde el `<Header>` del mensaje SOAP. En el fragmento que aparece a continuación , puede verse la definición de este parámetro, pero de modo que

representa lo mismo para cada método, ha sido obviado en las definiciones de los métodos antes presentados.

```
@WebMethod
    boolean AddRule(
        @WebParam(mode = WebParam.Mode.IN,
            name = "PID",
            partName = "PID",
            targetNamespace =
JessletConstants.WS_TARGETNAMESPACE,
            header = true) String pid,
        @WebParam(name = "rule") Rule rule) throws
JessletFault;
```

El parámetro en `<PID>` puede ser cualquier cadena de caracteres. Si el proyecto se encuentra en la base de datos de proyectos, entonces es cargado. Sino, es creado usando el nuevo identificador, y luego se lo carga. Para esta funcionalidad, la clase `Jesslet` fue dotada de tres funciones: `getPID`, `load`, y `save`. `getPID` recupera el identificador desde la cabecera del mensaje SOAP. Se debe notar que si la ejecución no se encuentra en un contexto de un servicio web, por ejemplo en las pruebas, el identificador será recuperado como `null`. El método `load` utiliza el identificador para recuperar desde la base de datos los contenidos del proyecto, mientras que `save` los almacena. Estos dos últimos métodos utilizan un arreglo de *bytes* para representar estos contenidos, y son los encargados de interactuar con los métodos `bload` y `bsave` de `Rete`. Para enviar y recibir los datos desde esta última clase, utilizan las clases nativas de Java `ByteArrayInputStream` y `ByteArrayOutputStream`.

La **base de datos de proyectos** se abstrae mediante la *clase abstracta* `Storage`, la cual establece *tres métodos* necesarios para tales fines: `LoadProject`, `SaveProject`, y `ProjectExists`. El primero se encarga de recuperar un proyecto, y recibe como parámetro el identificador del proyecto (`pid`). Los contenidos del proyecto se devuelven como un arreglo de bytes (*stream*), formato que utiliza la clase `Rete` en Jess para representar el contexto (estado) de un sistema.

La Figura que aparece a continuación resume en un diagrama de clases las relaciones entre las clases, sus métodos y atributos.

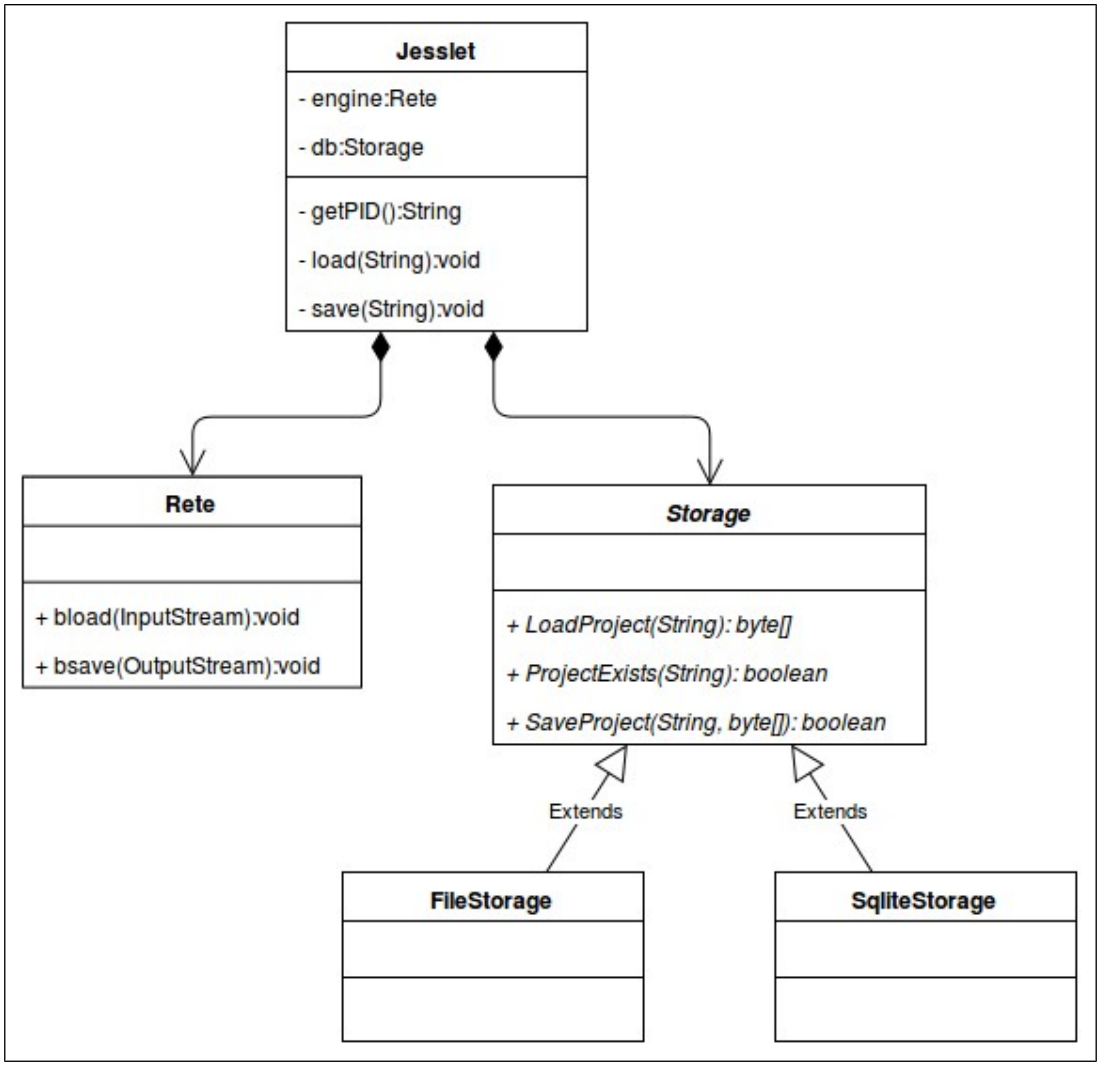


Figura 7.14. Diagrama de Clases del Almacenamiento de Proyectos.

Como puede verse, existen dos clases que heredan de `Storage`: `FileStorage` y `SqliteStorage`.

La primera implementación utiliza el sistema de archivos de manera directa para la persistencia de los datos de los proyectos. Cada proyecto es almacenado en un archivo con el mismo nombre que su identificador, en el directorio donde se ejecuta la aplicación.

La segunda clase, `SqliteStorage`, como su nombre lo indica, utiliza la biblioteca **SQLite** [SQLITE] para gestionar el almacenamiento y recupero de los proyectos. Dentro del proyecto, se tiene un archivo denominado `jess.db`, el cual representa el esqueleto inicial de la base de datos. Cuando el servicio intenta guardar o

recuperar datos mediante `SqliteStorage` por primera vez, el sistema crea un archivo homónimo en el directorio de ejecución para realizar las consultas.

VII.3.7.2 Historia de Usuario: *Archivo de Configuración Centralizado*

Un archivo de configuración central tiene el propósito de parametrizar el sistema y hacerlo más flexible ante ciertos cambios y variaciones del entorno en el cual se ejecuta. Un ejemplo clásico de esta situación son los parámetros de conexión a una base de datos. Estos datos, muchas veces, necesitan ser actualizados, por lo que es necesario que esto se haga de manera única y sin necesidad de recompilar parte del software.

Un archivo de configuración (o *archivo de propiedades*) `config.properties` ha sido creado dentro del Jesslet, el cual contiene los datos de configuración necesarios. Para su acceso, se programó una clase especial llamada `PropertiesManager`, la cual tiene el solo objetivo de acceder al archivo de propiedades y devolver el valor de una de ellas. Para ello, cuenta con un único método llamado `getProperty`, el cual recibe el nombre de la propiedad que se quiere recuperar, y devuelve una cadena con su valor

```
public String getProperty(String propertyName);
```

Este método se encuentra *sobrecargado* con un segundo argumento que representa un valor por defecto, en el caso de que la propiedad no haya sido encontrada.

```
public String getProperty(String propertyName, String
defaultValue);
```

Debido a que en toda la ejecución del sistema sólo es necesario contar con una única instancia de `PropertiesManager`, esta clase se ha creado siguiendo el patrón de diseño **Singleton**. Posee un **constructor privado**, y un **método estático** `getInstance` para recuperar esta instancia única.

```
private PropertiesManager ();

public static PropertiesManager getInstance();
```

Durante la inicialización del servicio, el Jesslet buscará el archivo `config.properties` en el mismo directorio de ejecución. De no encontrarlo, procederá a utilizar el archivo definido (y compilado) dentro de sí mismo. El código que aparece a continuación es el archivo de configuración original que se encuentra compilado en el código del servicio.

```
# Nombre del stream de salida de las recomendaciones
jess.outstreams.recommendations=wsout

# Nombre del stream del watch de las reglas
jess.outstreams.watches=watchout

log.format=plain

# Nombre de la clase Storage a ser utilzada.
jesslet.storage.class=ar.edu.unse.fce.jesslet.service.FileStorage
jesslet.storage.config=/home/ricardo/srv/jesslet/projects

jesslet.service.location=http://127.0.0.1:8888/jesslet
```

VII.4 Funciones para Fecha/Hora

Para poder implementar el **Sistema de Sugerencia de Vacunas** (ver capítulo VIII) utilizando el *Jesslet* fue necesario contar con las funciones de cálculo de fechas programadas en el archivo `fechahora.clp`. Para ello se desarrolló un conjunto de funciones análogas en un archivo `datetime.clp`, el cual puede ser encontrado en el el directorio `/src/ar/edu/unse/fce/jesslet/jess/`. Cuando se debe crear un nuevo proyecto, este archivo es cargado inmediatamente usando el método `batch` de la instancia de la clase `Rete`. Así, todo proyecto que sea administrado a través de este servicio web tendrá a su disponibilidad las funciones `datetime-*`.

Estas funciones trabajan con valores del tipo **fecha/hora**, que no son otra cosa que **cadenas de caracteres** en el **formato de fecha y hora local del estándar ISO** [ISO8601] (`yyyy-mm-ddThh:mm:ss`), por ejemplo `1985-07-04T14:00:00`. La lista que aparece a continuación, presenta cada una de las funciones de fecha/hora disponibles.

- `datetime-now`: Devuelve la fecha y hora actual. Esta función no recibe parámetros.
- `datetime-format`: Aplica un formato sobre un valor fecha y hora, y lo devuelve como resultado. Recibe como parámetros la **fecha** a la que se aplica el formato, y el **formato** a aplicar. Para la definición de este formato se usan los patrones definidos para la clase `SimpleDateFormat` [clase SimpleDateFormat] de Java.
- `datetime-timestamp`: Recupera el **tiempo UNIX** de una fecha y hora, y tiene como parámetro único este valor.
- `datetime-fromformat`: Realiza la acción inversa de `datetime-format`. Intenta crear una fecha y hora a partir de un valor de texto con cierto formato. Esta función recibe dos parámetros: el texto a ser convertido, y el formato desde el cual se trabajará. Nuevamente, este formato debe respetar los patrones utilizados para la clase `SimpleDateFormat`.
- `datetime-diff`: Calcula la *diferencia* entre dos valores de fecha y hora en una *unidad* dada. Este procedimiento recibe tres parámetros: la fecha y hora inicial, la fecha y hora final, y la unidad sobre la cual será calculada. Esta unidad puede tomar los valores `s` para *segundos*, `m` para *minutos*, `h` para horas, `d` para días, `w` para semanas, `M` para meses, y `y` para años.
- `datetime-add`: Adiciona una cantidad de unidades de tiempo a una fecha dada. Los parámetros que esta función espera son la fecha a ser modificada, la cantidad de unidades de tiempo a ser agregada, y la unidad de tiempo. Esta última unidad puede tomar alguno de los valores para las unidades de tiempo presentada en `datetime-diff`.

El fragmento que aparece a continuación muestra un ejemplo del funcionamiento de algunos de los procedimientos `datetime-*`.

```
Jess> (bind ?hoy (datetime-now))
"2015-12-03T10:54:03"
Jess> (bind ?fecha-nac 1985-07-04T14:00:00)
```

```
1985-07-04T14:00:00
Jess> (printout t "Hoy es " (datetime-format ?hoy "EEEE d 'de'
MMMM 'del' y") crlf)
Hoy es jueves 3 de diciembre del 2015
Jess> (printout t "Tiene " (datetime-diff ?fecha-nac ?hoy w) "
semanas de vida." crlf)
Tiene 1586 semanas de vida.
```

VII.5 Pruebas desde Otros Entornos

Una de las ventajas de los servicios web radica en el hecho de que pueden ser accedidos desde un sinnúmero de aplicaciones y plataformas. A continuación se describe el desarrollo de uno de los aplicativos que utilizan el *Jesslet* para sus operaciones en este trabajo. La finalidad de este desarrollo es el de demostrar la implementación y el funcionamiento de un sistema experto en *Jess* una plataforma distinta a Java. En este caso se trata de una **aplicación de escritorio** que permite administrar y probar uno o más proyectos de sistemas expertos.

Asimismo, el segundo, es una **extensión para el navegador Google Chrome**, y tiene por función realizar consultas sobre la implementación del *sistema de sugerencia de vacunas SEVAC* en el *Jesslet* (ver capítulo VIII).

VII.5.1 Aplicación de Administración: PyJesslet

PyJesslet es un programa de escritorio escrito en **Python**, el cual tiene como propósito administrar los hechos, las reglas de inferencias, ejecutar y recuperar los resultados de un sistema experto a través del *Jesslet*.

La Figura 7.15 muestra la arquitectura general de *PyJesslet*, en la cual se observa como la aplicación utiliza el servicio web programado. Más en detalle, ilustra cómo trabaja cada componente y cómo es la relación entre ellos. En particular, *PyJesslet* utiliza un módulo especial de Python para manejar las particularidades de la mensajería *SOAP*; mientras que en *Jesslet*, los mensajes XML que se reciben son gestionados por las clases del paquete *JAX-WS* de Java. Estas últimas determinan qué método será invocado en la clase principal, y, consecuentemente, en el motor de Jess (`jess.Rete`) en sí; tal como se especificó anteriormente.

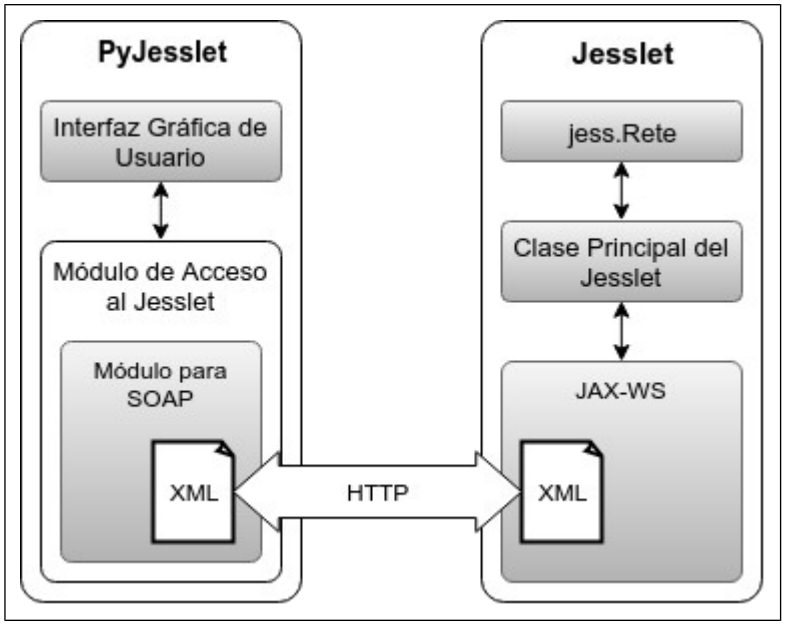


Figura 7.15. Conexión entre PyJesslet y el Jesslet

La Figura 7.16 muestra la ventana principal del *PyJesslet*, junto con algunas de las opciones del menú principal, las cuales se detallarán a continuación:

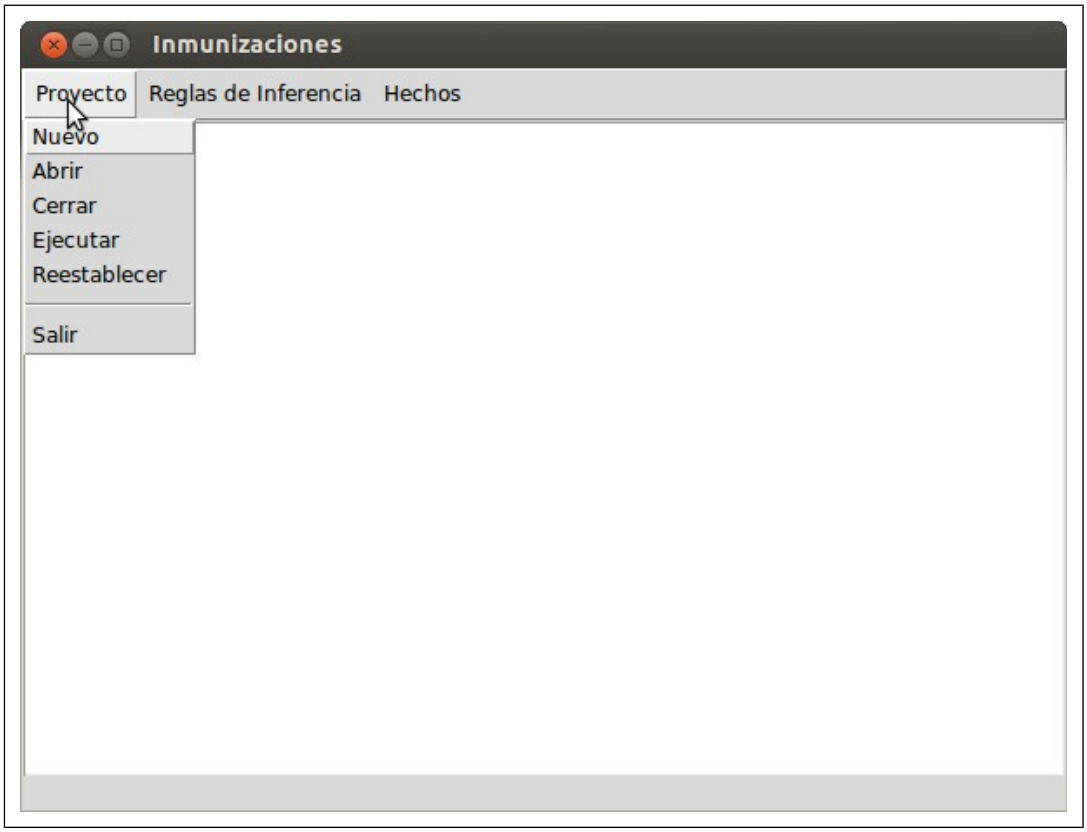


Figura 7.16. Ventana Principal de PyJesslet.

Esta aplicación, cuyo código fuente se encuentra versionado mediante *Mercurial* y usando el servicio *BitBucket* como repositorio central en https://bitbucket.org/rgmiranda/pyjesslet, además fue utilizada para la implementación del sistema experto **SEVAC** (ver capítulo VIII).

VII.5.1.1 Proyectos

Desde la opción *Proyecto* pueden administrarse los proyectos asociados. Un proyecto es, en esencia, un sistema experto en Jess. El Jesslet requiere para la invocación de los métodos, un parámetro `PID` en la cabecera el mensaje SOAP. El valor de este puede ser cualquier cadena de uno o más caracteres, y es el identificador del proyecto sobre el cual el Jesslet ha de realizar el trabajo. La Figura 7.17 es una captura de pantalla de la ventana de apertura de proyectos. Aquí puede verse el nombre del proyecto junto con su *PID* asociado.

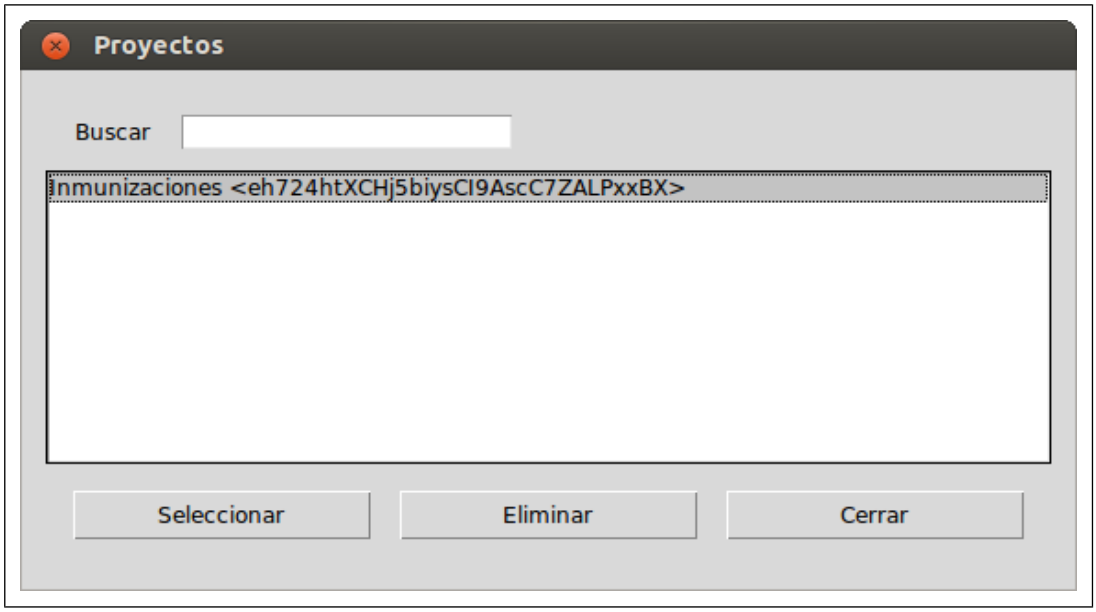


Figura 7.17. Ventana de Apertura de Proyectos.

VII.5.1.2 Administración de las Reglas de Inferencia

Las reglas de inferencia pueden ser administradas accediendo a la opción *Ver Todas* en el submenú *Reglas de Inferencia*. *Nueva* abre directamente el formulario para agregar una nueva regla en el sistema.

La Figura 7.18 muestra el formulario de administración de reglas, en el cual se listan reglas de inferencias relacionadas con la aplicación de la vacuna *BCG*, durante la implementación del **SEVAC** usando la herramienta *PyJesslet*. Desde esta ventana puede crearse una nueva regla, actualizar una existente, o eliminarla con los botones en la parte inferior.

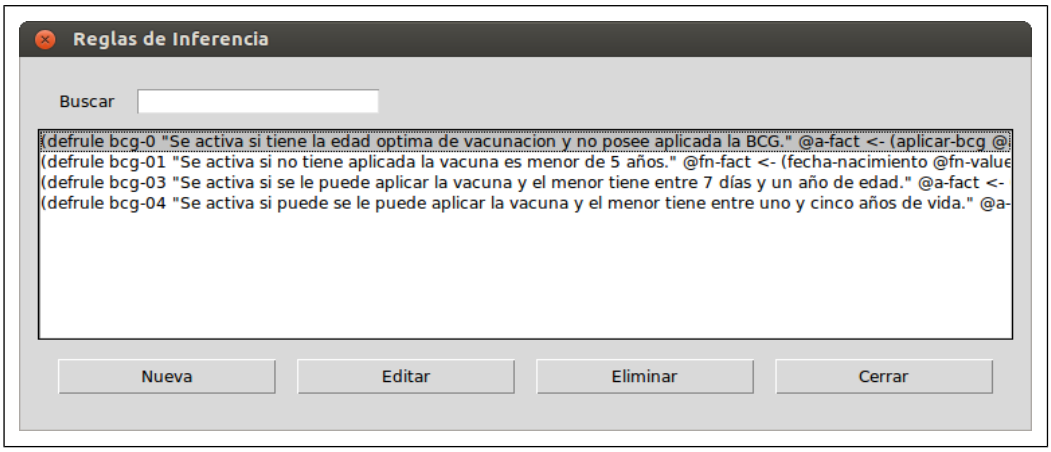


Figura 7.18. Ventana de administración de reglas de inferencias.

El formulario para una nueva regla de inferencia permite cargar el *nombre* de la regla, una *descripción* breve que será utilizada como documentación de la misma, las *condiciones* (antecedente) de activación, y las *acciones* a ser llevadas a cabo al ser disparada. Hay que notar que el nombre de una regla no puede ser alterado una vez que esta ha sido creada. Esta situación puede verse en el formulario presentado en la Figura 7.19: el formulario posee deshabilitado el campo de texto correspondiente.

Los antecedentes aparecen en el listado dentro del primer panel (titulado *Condiciones*) junto con las opciones para su gestión, y los consecuentes se presentan en el segundo (titulado *Acciones*).

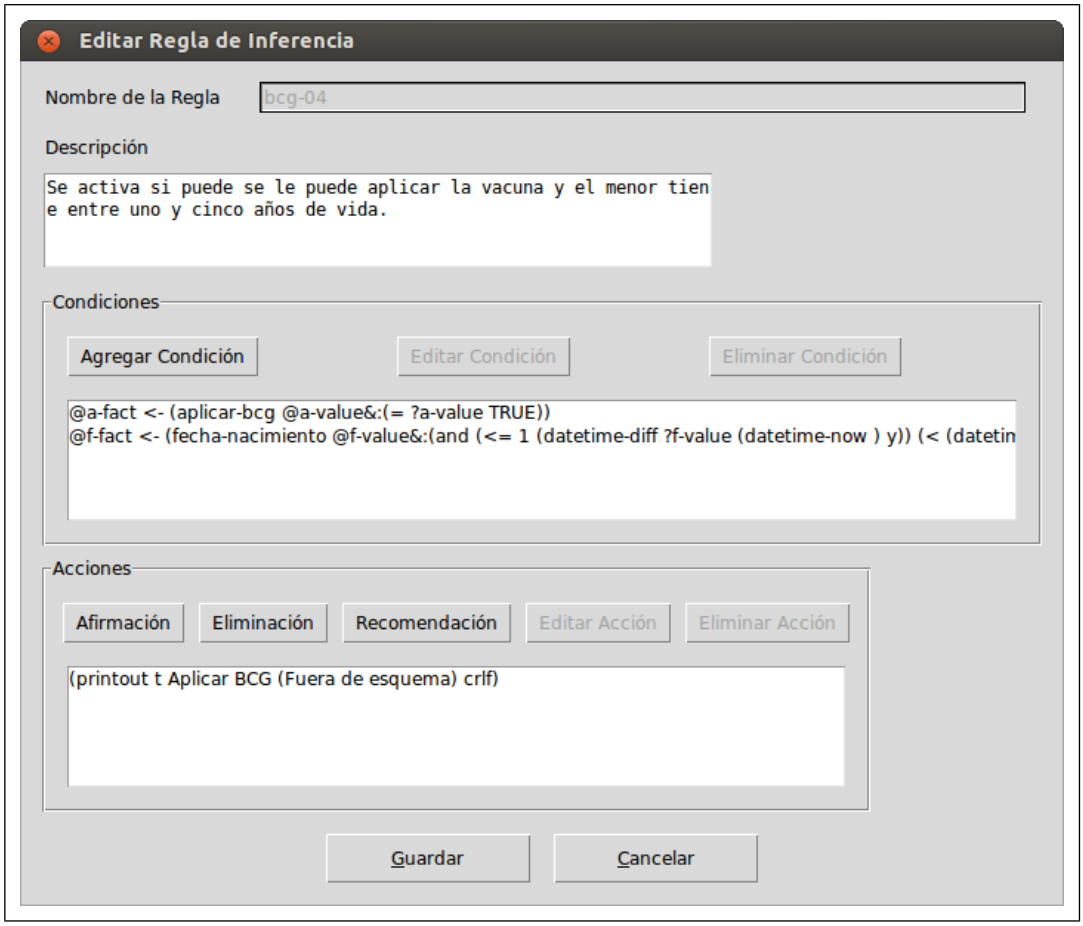


Figura 7.19. Formulario de Edición de una Regla.

Un detalle que podría prestar a confusión y conviene aclarar es la notación con la cual se expresan las reglas de inferencia, sus condiciones y acciones en los formularios. Las reglas de inferencias en la ventana de la Figura 7.18, y las condiciones y acciones de la Figura 7.19 se expresan usando el **lenguaje Jess** (ver capítulo III). Esto no necesariamente debe ser así. Internamente, las reglas, acciones, y condiciones son **objetos** con sus atributos y métodos, y estos pueden ser presentados de innumerables formas. Se optó por expresarlos de esta manera a los fines de poder llevar un mejor control sobre los datos que se están comunicando desde y hacia el Jesslet. Recordemos que se trata de un prototipo, y muchas de sus funcionalidades pueden ser propensas a fallos o provocar situaciones inesperadas.

VII.5.1.3 Antecedentes de la Reglas

Los antecedentes de una regla de inferencia son todos los hechos (junto con sus restricciones) que determinan su activación. Cuando se agrega o modifica una condición, se presenta el formulario que aparece en la Figura 7.20.

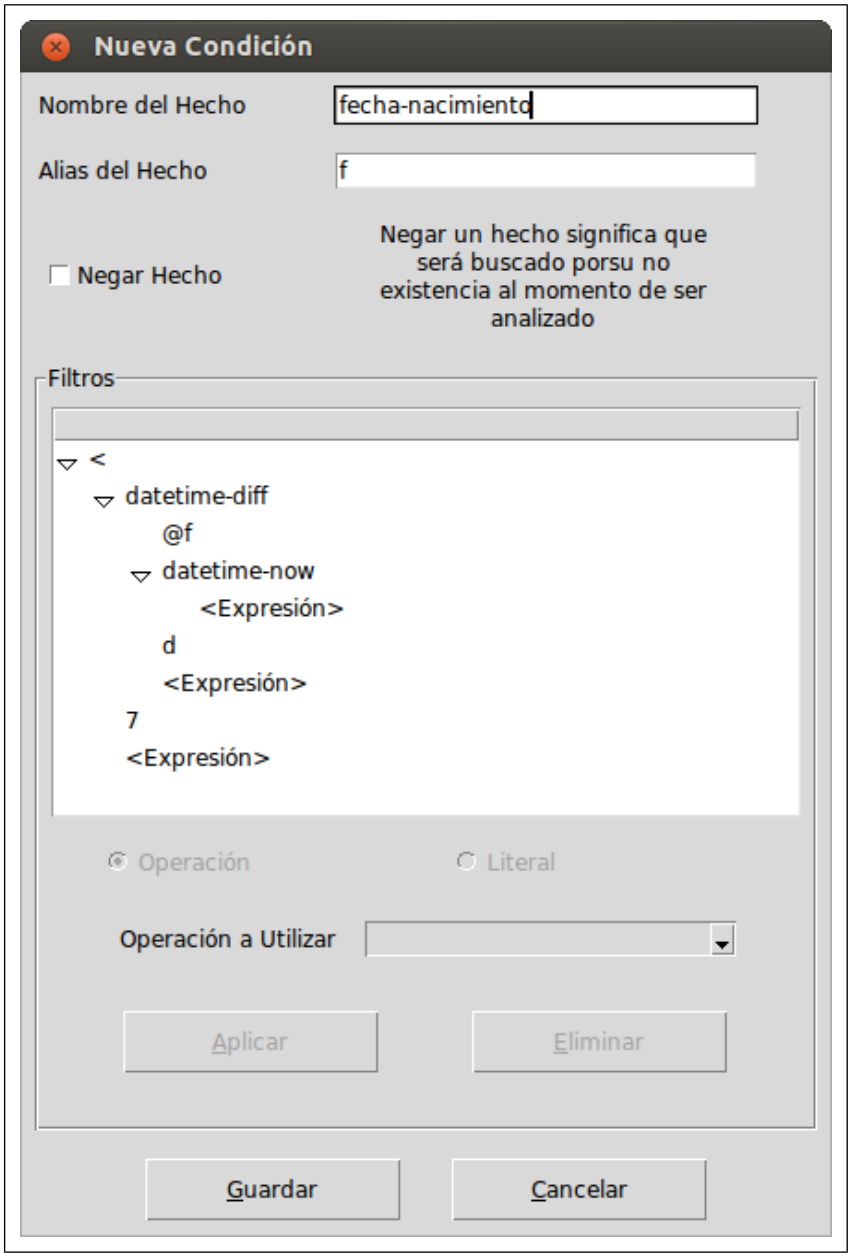


Figura 7.20. Formulario de Edición de una Condición.

El **nombre del hecho** es el nombre *cabecera* que posee el hecho en la memoria de trabajo. El **alias** es el *identificador de la variable* usada para almacenar el hecho y su valor. Si la condición es marcada con **negar**, se le indicará al motor de inferencias que

este hecho *no debe existir* para cumplir con la condición. El panel de **filtros** permite establecer restricciones con respecto al valor de este u otros hechos que aparezcan entre los antecedentes de la regla.

En el formulario puede verse como el filtro se define y expresa como un *árbol*. Esto se realizó de esta forma debido a que existe una traducción casi directa entre un árbol de operaciones y una expresión en lenguaje *Jess*, y visualizar las expresiones de esta manera simplificaría la depuración de errores lógicos y de programación. La expresión del ejemplo se convierte en la siguiente:

```
(< (datetime-diff ?f-value (datetime-now) d) 7)
```

Esta expresión (*booleana*) será utilizada como **restricción de función predicado** dentro de la condición. En este caso, devuelve `TRUE` si la diferencia en días entre valor `?f-value` y la fecha y hora actual es menor a 7, es decir, si la persona tiene menos de 7 días de vida.

Desde los controles que aparecen en la parte de abajo del panel de filtros puede definirse si estos son una *expresión literal* (`"Una cadena"`, `256`, `TRUE`), o una *expresión compuesta* `((= 8 (* 2 2 2)))`. Si se trata de esta última, entonces se le permite agregar otras expresiones anidadas para dar lugar a los parámetros.

VII.5.1.4 Consecuentes de la Reglas

Los consecuentes de las reglas son todas las acciones que se llevan a cabo al ser disparada la regla, y, como ya fue mencionado, para evitar una alta complejidad, el conjunto de acciones disponibles en el Jesslet fue limitado a tres: creación de hechos (`assert`), eliminación de hechos (`remove`), y de presentación de mensajes para el operador del sistema (`recommend`). En el panel de *Acciones* puede verse una serie de botones que se corresponden con estos tipos de acciones, y abren el formulario correspondiente.

En la Figura 7.21 se presenta el formulario que permite establecer una acción para agregar un hecho a la memoria de trabajo. Debido a que el valor que puede contener un

hecho puede ser una expresión, aparece un control de vista de árbol similar al presentado para la definición de los filtros de un hecho en el antecedente.

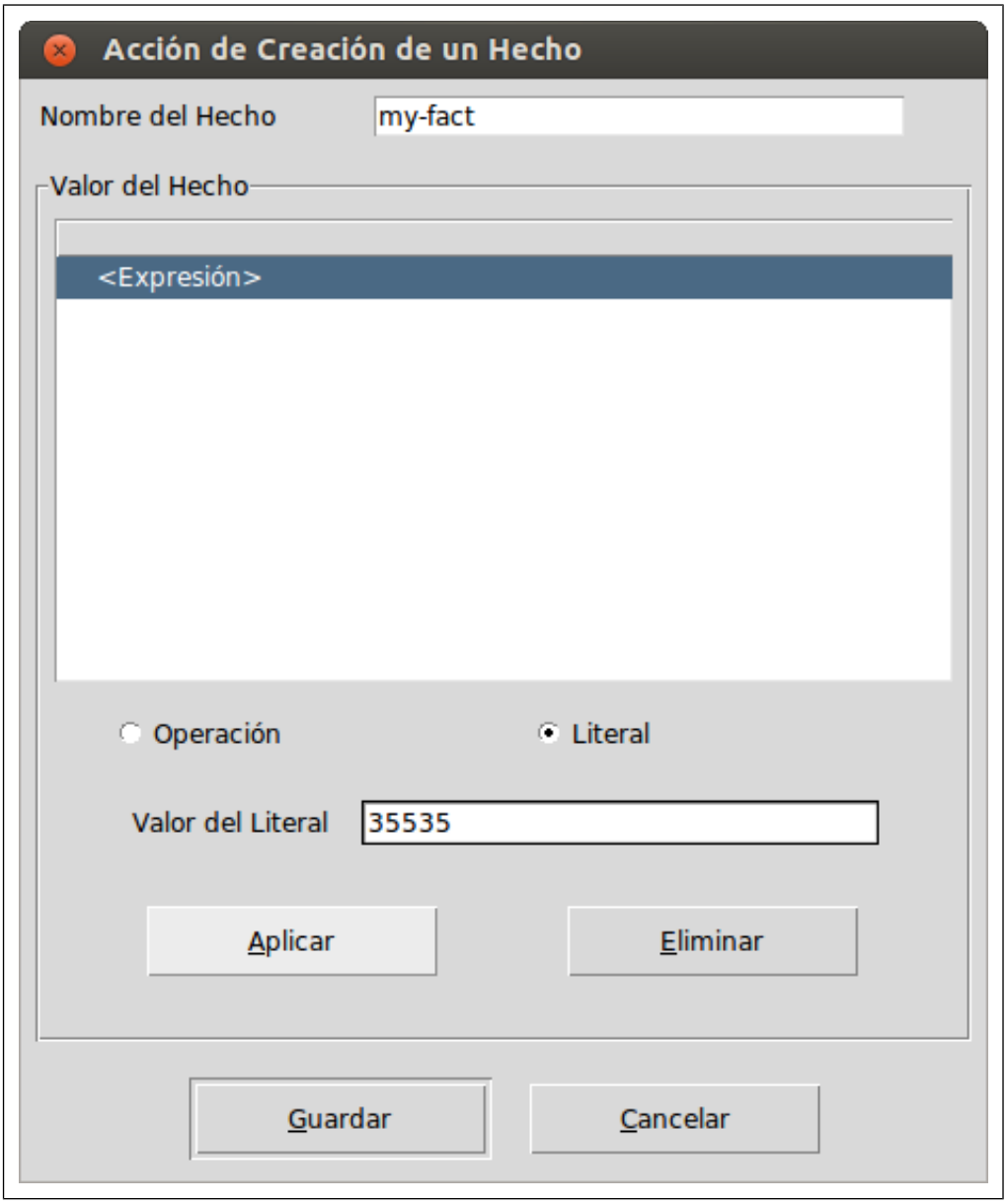


Figura 7.21 . Formulario de Acción de Creación de Hecho.

La eliminación de un hecho sólo demanda el ingreso del **identificador** de este. Este identificador debe ser el **valor de un alias** de un hecho definido en alguna de las condiciones del antecedente, **antepuesto con el carácter** `@`. La ventana en la Figura 7.22 contiene el formulario para esta acción.

Figura 7.22 Formulario de Acción de Eliminación de un Hecho.

Para disponer una acción de recomendación para el operador también se debe establecer como único parámetro el mensaje a ser presentado (Figura 7.23).

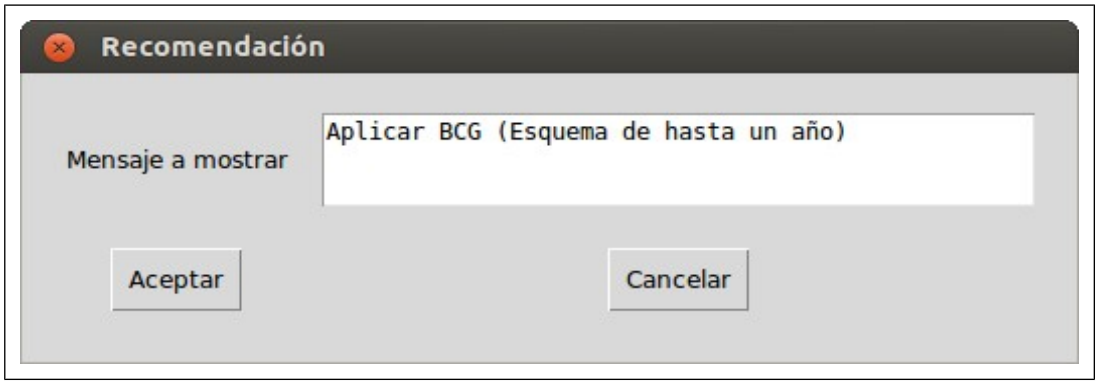


Figura 7.23. Formulario de Acción de Recomendación.

VII.5.1.5 Hechos

Los hechos de la memoria de trabajo del proyecto se administran desde la opción *Hechos* del menú principal. Si se accede a *Ver Hechos*, se abre una ventana con el listado de hechos que actualmente posee el proyecto. Cabe aclarar que el hecho inicial (`initial-fact`), creado cuando se invoca al método `reset` de Jess, no aparece en este listado. La Figura 7.24, que muestra esta ventana, contiene un único hecho (además del inicial) `fecha-nacimiento`.

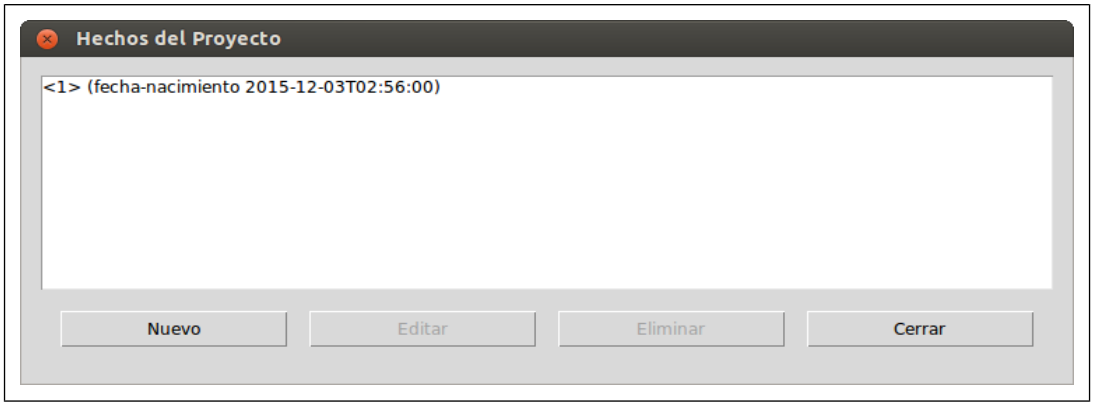


Figura 7.24. Administración de Hechos.

También puede accederse directamente al formulario de creación de un nuevo hecho desde *Nuevo Hecho* en el menú principal, o bien desde el botón *Nuevo* en la ventana del listado. Cualesquiera de estas opciones, abre el formulario para definir un hecho en la memoria de trabajo del proyecto. La Figura 7.25 muestra este formulario en donde se solicita el nombre del hecho, una descripción breve, y su valor.

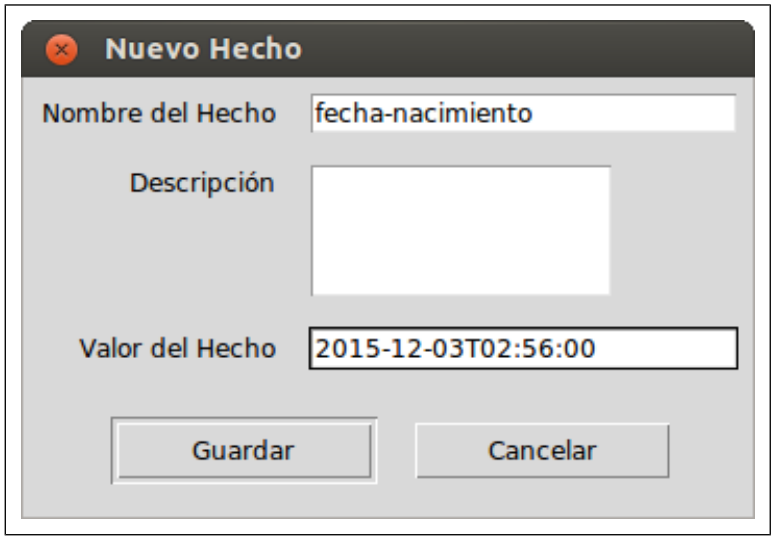


Figura 7.25. Formulario de Edición de un Hecho.

VII.5.1.6 Restablecimiento y Ejecución

Para restablecer un proyecto se utiliza la opción *Restablecer* de la parte de *Proyecto* en el menú principal, con lo cual se invoca al método `Reset` en el Jesslet. Si la operación ha sido exitosa, en la caja de presentación de mensajes en el formulario principal aparece algo como lo siguiente:

```
Restableciendo el proyecto <pid>..

El proyecto a sido restablecido.
```

Vale recordar que este método elimina todos los hechos de la memoria de trabajo y crear el hecho inicial cero, por lo que su invocación debe hacerse antes de cargar lo hechos para una ejecución.

La ejecución de un proyecto se hace mediante la opción *Ejecutar*, con lo que se invoca al método `Run` en el Jesslet. Durante la ejecución pueden activarse y desactivarse reglas, crearse y borrarse hechos, y generarse mensajes para el operador. Estos mensajes (recomendaciones), de existir, aparecerán en la caja de mensajes. En el ejemplo que aparece a continuación, después de la ejecución se recomienda aplicar la vacuna *BCG* al beneficiario.

```
Ejecutando el proyecto <pid>...

Se ha terminado con la ejecución del programa.
Aplicar BCG (Esquema óptimo)
```

Si la ejecución no ha devuelto recomendación alguna, entonces el mensaje de resultado se verá como el siguiente:

```
Ejecutando el proyecto <pid>...

Se ha terminado con la ejecución del programa.
No se ha producido recomendación alguna.
```

Por último, resulta importante destacar que para este prototipo inicial, aún no se ha trabajado sobre la presentación de la traza de activación y ejecución de reglas de inferencia, y la generación y eliminación de hechos.

VII.6 Resumen

En esta parte se ha expuesto el proceso seguido para la creación del prototipo inicial del *Jesslet*. En primer lugar, se han establecido herramientas básicas para el desarrollo del software, como los estándares de codificación y el sistema de control de versiones a utilizar. Posteriormente, se presentaron las historias de usuario definidas

para poder llegar al producto final, y cómo fueron programadas a lo largo de una serie de ciclos de desarrollo, tal como lo propone la XP. En cada ciclo se ha mostrado cómo ha sido creada cada historia, junto con los detalles y cuestiones más sobresalientes que emergieron en el proceso. Finalmente, se ha presentado el programa *PyJesslet*, el cual tiene la finalidad de administrar proyectos en el *Jesslet*, sus reglas de inferencia y hechos, y ejecutarlos para recuperar sus recomendaciones. Este último fue desarrollado con el fin de poner de manifiesto la capacidad y versatilidad del *Jesslet* al momento de trabajar con el *shell Jess* desde diferentes lenguajes.

Capítulo VIII *SEVAC* Sistema de Sugerencia de Vacunas en Jess

El shell Jess es un potente motor para la creación de sistemas expertos basados en reglas de inferencia. A los fines de demostrar sus capacidades, en este capítulo se desarrolló el **sistema experto SEVAC** para realizar sugerencias sobre la aplicación de vacunas a una persona, basándose en su edad e historial de inmunizaciones aplicadas.

Estas sugerencias estarán basadas en el **calendario nacional de vacunación** y en la información de sobre inmunizaciones provista por la Dra. Florencia Coronel, asesora del presente trabajo.

VIII.1 Dominio de Aplicación

En el contexto de este sistema experto de sugerencias, existen ciertos conceptos que deben ser desarrollados a los fines de evitar malentendidos y definiciones ambiguas.

En primer lugar, en relación al sistema, una **inmunización** se entiende como una combinación de una vacuna, y su número de dosis. Por ejemplo:

- *Hepatitis B 2da Dosis*
- *SABIN Refuerzo*
- *Quíntuple 3ra Dosis*

Por otro lado, un **esquema** es una clasificación de las inmunizaciones aplicadas que se realiza en base a la *edad de la persona* y los *factores de riesgo* que afectaban a la persona al momento ser inmunizado. Los esquemas que pueden encontrarse son:

- **Esquema Normal:** La aplicación de la dosis se ha realizado de acuerdo al calendario y en situaciones ideales contempladas para la vacuna.
- **Esquema Alternativo:** La aplicación de la dosis se ha realizado de acuerdo a parámetros que no podrían ser considerados normales, pero que igualmente resultan aceptables.

- **Fuera de Esquema:** La aplicación se ha realizado en parámetros fuera de lo normal.

El esquema en el cual habrá de clasificarse cada dosis dependerá exclusivamente de la vacuna correspondiente.

VIII.2 Fuentes de Conocimiento

El conocimiento para el *SEVAC* se obtuvo desde dos fuentes:

- Información pública respecto al *Calendario Nacional de Vacunación* brindada por el *Ministerio de Salud de la Nación* http://www.msal.gob.ar/index.php/programas-y-planes/184-calendario-nacional-de-vacunacion-2016.
- Entrevistas realizadas a la Dra. Coronel.

Cabe aclarar que lo que finalmente resultó ser la educción de conocimientos para SEVAC originalmente no se hizo para desarrollar un sistema experto, sino en el marco del desarrollo de otro sistema solicitado por la Dra. Coronel. El propósito de esta aplicación es el *registro de aplicaciones de vacunas* del Calendario Nacional en los centros de salud habilitados.

VIII.3 Proceso de Desarrollo

El desarrollo del ***SEVAC*** se realizó siguiendo la metodología **IDEAL** propuesta por [GARCIA2004]. Debido al contexto en el cual fue desarrollado este software, muchas etapas de la metodología fueron obviadas. Puntualmente, se realizó solamente la fase II, la cual abarca la creación de los prototipos del sistema experto, haciendo hincapié en las etapas de **conceptualización** y de **formalización**.

En este punto cabe notar que el sistema no tiene casi interacción con el operador. Existen dos motivos para esta decisión:

1. Dada la gran cantidad de vacunas y dosis involucradas, evidentemente el preguntar al operador sobre las inmunizaciones recibidas por una persona puede resultar molesto y poco usable.

2. Debido a que se busca luego convertir este mismo sistema (o una versión análoga) en un componente a modo de servicio web, la interacción con el usuario debe ser reducida. Una de las ideas fundamentales, en este sentido, es que su principal interacción ha de ser con otros sistemas y no tanto con personas.

A futuro, podría pensarse en mejorar esta primera aproximación, y darle la posibilidad al sistema de reportar un calendario completo con rangos de fechas dentro de los cuales deberían aplicarse las inmunizaciones a la persona.

VIII.3.1 Conceptualización

Durante esta etapa se crearon los modelos para representar los **conocimientos fácticos** y los **conocimientos tácticos**.

Para los primeros se trabajó el **diccionario de conceptos** y las **tablas de concepto-atributo-valor**. El diccionario de conceptos permite identificar la terminología clave y de más alto nivel. Para cada concepto se establece su utilidad o función, sinónimos, y los atributos que lo definen. Una tabla de concepto-atributo-valor se utiliza para ordenar los conceptos identificados, y en esta se definen los atributos de los conceptos en un grado más de detalle, incluyendo los valores que pueden tomar los mismos.

Los conocimientos tácticos fueron abordados mediante las **tablas de decisión** y los **grafos de causalidad**. Las primeras sirven para describir procesos y procedimientos de información. En estas tablas se presentan las condiciones (conjunto de circunstancias) que dan lugar a un conjunto de acciones. Una tabla típica se divide en cuatro cuadrantes. El primer cuadrante lista las condiciones, y en el segundo aparecen las acciones. El tercer cuadrante contiene las combinaciones de valores de las condiciones que darán lugar a la toma de decisión, y el cuarto indica qué acciones se deben realizar en cada caso. Por otro lado, los grafos de causalidad aparecen como una aproximación a las **reglas de producción** con las que se llenará la base de conocimientos del *SEVAC*. Un grafo causal es una representación automáticamente manipulable del conocimiento asociado a los procesos deductivos del experto de campo, la cual permite comparar el procedimiento que realiza el experto de campo con el que

realizará el sistema. Se construyen en base a las tablas de decisión, y a las entrevistas y observaciones realizadas con la persona experta.

El diccionario de conceptos, que aparece en la Tabla 8.1, lista y define los conceptos y términos que se manejan dentro del dominio del sistema.

Concepto	Función	Sinónimos	Atributos
Beneficiario	Persona que recibió una aplicación de una vacuna.	Persona	Posee nombre, apellido, un número de documento (DNI), y fecha de nacimiento.
Vacuna	Una de las vacunas especificadas en el Calendario Nacional.		Se compone por el nombre y un código alfanumérico identificatorio.
Dosis	Una de las dosis asociadas a una vacuna del Calendario.		Una dosis asociada a una vacuna que contiene un nombre, y un código numérico.
Aplicación	Una aplicación de una dosis de una vacuna a una persona.	Inmunización	Se compone esencialmente de una dosis de una vacuna, una persona, y una fecha de aplicación.

Tabla 8.1. Diccionario de Conceptos

La tabla concepto-atributo-valor (Tabla 8.2) ahonda en detalles para cada concepto, presentando sus atributos, y los valores que éstos pueden tomar.

Concepto	Atributos	Valor	Observaciones
Beneficiario	Nombre	Cadena de caracteres	
	Apellido	Cadena de caracteres	
	Número de DNI	Número entero positivo	
	Fecha de Nacimiento	Fecha	
Vacuna	Nombre	Cadena de caracteres	
	Código	Cadena de caracteres	Código único interno de una vacuna
Dosis	Nombre	Cadena de caracteres	
	Orden	Número entero positivo	Indica el orden de la dosis dentro del esquema de la vacuna
	Código	Número entero positivo	Código de la dosis para la vacuna
	Vacuna	Cadena de caracteres	Código único de la vacuna asociada a la dosis
Aplicación	Fecha de Aplicación	Fecha	
	Vacuna	Cadena de caracteres	Código único de la vacuna aplicada
	Dosis	Número entero positivo	Código de la dosis aplicada
	Número de DNI	Número entero positivo	

Tabla 8.2. Tabla Concepto-Atributo-Valor

Como se mencionó previamente, el objetivo del *SEVAC* es presentar alertas de aplicación de vacunas basándose en la edad de la persona y su historial de inmunizaciones. Debido al gran número de vacunas que aparecen en el calendario, y

todas los rangos etarios a ser analizados, tanto las tablas de decisión como los grafos de causalidad del *SEVAC* se presentan discriminadas por cada vacuna tratada.

VIII.4 Las Vacunas

Las vacunas a ser incluidas en el desarrollo de este sistema de sugerencia son las siguientes:

1 **BCG (Tuberculosis)**

2 **Hepatitis B**

3 **Neumococo Conjugada (Meningitis y Neumonía)**

4 **Quíntuple (Difteria, Tos Convulsa, Tétanos, Hepatitis B, Influenza B)**

5 **Cuádruple (Difteria, Tos Convulsa, Tétanos, Influenza B)**

6 **SABIN (Poliomielitis)**

7 **Triple Viral (Sarampión, Rubeola, Paperas)**

8 **Triple Bacteriana Celular (Difteria, Tos Convulsa, Tétanos)**

9 **Triple Bacteriana Acelular (Difteria, Tos Convulsa, Tétanos)**

10 **Hepatitis A**

11 **HPV**

Como puede apreciarse, no todas las vacunas del calendario han sido consideradas. La razón de esto es que no todos los casos requieren decisiones que pueden ser consideradas lo suficientemente complejas. Los casos más sencillos han sido dejados de lado de momento.

VIII.4.1 BCG

La vacuna BCG es una vacuna contra la Meningitis y la forma miliar de la Tuberculosis, y consta de una única dosis. Dependiendo de la edad de la persona, la aplicación de esta dosis única puede ubicarse en los siguientes esquemas:

- **Antes de los 7 días** de vida del niño se reporta en el **esquema normal**.

- **Entre los 7 días y el año** de edad se reporta en el **esquema alternativo**.
- **Entre uno y cinco años** se reporta como **fuera de esquema**.
- A niños **mayores de cinco años**, no se les puede aplicar esta vacuna.

La Tabla 8.3 contiene la tabla de decisiones para esta vacuna.

BCG				
Edad del Beneficiario	menor que 7 días	entre 7 días y un año	entre 1 y 5 años	5 años o más
Aplicada Dosis BCG	No	No	No	No
Acciones				
Aplicar BCG	Sí	Sí	Sí	No
Reportar Esquema	Normal	Alternativo	Fuera de Esquema	NO APLICA

Tabla 8.3. Tabla de decisión de la vacuna BCG

A continuación se presenta el grafo de causalidad para esta vacuna, junto con la tabla de símbolos (Tabla 8.4) y sus significados.

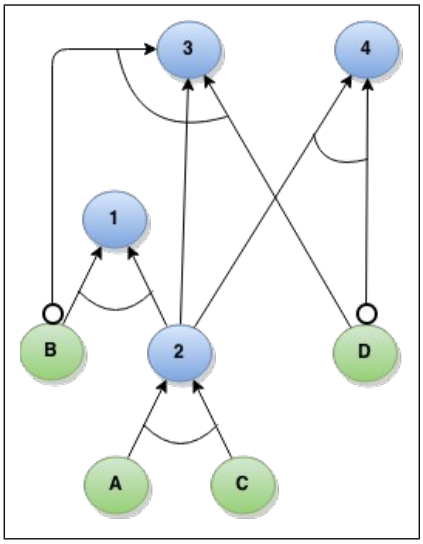


Figura 8.1. Grafo de causalidad BCG.

Símbolo	Significado
A	bcg-aplicada = TRUE

Símbolo	Significado
B	edad < 7 días
C	edad < 5 años
D	edad < 1 año
1	reportar-esquema-optimo = TRUE
2	aplicar-bcg = TRUE
3	reportar-bcg-esquema-alternativo = TRUE
4	reportar-bcg-fuera-esquema = TRUE

Tabla 8.4. Tabla de símbolos

VIII.4.2 Hepatitis B

La vacuna contra la Hepatitis B se compone de *tres dosis*, y su esquema se define de la siguiente manera:

- Si la **1ra dosis** se aplica **antes de la 12 horas de vida**, entonces se inicia como un **esquema normal**.
- Si la **1ra dosis** es aplicada **entre las 12 horas y el mes de vida**, entonces se inicia **fuera de esquema**.
- Si el niño tiene la **1ra dosis aplicada antes del mes**, el esquema puede ser completado (2da y 3ra dosis) mediante la aplicación de la **Quíntuple (Pentavalente)**.
- A partir de los **11 años de edad**, es posible iniciar o completar el esquema en caso de ser necesario. La **2da dosis** debe ser aplicada con **un mes** de diferencia de la 1ra, y la **3ra dosis**, con **6 meses** de diferencia de la 2da.
- Las dosis aplicadas a los **11 años** se reportan dentro de un **esquema normal**.
- Toda dosis aplicada **entre los 12 y 17 años** debe ser reportada como **fuera de esquema**.
- Las dosis que se apliquen a **mayores de 18 años** se reportan en un **esquema alternativo**.

La Tablas 8.5, 8.6 y 8.7 presentan las tablas de decisión discriminadas por cada dosis de la vacuna.

Hepatitis B - Primera Dosis					
Edad del Beneficiario	menor de 12 horas	entre 12 horas y un mes	11 años	entre 12 y 17 años	18 años o más
Aplicada 1ra Dosis Hepatitis B	No	No	No	No	No
Aplicada 2da Dosis Hepatitis B	No	No	No	No	No
Aplicada 3ra Dosis Hepatitis B	No	No	No	No	No
Aplicada 2da Dosis Quíntuple	No	No	No	No	No
Aplicada 3ra Dosis Quíntuple	No	No	No	No	No
Acciones					
Aplicar 1ra Dosis Hepatitis B	Sí	Sí	Sí	Sí	Sí
Reportar Esquema	Normal	Fuera de Esquema	Esquema Normal	Fuera de Esquema	Esquema Alternativo

Tabla 8.5 Tabla de decisiones de la primera dosis de la Hepatitis B

Hepatitis B - Segunda Dosis			
Edad del Beneficiario	11 años	entre 12 y 17 años	18 años o más
Aplicada 1ra Dosis Hepatitis B	Sí	Sí	Sí
Aplicada 2da Dosis Hepatitis B	No	No	No
Aplicada 3ra Dosis Hepatitis B	No	No	No
Aplicada 2da Dosis Quíntuple	No	No	No
Aplicada 3ra Dosis Quíntuple	No	No	No
Acciones			
Aplicar 2da Dosis Hepatitis B	Sí	Sí	Sí
Reportar Esquema	Esquema Normal	Fuera de Esquema	Esquema Alternativo

Tabla 8.6. Tabla de decisiones de la segunda dosis de la Hepatitis B

Hepatitis B - Tercera Dosis						
Edad del Beneficiario	11 años	entre 12 y 17 años	18 años o más	11 años	entre 12 y 17 años	18 años o más
Aplicada 1ra Dosis Hepatitis B	Sí	Sí	Sí	Sí	Sí	Sí
Aplicada 2da Dosis Hepatitis B	Sí	Sí	Sí	No	No	No
Aplicada 3ra Dosis Hepatitis B	No	No	No	No	No	No
Aplicada 2da Dosis Quíntuple	No	No	No	Sí	Sí	Sí
Aplicada 3ra Dosis Quíntuple	No	No	No	No	No	No
Acciones						
Aplicar 3ra Dosis Hepatitis B	Sí	Sí	Sí	Sí	Sí	Sí
Reportar Esquema	Esquema Normal	Fuera de Esquema	Esquema Alternativo	Esquema Normal	Fuera de Esquema	Esquema Alternativo

Tabla 8.7. Tabla de decisiones de la tercera dosis de la Hepatitis B

La Figura 8.2 y la Tabla 8.8 presentan el grafo de causalidad resultante, y sus símbolos respectivamente.

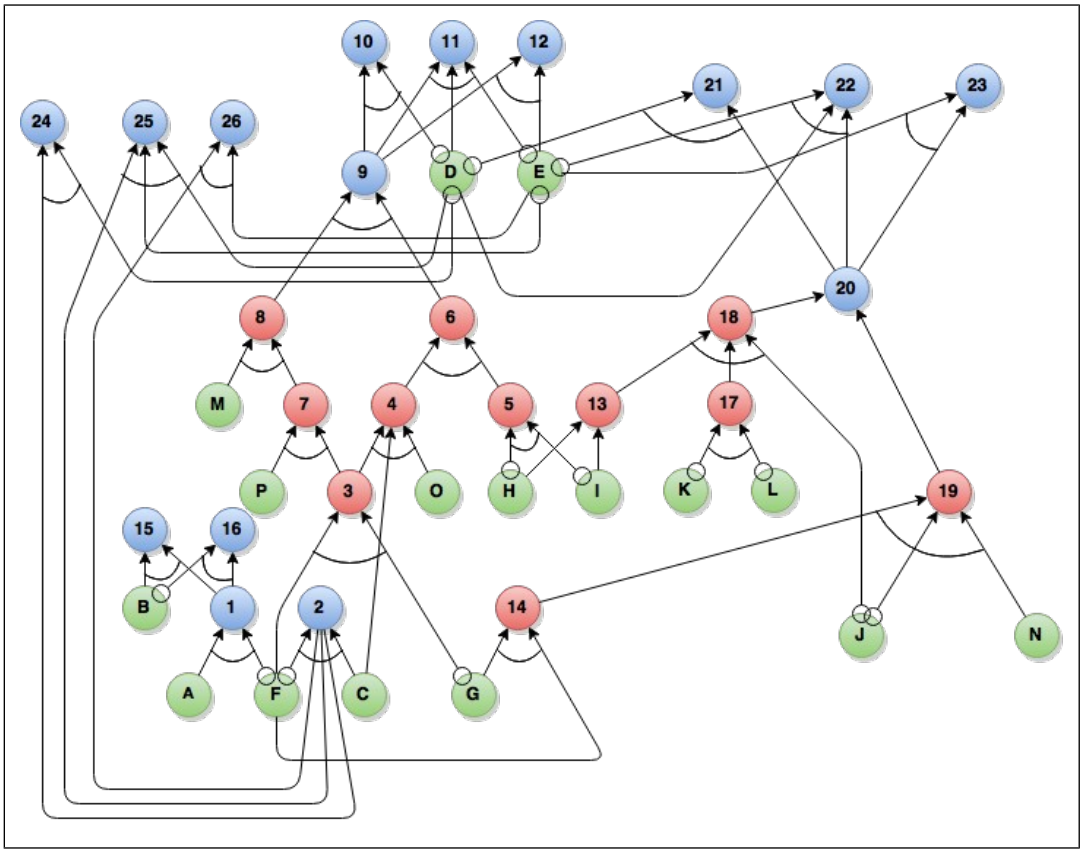


Figura 8.2. Grafo de causalidad Hepatitis B.

Símbolo	Significado
A	Edad < 1 mes
B	Edad < 12 horas
C	Edad >= 11 años
D	Edad >= 12 años
E	Edad >= 18 años
F	hb-1-aplicada = TRUE
G	hb-2-aplicada = TRUE
H	hb-2c-aplicada = TRUE
I	dpthb-2-aplicada = TRUE
J	hb-3-aplicada = TRUE
K	hb-3c-aplicada = TRUE
L	dpthb-3-aplicada = TRUE
M	DIFF(fecha-inmunizacion-hb-1, fecha-actual) > 1 mes

Símbolo	Significado
N	DIFF(fecha-inmunizacion-hb-2, fecha-actual) > 1 mes
O	DIFF(fecha-nacimiento, fecha-inmunizacion-hb-1) < 1 mes
P	DIFF(fecha-nacimiento, fecha-inmunizacion-hb-1) >= 11y
1	aplicar-hb-1-recien-nacido = TRUE
2	aplicar-hb-1-mayores = TRUE
3	esquema-hb-2-completo = FALSE
4	chequear-dpthb-2 = TRUE
5	esquema-dpthb-2-completo = FALSE
6	continuar-esquema-dpthb-2 = TRUE
7	chequear-diferencia-hb-1 = TRUE
8	continuar-esquema-hb-2 = TRUE
9	aplicar-hb-2 = TRUE
10	reportar-hb-2-optimo = TRUE
11	reportar-hb-2-fuera-esquema = TRUE
12	reportar-hb-2-alternativo = TRUE
13	esquema-dpthb-2-completo = TRUE
14	esquema-hb-2-completo = TRUE
15	reportar-hb-1-optimo = TRUE
16	reportar-hb-1-fuera-esquema = TRUE
17	esquema-dpthb-3-completo = FALSE
18	completar-esquema-dpthb-3 = TRUE
19	completar-esquema-hb-3 = TRUE
20	aplicar-hb-3 = TRUE
21	reportar-hb-3-optimo = TRUE
22	reportar-hb-3-fuera-esquema = TRUE
23	reportar-hb-3-alternativo = TRUE
24	reportar-hb-1-dosis-pediatrica = TRUE
25	reportar-hb-1-fuera-esquema = TRUE
26	reportar-hb-1-adultos = TRUE

Tabla 8.8. Símbolos del grafo de causalidad de la Hepatitis B

VIII.4.3 SABIN

La vacuna SABIN es la vacuna contra la **poliomielitis** que se aplica de manera oral. Consta de cuatro dosis y un refuerzo, y su aplicación de estas posee las siguientes restricciones:

- Idealmente, la **1ra dosis** debe iniciar el esquema **entre los 2 y 4 meses** de vida.
- La **2da dosis** debe ser aplicada **a partir de los 4 hasta los 6 meses** del niño, y con una diferencia mínima de **un mes** con respecto a la aplicación de la primera.
- Una **3ra dosis** se aplica **entre los 6 y los 18 meses** de edad, y también debe tener una diferencia mínima de **un mes** con respecto a la anterior.
- La aplicación de la **4ta dosis** debe hacerse **entre los 18 y 24 meses**, de edad, y con una diferencia mínima de **seis meses** con respecto a la tercera.
- El **refuerzo** debe ser aplicado a los **5 o 6 años de vida**; durante el ingreso escolar.
- Todas las restricciones antes descriptas configuran el **esquema normal** de aplicación de la vacuna SABIN.
- **Fuera de esquema**, puede aplicarse esta vacuna **entre los 2 y los 18 años**, pero se excluye, en este caso, el **refuerzo**. Sólo corresponderían las las cuatro dosis, con los mismos períodos de separación mínimos descriptos previamente.

Sabin - Primera Dosis			
Edad del Beneficiario	`entre 2 y 4 meses`	`entre 2 y 18 años`	`mayor a 18 años`
Aplicada 1ra Dosis Sabin	`No`	`No`	`Cualesquiera`
Acciones			
Aplicar 1ra Dosis Sabin	`Sí`	`Sí`	`No`
Reportar Esquema	`Esquema Normal`	`Fuera de Esquema`	`NO APLICA`

Tabla 8.9. Tabla de decisiones de la primera dosis de la Sabin

Sabin - Segunda Dosis			
Edad del Beneficiario	entre 4 y 6 meses	entre 2 y 18 años	mayor a 18 años
Aplicada 1ra Dosis Sabin	Sí	Sí	Cualesquiera
Aplicada 2da Dosis Sabin	No	No	Cualesquiera
Diferencia Fecha Aplicación 1ra Dosis	mayor o igual a 1 mes	mayor o igual a 1 mes	Cualesquiera
Acciones			
Aplicar 2da Dosis Sabin	Sí	Sí	No
Reportar Esquema	Esquema Normal	Fuera de Esquema	NO APLICA

Tabla 8.10. Tabla de decisiones de la segunda dosis de la Sabin

Sabin - Tercera Dosis			
Edad del Beneficiario	entre 6 y 18 meses	entre 2 y 18 años	mayor a 18 años
Aplicada 1ra Dosis Sabin	Sí	Sí	Cualesquiera
Aplicada 2da Dosis Sabin	Sí	Sí	Cualesquiera
Aplicada 3ra Dosis Sabin	No	No	Cualesquiera
Diferencia Fecha Aplicación 2da Dosis	mayor o igual a 1 mes	mayor o igual a 1 mes	Cualesquiera
Acciones			
Aplicar 3ra Dosis Sabin	Sí	Sí	No
Reportar Esquema	Esquema Normal	Fuera de Esquema	NO APLICA

Tabla 8.11. Tabla de decisiones de la tercera dosis de la Sabin

Sabin - Cuarta Dosis			
Edad del Beneficiario	entre 18 y 24 meses	entre 2 y 18 años	mayor a 18 años
Aplicada 1ra Dosis Sabin	Sí	Sí	Cualesquiera
Aplicada 2da Dosis Sabin	Sí	Sí	Cualesquiera
Aplicada 3ra Dosis Sabin	Sí	Sí	Cualesquiera
Aplicada 4ta Dosis Sabin	No	No	Cualesquiera
Diferencia Fecha Aplicación 3ra Dosis	mayor o igual a 1 mes	mayor o igual a 1 mes	Cualesquiera
Acciones			
Aplicar 4ta Dosis Sabin	Sí	Sí	No
Reportar Esquema	Esquema Normal	Fuera de Esquema	NO APLICA

Tabla 8.12. Tabla de decisiones de la cuarta dosis de la Sabin

Sabin - Refuerzo		
Edad del Beneficiario	entre 5 y 6 años	menor que 5 o mayor que 6 años
Aplicada 1ra Dosis Sabin	Sí	Cualesquiera
Aplicada 2da Dosis Sabin	Sí	Cualesquiera
Aplicada 3ra Dosis Sabin	Sí	Cualesquiera
Aplicada 4ta Dosis Sabin	Sí	Cualesquiera
Edad Aplicación 4ta Dosis Sabin	menor que 24 meses	Cualesquiera
Acciones		
Aplicar Refuerzo Sabin	Sí	No
Reportar Esquema	Esquema Normal	NO APLICA

Tabla 8.13. Tabla de decisiones del refuerzo de la Sabin

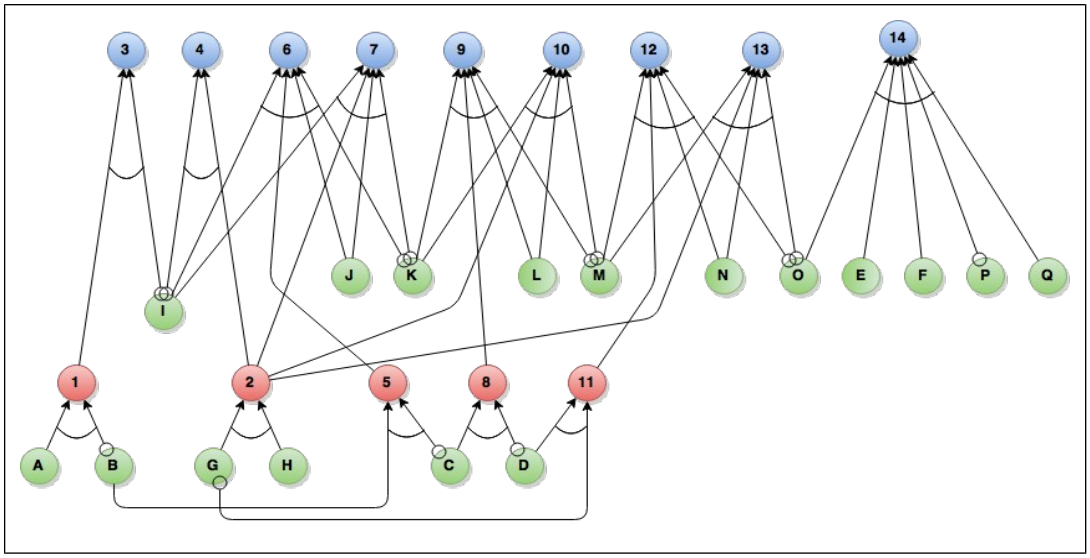


Figura 8.3. Grafo de causalidad SABIN

Símb.	Significado
A	Edad >= 2 meses
B	Edad >= 4 meses
C	Edad >= 6 meses
D	Edad >= 18 meses
E	Edad >= 5 años
F	Edad <= 6 años
G	Edad >= 2 años
H	Edad <= 18 años
I	sabin-1-aplicada = TRUE
J	DIFF(sabin-1-fecha-inmunizacion, fecha-actual) >= 1 mes
K	sabin-2-aplicada = TRUE
L	DIFF(sabin-2-fecha-inmunizacion, fecha-actual) >= 1 mes
M	sabin-3-aplicada = TRUE
N	DIFF(sabin-3-fecha-inmunizacion, fecha-actual) >= 6 meses
O	sabin-4-aplicada = TRUE
P	sabin-99-aplicada = TRUE
Q	Edad-inmunizacion-sabin-4 < 2 años
1	tiene-2-meses = TRUE
2	tiene-12-18-anios = TRUE
3	aplicar-sabin-1 = TRUE
4	aplicar-sabin-1-fuera-esquema = TRUE
5	tiene-4-meses = TRUE
6	aplicar-sabin-2 = TRUE
7	aplicar-sabin-2-fuera-esquema = TRUE
8	tiene-6-meses = TRUE
9	aplicar-sabin-3 = TRUE
10	aplicar-sabin-3-fuera-esquema = TRUE
11	tiene-18-meses = TRUE
12	aplicar-sabin-4 = TRUE
13	aplicar-sabin-4-fuera-esquema = TRUE
14	aplicar-sabin-99 = TRUE

Tabla 8.14. Tabla de decisiones del refuerzo de la Sabin

VIII.4.4 Quíntuple (Pentavalente)

Esta vacuna previene la *Difteria*, la *Tos Convulsa*, *Tétanos*, *Hepatitis B*, y la *Influenza B*, y consta de **tres dosis**:

- La **1ra dosis** debe ser aplicada, idealmente, **entre los 2 y 4 meses** de vida para iniciar el esquema.
- La **2da dosis** ha de ser aplicada **entre los 4 y los 6 meses** del niño, y con una diferencia mínima de **un mes** con respecto a la aplicación de la primera.
- Finalmente, la **3ra dosis** se aplica **entre los 6 y los 18 meses** de edad, y esta también debe tener una diferencia mínima de **un mes** con respecto a la anterior.
- Las condiciones antes descriptas configuran el **esquema normal** de la inmunización. Las dosis recibidas en cualquier otra edad se consideran como un **esquema alternativo**.
- El esquema de la pentavalente puede ser completado hasta los **5 años** del niño. No debe aplicarse dosis alguna pasada esta edad.
- De no estar disponible una dosis de la **Quíntuple**, puede ser aplicada una de la **Cuádruple** junto con una de **Hepatitis B**.

Quíntuple - Primera Dosis			
Edad del Beneficiario	entre 2 y 4 meses	entre 4 meses y 5 años	menor que 2 meses o mayor que 5 años
Aplicada 1ra Dosis Quíntuple	No	No	Cualesquiera
Acciones			
Aplicar 1ra Dosis Quíntuple	Sí	Sí	No
Reportar Esquema	Esquema Normal	Esquema Alternativo	NO APLICA

Tabla 8.15. Tabla de decisiones de la primera dosis de la Quíntuple

Quíntuple - Segunda Dosis			
Edad del Beneficiario	`entre 4 y 6 meses`	`entre 6 meses y 5 años`	`menor que 4 meses o mayor que 5 años`
Aplicada 1ra Dosis Quíntuple	`Sí`	`Sí`	`Cualesquiera`
Aplicada 2da Dosis Quíntuple	`No`	`No`	`Cualesquiera`
Diferencia Fecha Aplicación 1ra Dosis	`mayor o igual a 1 mes`	`mayor o igual a 1 mes`	`Cualesquiera`
Acciones			
Aplicar 2da Dosis Quíntuple	`Sí`	`Sí`	`No`
Reportar Esquema	`Esquema Normal`	`Esquema Alternativo`	`NO APLICA`

Tabla 8.16. Tabla de decisiones de la segunda dosis de la Quíntuple

Quíntuple - Tercera Dosis			
Edad del Beneficiario	`entre 6 y 18 meses`	`entre 18 meses y 5 años`	`menor que 6 meses o mayor que 5 años`
Aplicada 1ra Dosis Quíntuple	`Sí`	`Sí`	`Cualesquiera`
Aplicada 2da Dosis Quíntuple	`Sí`	`Sí`	`Cualesquiera`
Aplicada 3ra Dosis Quíntuple	`No`	`No`	`Cualesquiera`
Diferencia Fecha Aplicación 2da Dosis	`mayor o igual a 1 mes`	`mayor o igual a 1 mes`	`Cualesquiera`
Acciones			
Aplicar 2da Dosis Quíntuple	`Sí`	`Sí`	`No`
Reportar Esquema	`Esquema Normal`	`Esquema Alternativo`	`NO APLICA`

Tabla 8.17. Tabla de decisiones de la tercera dosis de la Quíntuple

A continuación se presenta el grafo de causalidad de la vacuna. El código asignado es `dpthb`.

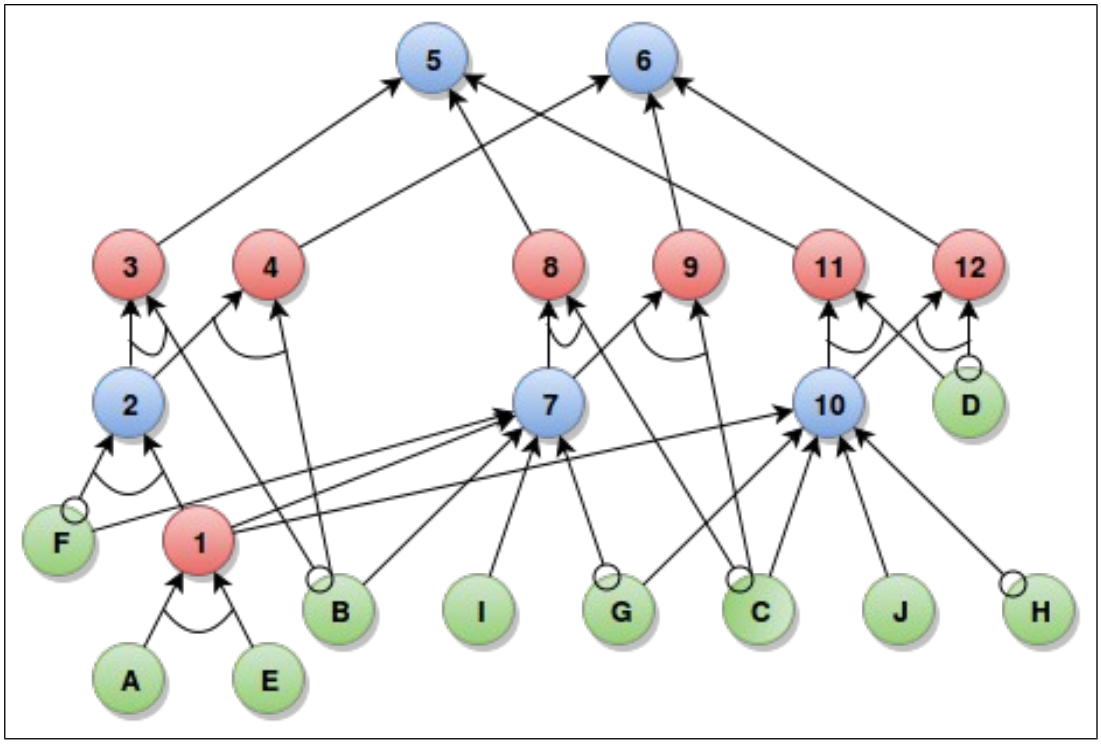


Figura 8.4. Grafo de causalidad Quíntuple.

Símbolo	Significado
A	edad >= 2M
B	edad >= 4M
C	Edad >= 6 meses
D	Edad <= 18 meses
E	Edad < 5 años
F	dpthb-1-aplicada = TRUE
G	dpthb-2-aplicada = TRUE
H	dpthb-3-aplicada = TRUE
I	DIFF(fecha-inmunizacion-dpthb-1, fecha-actual) >= 1 mes
J	DIFF(fecha-inmunizacion-dpthb-2, fecha-actual) >= 1 mes
1	tiene-edad-dpthb
2	aplicar-dpthb-1
3	reportar-dpthb-1-optimo = TRUE
4	reportar-dpthb-1-alternativo = TRUE
5	reportar-dpthb-esquema-optimo = "Aplicación Óptima"
6	reportar-dpthb-esquema-alternativo = "Aplicación Esquema Alternativo
7	aplicar-dpthb-2 = TRUE
8	reportar-dpthb-2-optimo = TRUE
9	reportar-dpthb-2-alternativo = TRUE
10	aplicar-dpthb-3 = TRUE
11	reportar-dpthb-3-optimo = TRUE
12	reportar-dpthb-3-alternativo = TRUE

Tabla 8.18 Tabla de símbolos del grafo de causalidad de la Quíntuple

VIII.4.5 Cuádruple

La vacuna cuádruple tiene la finalidad de prevenir la *Difteria*, la *Tos Convulsa*, *Tétanos*, e *Influenza B*, es decir, las mismas patologías que la **Quíntuple**, excluida la *Hepatitis B*. Su **esquema normal** consta de tan sólo una dosis que funciona como una *cuarta dosis* del esquema de esta, y debe ser aplicada **entre los 15 y los 18 meses** de vida.

Sin embargo, en un **esquema alternativo**, puede ser aplicada junto con una dosis de la **Hepatitis B** como una alternativa la **Quíntuple**. En este caso en particular, ambas

vacunas poseen exactamente las misma restricciones. En el ámbito de este proyecto, este último esquema no será considerado.

Cuádruple - Primera Dosis		
Edad del Beneficiario	`entre 15 y 18 meses`	`menor que 15 meses o mayor que 18 meses`
Aplicada 1ra Dosis Quíntuple	`Sí`	`Cualesquiera`
Aplicada 2da Dosis Quíntuple	`Sí`	`Cualesquiera`
Aplicada 3ra Dosis Quíntuple	`Sí`	`Cualesquiera`
Acciones		
Aplicar Dosis Cuádruple	`Sí`	`No`
Reportar Esquema	`Esquema Normal`	`NO APLICA`

Tabla 8.19. Tabla de decisiones de la Cuádruple

La Figura de a continuación presenta el grafo de casualidad para la vacuna Cuádruple, usando como código para esta, el símbolo `dpth`.

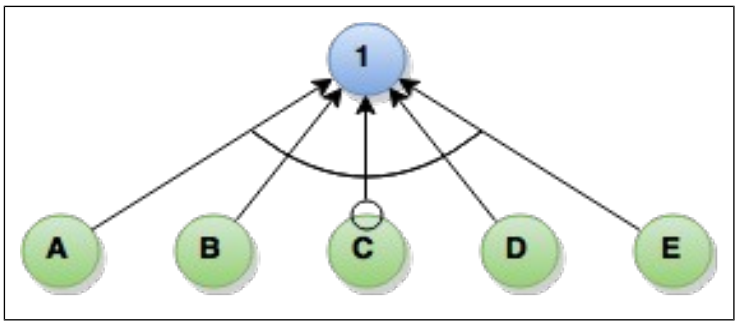


Figura 8.5. Grafo de causalidad Cuádruple.

Símbolo	Significado
A	`Edad >= 15 meses`
B	`Edad <= 18 meses`
C	`dpth-99-aplicada = TRUE`
D	`dpthb-3-aplicada = TRUE`
E	`DIFF( fecha-inmunizacion-dpthb-3, fecha-actual ) >= 6 meses`
1	`aplicar-dpth-99 = TRUE`

Tabla 8.20. Tabla de símbolos del grafo de causalidad de la Cuádruple

VIII.4.6 Neumococo Conjugada (13 Valente)

La vacuna Antineumococo Conjugada previene la *Meningitis*, *Neumonía*, y *Sepsis por Neumococo*. Se compone de **tres dosis**, y su esquema de aplicación tiene las siguientes condiciones:

- Para iniciar el esquema, la **1ra dosis** debe ser aplicada **a partir de los 2 meses** de vida del niño.
- La **2da dosis** debe ser aplicada **a partir de los 4 meses** de edad, y con una **diferencia mínima de dos meses** con respecto a la primera.
- Finalmente, la **3ra dosis** debe aplicarse **desde los 12 meses**, y debe tener una **diferencia mínima de dos meses** con respecto a la segunda aplicación.
- El esquema debe completarse **antes de los 18 meses** del niño. No se puede realizar aplicación alguna a partir de esta edad.

Neumococo Conjugada - Primera Dosis		
Edad del Beneficiario	`mayor o igual que 2 meses`	`mayor que 18 meses`
Aplicada 1ra Dosis Neumococo Conj.	`No`	`Cualesquiera`
Acciones		
Aplicar 1ra Dosis Neumococo Conj.	`Sí`	`No`
Reportar Esquema	`Esquema Normal`	`NO APLICA`

Tabla 8.21. Tabla de decisiones de la primera dosis de la Neumococo Conjugada

Neumococo Conjugada - Segunda Dosis		
Edad del Beneficiario	`mayor o igual que 4 meses`	`mayor que 18 meses`
Aplicada 1ra Dosis Neumococo Conj.	`Sí`	`Cualesquiera`
Aplicada 2da Dosis Neumococo Conj.	`No`	`Cualesquiera`
Diferencia Fecha Aplic. 1ra Dosis	`mayor o igual a 2 meses`	`Cualesquiera`
Acciones		
Aplicar 2da Dosis Neumococo Conj.	`Sí`	`No`
Reportar Esquema	`Esquema Normal`	`NO APLICA`

Tabla 8.22. Tabla de decisiones de la segunda dosis de la Neumococo Conjugada

Neumococo Conjugada - Tercera Dosis		
Edad del Beneficiario	`mayor o igual que 12 meses`	`mayor que 18 meses`
Aplicada 1ra Dosis Neumococo Conj.	`Sí`	`Cualesquiera`
Aplicada 2da Dosis Neumococo Conj.	`Sí`	`Cualesquiera`
Aplicada 3ra Dosis Neumococo Conj.	`No`	`Cualesquiera`
Diferencia Fecha Aplic. 2da Dosis	`mayor o igual a 2 mes`	`Cualesquiera`
Acciones		
Aplicar 3ra Dosis Neumococo Conj.	`Sí`	`No`
Reportar Esquema	`Esquema Normal`	`NO APLICA`

Tabla 8.23. Tabla de decisiones de la tercera dosis de la Neumococo Conjugada

En la Figura 8.6 que aparece abajo se define el grafo de causalidad para la vacuna Neumococo Conjugada, utilizando `neumo` como símbolo dentro de las reglas.

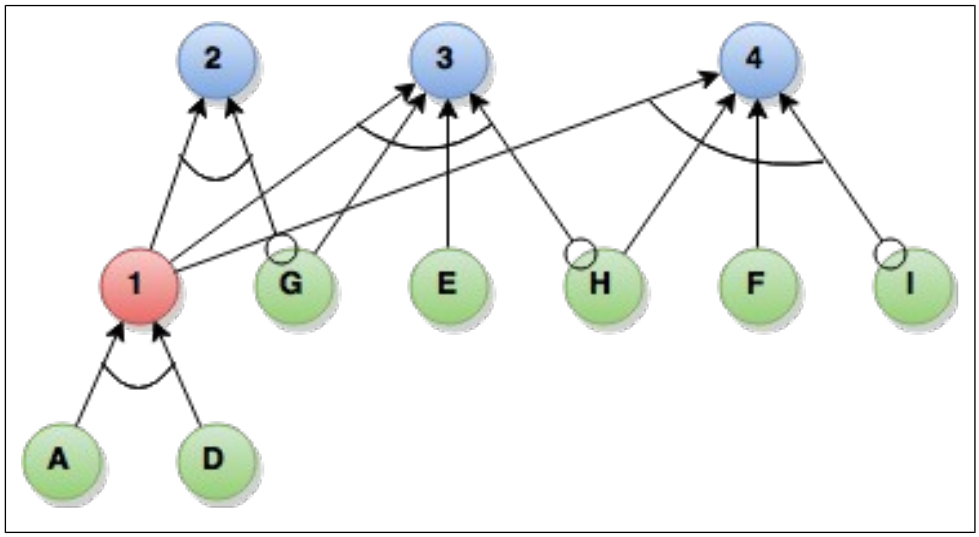


Figura 8.6. Grafo de Causalidad Neumococo Conjugada

Símbolo	Significado
A	edad >= 2 meses
B	edad >= 4 meses
C	edad >= 12 meses
D	edad < 18 meses
E	DIFF(fecha-inmunizacion-neumo-1, fecha-actual) >= 2 meses
F	DIFF(fecha-inmunizacion-neumo-2, fecha-actual) >= 2 meses
G	neumo-1-aplicada = TRUE
H	neumo-2-aplicada = TRUE
I	neumo-3-aplicada = TRUE
1	tiene-edad-neumo = TRUE
2	aplicar-neumo-1 = TRUE
3	aplicar-neumo-2 = TRUE
4	aplicar-neumo-3 = TRUE

Tabla 8.24. Tabla de símbolos del grafo de causalidad de la Neumococo Conjugada

VIII.4.7 Triple Viral

Esta vacuna previene el *Sarampión*, la *Rubeola* y *Paperas*, y su esquema se divide en dos dosis.

- La **1ra dosis** debe ser aplicada a los **12 meses** del niño.
- La **2da dosis** se aplica a los **5 o 6 años** de edad, durante el ingreso escolar. De no estar disponible esta vacuna, es posible aplicar una dosis de **Doble Viral** en su lugar.
- Como **esquema alternativo**, pueden aplicarse la **2da dosis** a los **11 años** de edad, si no hubiera recibido dos dosis de **Triple Viral**, o una de **Triple Viral** más una de **Doble Viral**.

Triple Viral - Primera Dosis		
Edad del Beneficiario	`12 meses`	`menor o mayor que 12 meses`
Aplicada 1ra Dosis Triple Viral	`No`	`Cualesquiera`
Acciones		
Aplicar 1ra Dosis Triple Viral	`Sí`	`No`
Reportar Esquema	`Esquema Normal`	`NO APLICA`

Tabla 8.25. Tabla de decisiones de la primera dosis de la Triple Viral

Triple Viral - Segunda Dosis			
Edad del Beneficiario	`5 o 6 años`	`11 años`	`distinto a 5, 6 u 11 años`
Aplicada 1ra Dosis Triple Viral	`Sí`	`Sí`	`Cualesquiera`
Aplicada 2da Dosis Triple Viral	`No`	`No`	`Cualesquiera`
Acciones			
Aplicar 2da Dosis Triple Viral	`Sí`	`Sí`	`No`
Reportar Esquema	`Esquema Normal`	`Esquema Alternativo`	`NO APLICA`

Tabla 8.26. Tabla de decisiones de la segunda dosis de la Triple Viral

La Figura 8.7 muestra el grafo de causalidad de la Triple Viral (`srp`) con las restricciones de edad antes expuestas.

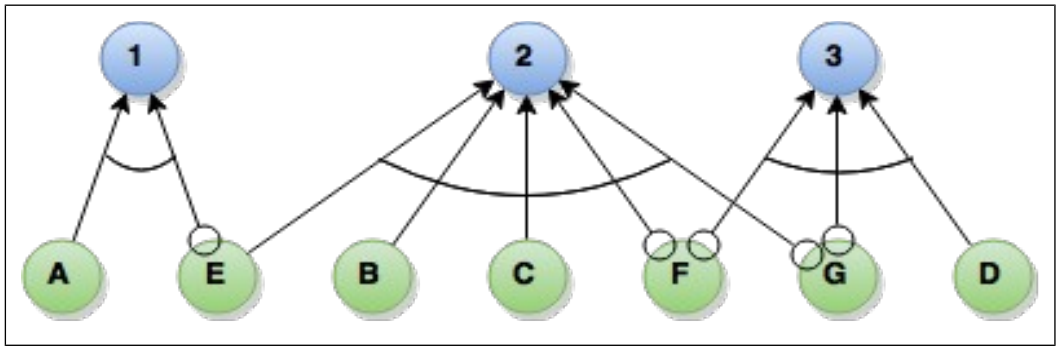


Figura 8.7. Grafo de Causalidad Triple Viral

Símbolo	Significado
A	Edad = 12 meses
B	Edad >= 5 años
C	Edad <= 6 años
D	Edad = 11 años
E	srp-1-aplicada = TRUE
F	srp-2-aplicada = TRUE
G	sr-0-aplicada = TRUE
1	aplicar-srp-1 = TRUE
2	aplicar-srp-2 = TRUE
3	aplicar-srp-2-esquema-alternativo = TRUE

Tabla 8.27. Tabla de símbolos de la Triple Viral

VIII.4.8 Doble Viral

La vacuna Doble Viral es una vacuna contra el *Sarampión* y la Rubeola. Consta de una **dosis única**, y debe ser aplicada a **adultos**, **puérperas** y **personal de salud** que no han completado el esquema de la **Triple Viral**. Debido a que este proyecto sólo abarca *restricciones relativas a la edad*, y no las *asociadas al grupo de riesgo* de la persona que recibe la inmunización, estas dos últimas condiciones no serán tomadas en cuenta.

La dosis de la Doble Viral También puede ser aplicada como una **alternativa** a la **segunda dosis** del esquema de la **Triple Viral**.

Doble Viral		
Edad del Beneficiario	mayor o igual a 18 años	menor que 18 años
Aplicada 2da Dosis Doble Viral	No	Cualesquiera
Acciones		
Aplicar Dosis Doble Viral	Sí	No
Reportar Esquema	Esquema Normal	NO APLICA

Tabla 8.28. Tabla de decisiones de la Doble Viral

El grafo de esta vacuna (`sr`) resulta relativamente escueto, dadas las condiciones de edad arriba presentadas.

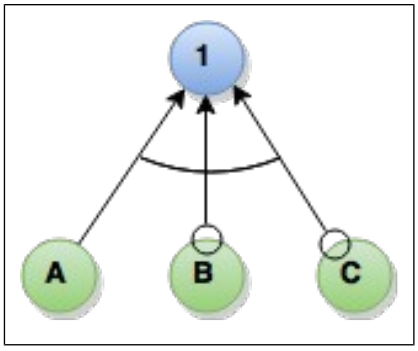


Figura 8.8. Grafo de Causalidad Doble Viral

Símbolo	Significado
A	Edad >= 18 años
B	srp-2-aplicada = TRUE
C	sr-0-aplicada = TRUE
1	aplicar-sr-0 = TRUE

Tabla 8.29. Tabla de símbolos de la Doble Viral

VIII.4.9 Triple Bacteriana Celular

Esta vacuna previene la *Difteria*, el *Tétanos* y la *Tos Convulsa*, y su esquema posee una **dosis única** que debe ser aplicada como **2do refuerzo** a la **Quíntuple** a los **5 o 6 años** de edad. No se puede vacunar a niños de **7 años o más**.

Triple Bacteriana Celular		
Edad del Beneficiario	5 o 6 años	menor que 5 o mayor que 6 años
Aplicada Dosis Cuádruple	Sí	Cualesquiera
Aplicada Dosis Triple Bacteriana	No	Cualesquiera
Acciones		
Aplicar Dosis Triple Bacteriana	Sí	No
Reportar Esquema	Esquema Normal	NO APLICA

Tabla 8.30. Tabla de decisiones de la Triple Bacteriana Celular

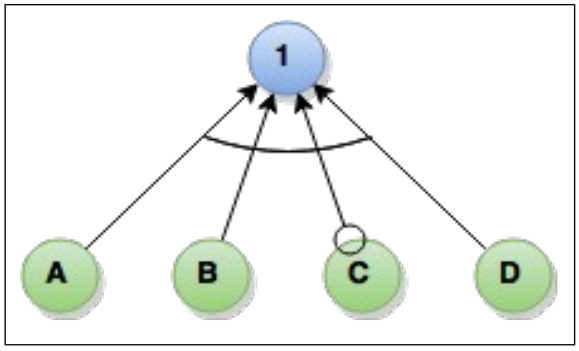


Figura 8.9. Grafo de Causalidad Triple Bacteriana Celular

Símbolo	Significado
A	Edad >= 5 años
B	Edad <= 6 años
C	dpt-99-aplicada = TRUE
D	dpth-99-aplicada = TRUE
1	aplicar-dpt-99 = TRUE

Tabla 8.31. Tabla de símbolos de la Triple Bacteriana Celular

VIII.4.10 Triple Bacteriana Acelular

La vacuna Triple Bacteriana Acelular previene la *Difteria*, el *Tétanos* y la *Tos Convulsa Acelular*, y posee un esquema de una **dosis única** a modo de **refuerzo** para la **Quíntuple**. Esta dosis debe ser aplicada a niños de **11 años** de edad, **personal de salud** que atiene a **menores de 1 año**, o **mujeres embarazadas** a partir de la **semana 20 de gestación**.

Nuevamente, en el alcance de este proyecto quedan excluidas las condiciones relacionadas con los **grupos de riesgo** (personal de saluda y mujeres embarazadas, en este caso), por lo que sólo se tomará en cuenta a los niños con 11 años de edad.

Triple Bacteriana Acelular		
Edad del Beneficiario	11 años	menor o mayor que 11 años
Aplicada Dosis Triple Bacteriana	Sí	Cualesquiera
Aplicada Dosis Triple Bacteriana Ac.	No	Cualesquiera
Acciones		
Aplicar Dosis Triple Bacteriana Ac.	Sí	No
Reportar Esquema	Esquema Normal	NO APLICA

Tabla 8.32.. Tabla de decisiones de la Triple Bacteriana Acelular

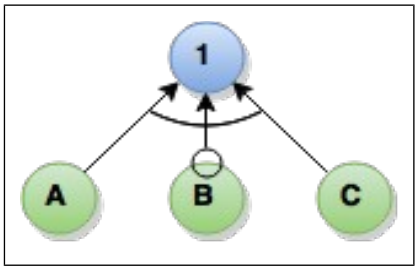


Figura 8.10. Grafo de Causalidad Triple Bacteriana Acelular

Símbolo	Significado
A	Edad = 11 años
B	dpta-99-aplicada = TRUE
C	dpt-99-aplicada = TRUE
1	aplicar-dpta-99 = TRUE

Tabla 8.33.. Tabla de símbolos del grafo de causalidad de la Triple Bacteriana Acelular

VIII.4.11 Hepatitis A

La vacuna contra la Hepatitis A posee una **dosis única** que debe ser aplicada a los **12 meses** de edad.

Hepatitis A		
Edad del Beneficiario	12 meses	menor o mayor que 12 meses
Aplicada Hepatitis A	No	Cualesquiera
Acciones		
Aplicar Hepatitis A	Sí	No
Reportar Esquema	Esquema Normal	NO APLICA

Tabla 8.34. Tabla de símbolos del grafo de causalidad de la Hepatitis A

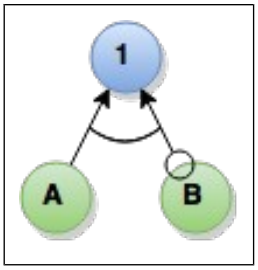


Figura 8.11. Grafo de Causalidad Hepatitis A

Símbolo	Significado
A	edad = 12 meses
B	ha-0-aplicada = TRUE
1	aplicar-ha-0 = TRUE

Tabla 8.35.. Tabla de símbolos del grafo de causalidad de la Hepatitis A

VIII.4.12 VPH

Esta vacuna previene el **Virus del Papiloma Humano**, y se aplica solamente a las **niñas de 11 años** de edad. Su esquema se compone de **tres dosis** con las siguientes condiciones:

- La **2da dosis** debe ser aplicada con una **diferencia de un mes** de la aplicación de la **1ra dosis**.
- La **3ra dosis** debe aplicarse **a partir de los 3 meses** de la aplicación de la **2da dosis**.

VPH - Primera Dosis		
Edad del Beneficiario	11 años	menor o mayor que 11 años
Aplicada VPH 1ra Dosis	No	Cualesquiera
Acciones		
Aplicar VPH 1ra Dosis	Sí	No
Reportar Esquema	Esquema Normal	NO APLICA

Tabla 8.36. Tabla de decisiones de la primera dosis de la VPH

VPH - Segunda Dosis		
Edad del Beneficiario	11 años	menor o mayor que 11 años
Aplicada VPH 1ra Dosis	Sí	Cualesquiera
Aplicada VPH 2da Dosis	No	Cualesquiera
Diferencia Fecha Aplic. 1ra Dosis	mayor o igual a 1 mes	Cualesquiera
Acciones		
Aplicar VPH 2da Dosis	Sí	No
Reportar Esquema	Esquema Normal	NO APLICA

Tabla 8.37. Tabla de decisiones de la segunda dosis de la VPH

VPH - Tercera Dosis		
Edad del Beneficiario	11 años	menor o mayor que 11 años
Aplicada VPH 1ra Dosis	Sí	Cualesquiera
Aplicada VPH 2da Dosis	Sí	Cualesquiera
Aplicada VPH 3ra Dosis	No	Cualesquiera
Diferencia Fecha Aplic. 2da Dosis	mayor o igual a 3 meses	Cualesquiera
Acciones		
Aplicar VPH 3ra Dosis	Sí	No
Reportar Esquema	Esquema Normal	NO APLICA

Tabla 8.38. Tabla de decisiones de la tercera dosis de la VPH

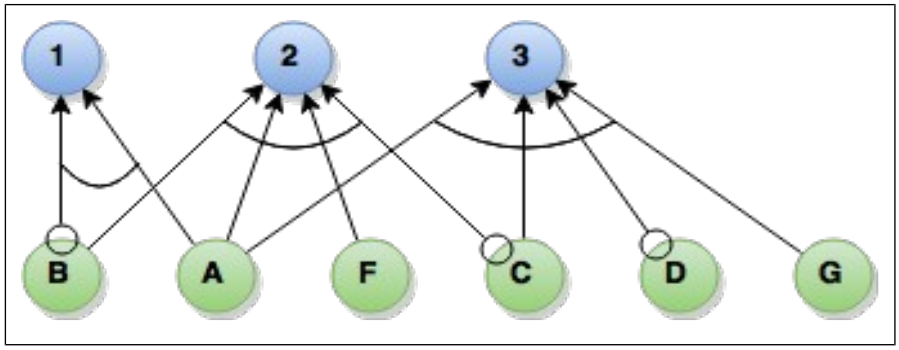


Figura 8.12. Grafo de causalidad VPH.

Símbolo	Significado
A	edad = 11 años
B	vph-1-aplicada = TRUE
C	vph-2-aplicada = TRUE
D	vph-3-aplicada = TRUE
F	DIFF(fecha-inmunizacion-vph-1, fecha-actual) >= 1 meses
G	DIFF(fecha-inmunizacion-vph-2, fecha-actual) >= 3 meses
1	aplicar-vph-1 = TRUE
2	aplicar-vph-2 = TRUE
3	aplicar-vph-3 = TRUE

Tabla 8.39. Diccionario de símbolos del grafo de causalidad de la VPH

VIII.5 Formalización

La formalización del *SEVAC* se realizó mediante la creación de **reglas de producción** o **reglas de inferencia**, las cuales se desprenden de los grafos de causalidad obtenidos en la conceptualización. A modo de ejemplo, se muestra la regla correspondiente al grafo de causalidad de la vacuna BCG (Figura 8.1)

```
IF NOT(bcg-aplicada = TRUE) AND edad < 5 años THEN
    SET aplicar-bcg = TRUE

IF aplicar-bcg = TRUE AND edad < 7 dias THEN
    SET reportar-esquema-normal = TRUE

IF aplicar-bcg = TRUE AND NOT(edad < 7 dias) AND edad < 1 año
THEN
    SET reportar-bcg-esquema-alternativo = TRUE

IF aplicar-bcg = TRUE AND NOT(edad < 1 año) THEN
    SET reportar-bcg-fuera-esquema = TRUE
```

Todas las reglas de producción se muestran en el Anexo C.

VIII.6 Programación e Implementación sobre Jess

La programación del *SEVAC* se hizo mediante la traducción de las reglas de inferencia definidas para cada vacuna (ver Anexo C), a código en lenguaje de *Jess*. Para cada vacuna se creó un archivo `.clp`, que no es otra cosa que un *script* que contiene las reglas de inferencia correspondientes. Por ejemplo, para la vacuna **Triple Bacteriana Acelular**, se tiene un archivo denominado `dpta.clp`, que contiene lo que aparece en el extracto siguiente.

```
( defrule puede-aplicar-dpta
  ( edad-anios ?e &: ( = ?e 11 ) )
  ( dpt-99-aplicada TRUE )
  ( not ( dpta-99-aplicada TRUE ) )
  =>
  ( printout t "Aplicar Triple Bacteriana Acelular" crlf )
)

( provide dpta )
```

Dentro del proyecto existen tres archivos especiales:

- `fechahora.clp`: Define funciones para cálculos de fecha, hora e intervalos.
- `edades.clp`: Calcula, utilizando las funciones de `fechahora.clp`, la edad del beneficiario con diferentes unidades (años, meses, semanas, días, horas).
- `vacunas.clp`: Es el *script* principal y el encargado de ejecutar el *SEVAC*.

Todo el código fuente y pruebas del sistema se encuentra en el repositorio público **https://bitbucket.org/rgmiranda/sevac**, y ha sido versionado mediante el sistema de control de versiones **Mercurial**.

VIII.7 Implementación con *PyJesslet*

Con el fin de analizar el desempeño del *PyJesslet* sobre un sistema experto completo, probar la carga de reglas de inferencias y hechos, y su ejecución, el *SEVAC* fue implementado mediante esta aplicación de escritorio. Durante este proceso, el ingreso de las reglas de inferencias especificadas resultó ser relativamente directa. Vale recordar que los patrones en los hechos en el antecedente y las acciones susceptibles de

ser realizadas en el consecuente, fueron restringidos para simplificar su representación en los mensajes SOAP (ver capítulo VII). Es por ello que la adaptación de las reglas definidas en el sistema de sugerencia de vacunas se hizo realizando ciertas modificaciones mínimas sobre las reglas.

En primer lugar, todos los hechos en el Jesslet deben representarse como **hechos ordenados de valor único**, por lo que algunos patrones en los antecedentes y ciertas acciones en el consecuente tuvieron que ser cambiados en oportunamente. Por ejemplo, puede encontrarse acciones para la creación de hechos de la forma `(assert (aplicar-bcg))`, como se presenta en el extracto siguiente.

```
(defrule
  ; ...
  =>
  ; ...
  (assert (aplicar-bcg))
  ; ...
  )
```

En el Jesslet, esta acción se transforma de manera tal que el valor asignado a este hecho es el booleano `TRUE`.

```
(defrule
  ; ...
  =>
  ; ...
  (assert (aplicar-bcg TRUE))
  ; ...
  )
```

Como resultado de esto, el antecedente en donde aparece este hecho:

```
(defrule
  ; ...
  (aplicar-bcg)
  ; ...
  =>
  ; ...
  )
```

En el caso del Jesslet, este patrón se altera de la siguiente manera:

```
(defrule
  ; ...
  ?a-fact <- (aplicar-bcg ?a-value&:(= ?a-value TRUE))
  ; ...
  =>
  ; ...
  )
```

Por otro lado, se ha llevado a cabo una *simplificación conceptual* en relación a los antecedentes para la activación de ciertas reglas de inferencia. En el sistema de sugerencia, por ejemplo, para indicar que la primera dosis de la vacuna SABIN había sido aplicada se deben declarar los hechos `sabin-1-aplicada` y `fecha-inmunizacion-sabin-1`, el primero con valor booleano `TRUE`, el segundo con la fecha de la inmunización.

```
(assert (sabin-1-aplicada TRUE))
(assert (fecha-inmunizacion-sabin-1 2015-09-13T10:24:00))
```

En la implementación sobre el Jesslet, las reglas que se activan con estos patrones han sido modificadas para utilizar un único hecho de la forma `<vacuna>-<dosis>`, que almacena la fecha de la inmunización. La existencia de este hecho implicaría que la dosis a sido aplicada, y la no existencia se asume como la no aplicación de la misma. El ejemplo anterior quedaría resumido:

```
(assert (sabin-1 2015-09-13T10:24:00))
```

En el fragmento siguiente se muestra un resultado de una ejecución definiendo como único hecho inicial la fecha de nacimiento de la persona.

```
Restableciendo el proyecto FOO...

El proyecto a sido restablecido.
Agregado: <1> (fecha-nacimiento 2015-10-01T00:00:00)
Ejecutando el proyecto FOO...

Se ha terminado con la ejecución del programa.
Aplicar Quíntuple, 1ra Dosis (Esquema Normal)
Aplicar BCG (Esquema de Hasta 1 Año)
```

```
Aplicar Neumococo Conjugada, 1ra Dosis
Aplicar SABIN 1ra Dosis (Esquema Normal)
```

Si además de la fecha de nacimiento, se crea un hecho correspondiente a la primera dosis de la vacuna SABIN, la ejecución se presentaría sin la recomendación correspondiente.

```
Restableciendo el proyecto FOO...

El proyecto a sido restablecido.
Agregado: <1> (fecha-nacimiento 2015-10-01T00:00:00)
Agregado: <2> (sabin-1 2015-12-05T00:00:00)
Ejecutando el proyecto FOO...

Se ha terminado con la ejecución del programa.
Aplicar Quíntuple, 1ra Dosis (Esquema Normal)
Aplicar BCG (Esquema de Hasta 1 Año)
Aplicar Neumococo Conjugada, 1ra Dosis
```

Estos resultados, devueltos por *PyJesslet*, con las sugerencias de vacunas demuestran la versatilidad que se consigue al implementar como un servicio web el shell *Jess*.

VIII.7.1 Cliente de Inmunizaciones: *Chrolie*

El *Chrolie* es una extensión para el navegador **Google Chrome** escrita en lenguaje **Javascript**, la cual fue creada con dos objetivos en mente. Por un lado, pretende demostrar el funcionamiento del *Jesslet* desde una plataforma liviana y sin las capacidades de otros ambientes de ejecución, y por el otro, sirve como un caso de prueba para acceder al conocimiento del *SEVAC* en sí. Para tal fin, la extensión permite buscar una persona en la base de datos de inmunizaciones, recuperar sus registros de aplicaciones de vacunas, y, utilizando estos datos, realizar una consulta sobre la instancia del *SEVAC* en el *Jesslet* (presentada en el apartado anterior). La fecha de nacimiento y las de las aplicaciones de vacunas previamente registradas serán los hechos utilizados para la ejecución. La Figura 8.13 presenta un diagrama conceptual en donde aparecen las interacciones del *Chrolie* con estos sistemas externos.

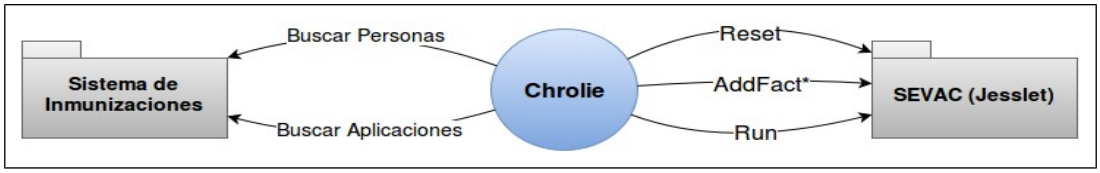


Figura 8.13. Diagrama Conceptual del Funcionamiento del Chrolie.

El *sistema de inmunizaciones* se encarga de proveer al *Chrolie* de los datos necesarios, tanto de las personas, como de las inmunizaciones recibidas por ellas. Las búsquedas sobre este sistema se lleva a cabo mediante un *servicio web REST*[1], utilizando *JSON*[2]como formato para la transferencia de los datos. En la Figura 8.14 puede verse la vista de una búsqueda usando el número de documento `52000000`.

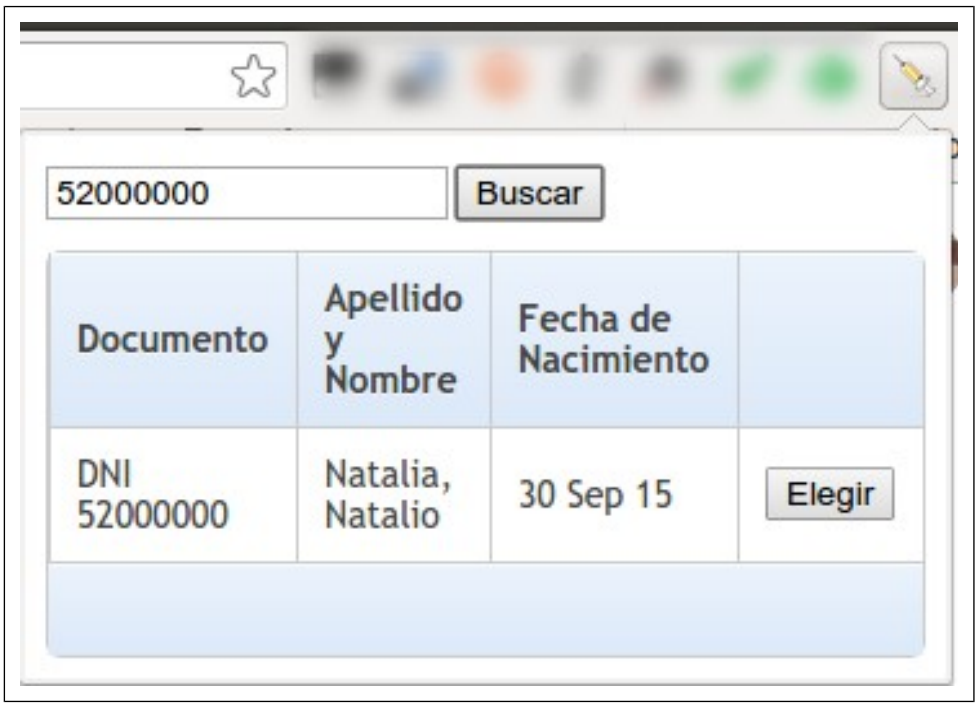


Figura 8.14. Vista de la Búsqueda de Personas.

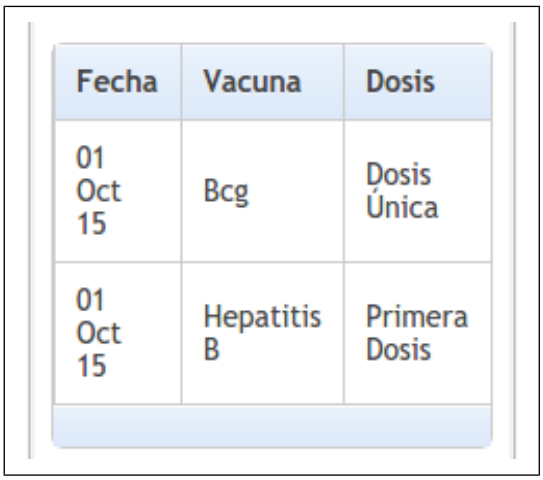
1. REST (Representational State Transfer) es un estilo de arquitectura para la creación de servicios web en donde los recursos son accesibles mediante representaciones textuales, y permite la ejecución de operaciones carentes de estado. Comúnmente, esto se hace mediante el protocolo HTTP, utilizando sus cuatro verbos: GET, POST, PUT, DELETE para las operaciones ABML.

2. JSON (JavaScript Object Notation) es un formato ligero de texto para el intercambio de datos, el cual se encuentra basado en la notación de objetos del lenguaje Javascript.

Tras bambalinas, los datos de la búsqueda se devuelven en el formato JSON, el cual se muestra en el siguiente fragmento.


```
[
  {
    "id": 38646,
    "nombre": "Natalio",
    "apellido": "Natalia",
    "tipo_documento": "dni",
    "documento": "52000000",
    "fecha_nacimiento": "2015-10-01",
    "fecha_alta": "2015-12-17 17:22:12.731667"
  }
]
```


El usuario puede elegir la persona sobre la cual se realizará la consulta. La razón de ser de este mecanismo de selección es el hecho de que pueden existir, en la base de datos, números de documentos duplicados, con lo cual devolver la primera persona encontrada puede devenir en resultados inesperados o equívocos. Una vez escogida la persona de la consulta, *Chrolie* busca las inmunizaciones que esta ha recibido anteriormente. La Figura 8.15 que aparece a continuación muestra estos resultados.



Fecha	Vacuna	Dosis
01 Oct 15	Bcg	Dosis Única
01 Oct 15	Hepatitis B	Primera Dosis

Figura 8.15. Resultados de Recupero de Aplicaciones.

Nuevamente, para este caso, los datos aún se obtienen desde la base de datos de inmunizaciones se transfieren en formato JSON.


```
[
  {
```

```
    "id": 47678,
    "fecha_inmunizacion": "2015-10-01",
    "fecha_alta": "2015-12-17 17:24:13.861095-03",
    "id_beneficiario": 38646,
    "id_vacuna": 2,
    "id_dosis": 2,
    "lote": "xxxx",
    "nombre_beneficiario": "Natalio",
    "apellido_beneficiario": "Natalia",
    "tipo_documento": "dni",
    "documento": "52000000",
    "vacuna": "BCG",
    "codigo_vacuna": "bcg",
    "codigo_dosis": "0",
    "dosis": "Dosis Única",
  },
  {
    "id": 47680,
    "fecha_inmunizacion": "2015-10-01",
    "fecha_alta": "2015-12-17 17:26:14.802725-03",
    "id_beneficiario": 38646,
    "id_vacuna": 3,
    "id_dosis": 3,
    "lote": "xxxx",
    "nombre_beneficiario": "Natalio",
    "apellido_beneficiario": "Natalia",
    "tipo_documento": "dni",
    "documento": "52000000",
    "vacuna": "Hepatitis B",
    "codigo_vacuna": "hb",
    "codigo_dosis": "1",
    "dosis": "Primera Dosis",
  }
]
```


Con estos datos recuperados, se crearán los hechos en la memoria de trabajo del Jesslet para la consulta. Puntualmente, para el caso de las inmunizaciones se utilizarán `codigo_vacuna` y `codigo_dosis` para formar el nombre del hecho, y su valor estará dado por `fecha_inmunizacion`. El primer paso antes de la carga de los hechos es la reinicialización con el método `Reset`. Para poder conseguir esto, el *Chrolie* debe generar el mensaje SOAP correspondiente y enviarlo al *SEVAC* para su procesamiento. Para facilitar la creación de estos mensajes en Javascript se optó por utilizar la herramienta *handlebars*, con la cual es posible predefinir *plantillas*, y utilizarlas luego

para la generación de contenido parametrizado. Para el caso del método `Reset`, la plantilla creada es la que aparece en el siguiente fragmento.

```
<SOAP-ENV:Envelope xmlns:ns0="{{namespace}}"
    xmlns:ns1="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
        <ns0:PID>{{pid}}</ns0:PID>
    </SOAP-ENV:Header>
    <ns1:Body>
        <ns0:Reset>
        </ns0:Reset>
    </ns1:Body>
</SOAP-ENV:Envelope>
```

Como puede verse, dentro de la plantilla hay una sección definida entre dos llaves `{{pid}}`. Esto funciona a modo de *marcador de posición*, que será luego reemplazado cuando se genere el contenido final. En este ejemplo, el `pid` no es otra cosa que el ID del proyecto sobre el cual será ejecutado `Reset`. El mensaje producido es el siguiente:

```
<SOAP-ENV:Envelope xmlns:ns0="http://fce.unse.edu.ar/jesslet/"
    xmlns:ns1="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
        <ns0:PID>FOO</ns0:PID>
    </SOAP-ENV:Header>
    <ns1:Body>
        <ns0:Reset>
        </ns0:Reset>
    </ns1:Body>
</SOAP-ENV:Envelope>
```

El mensaje generado para la creación del hecho fecha-nacimiento se muestra en el fragmento a continuación:

```
<SOAP-ENV:Envelope xmlns:ns0="http://fce.unse.edu.ar/jesslet/"
    xmlns:ns1="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
        <ns0:PID>FOO</ns0:PID>
    </SOAP-ENV:Header>
    <ns1:Body>
        <ns0:AssertFact>
            <fact>
                <id></id>
                <name>fecha-nacimiento</name>
                <value xsi:type="ns0:literalExpression">
                    <type>DATETIME</type>
                    <value>2015-10-01T00:00:00</value>
                </value>
            </fact>
        </ns0:AssertFact>
    </ns1:Body>
</SOAP-ENV:Envelope>
```

La creación de los hechos correspondientes a las aplicaciones utiliza mensajes análogos a este último. Una vez que todos los hechos han sido cargados, se invoca al método `Run` para recuperar las recomendaciones (si es que hay alguna). La Figura 8.16 muestra los resultados de las recomendaciones recuperadas.

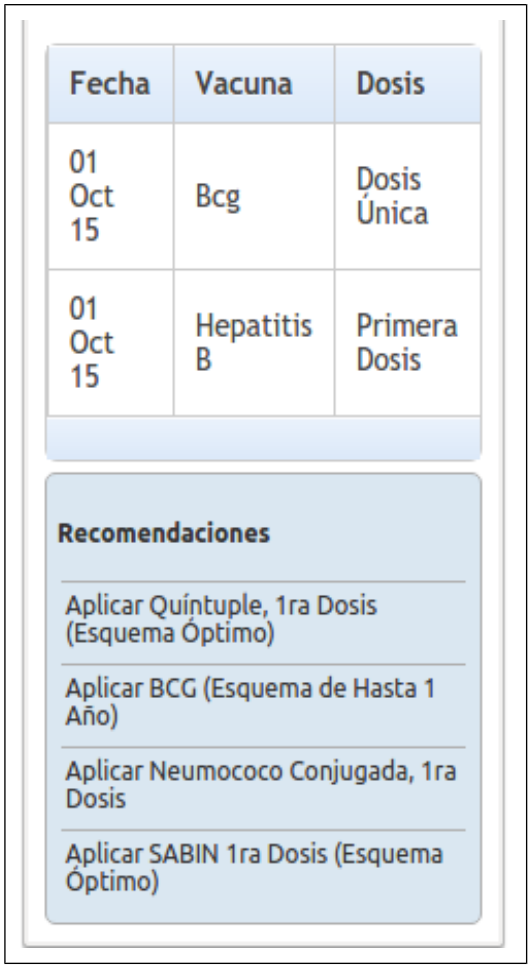


Figura 8.16. Recomendaciones en el Chrolie.

De esta forma, se consigue integrar, sin demasiado esfuerzo, toda la funcionalidad y potencia de un sistema experto completo dentro de una aplicación relativamente liviana y de rápido desarrollo.

Todo el código de **Chrolie** se encuentra bajo el control de versiones **Mercurial**, y está alojado también en el servicio *BitBucket*. Puede ser descargado desde https://bitbucket.org/rgmiranda/chrolie, y su ejecución se realizada en el contexto del navegador **Google Chrome**, en modo de depuración

VIII.8 Resumen

Las reglas definidas en este capítulo para el calendario de vacunación son la base para el desarrollo del sistema experto experimental sobre *Jess*, y su análogo sobre el *Jesslet*. Para cada tipo de vacuna implicada se ha realizado una presentación sobre sus

condiciones de aplicación, situaciones excepcionales (si es que hubiere alguna), y se ha planteado un grafo de causalidad inicial con las reglas de inferencias planteadas en base a esta información. Posteriormente se ha presentado el programa resultante de su codificación en lenguaje de *Jess* para su prueba. Con esto como base, se demostró la implementación del *SEVAC* en el servicio *Jesslet* a través de la herramienta *PyJesslet*. Finalmente, se presentó el *Chrolie* como un caso de prueba para la integración del *Jesslet*, y para el *SEVAC* como sistema experto accesible a través del *Jesslet*.

Capítulo IX Experimentación con el Jesslet

Una vez finalizada la construcción del servicio Jesslet y sus aplicaciones clientes, mediante las métricas antes presentadas (ver capítulo VII), se llevó a cabo el análisis del prototipo con el fin de afirmar o refutar las hipótesis inicialmente planteadas.

IX.1 Clases de Prueba

Esta etapa se ocupó de la programación de una aplicación, llamada ***Jessapp***, con dos *clases* que actúan como una interfaz entre un sistema y *Jess*. Una de estas clases fue programada para **acceder de manera directa** a Jess, mientras que la otra para que hacerlo **a través del Jesslet**. Esta experimentación parte con la idea de que ambas clases posean la misma interfaz (firma de sus métodos) y comportamiento, mientras que su implementación interna fuese distinta, para así poder sobre estas implementaciones, calcular las métricas anteriormente presentadas (ver capítulo VI).

Debido a que Jess se encuentra escrito en el lenguaje Java, para posibilitar un acceso más fluido al shell, la programación de estas clases *interfaz* también se realizó con este lenguaje. Ambas implementaron una `Interface` para garantizar que poseyeran exactamente los mismos métodos, y sobre ambas se ejecutaron las mismas pruebas automáticas y se exigieron exactamente los mismos resultados.

La Figura 9.1 presenta un diagrama con las clases y su relación con la interfaz. La clase `Jess` realiza un *acceso nativo* utilizando la clase `jess.Rete`, entre otras; mientras que la clase `Jesslet` utiliza el servicio web desarrollado para trabajar.

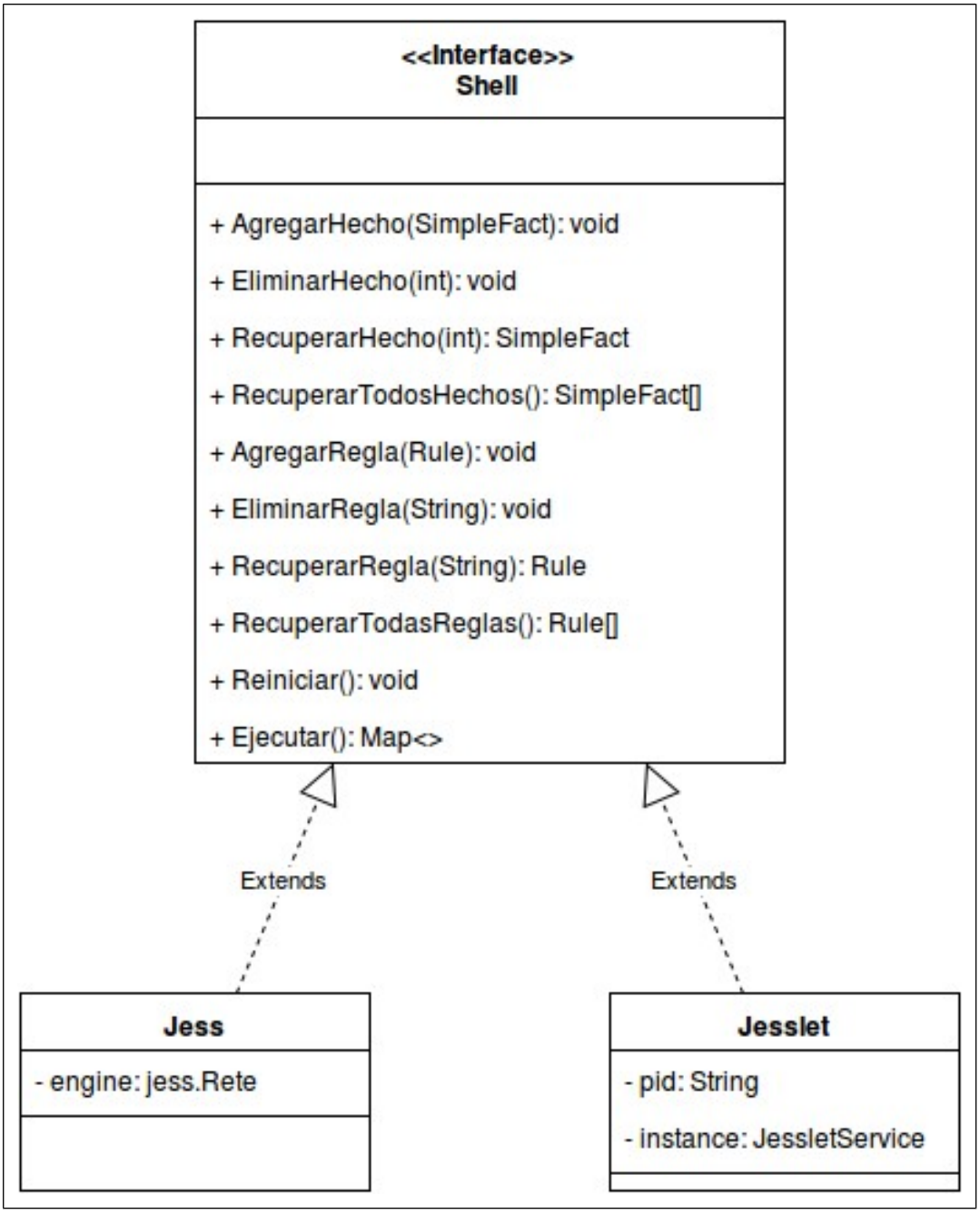


Figura 9.1. Clases para Acceder a Jess

IX.1.1 La Interfaz

La interfaz se denomina `Shell`, y establece el conjunto de métodos que debe poseer cada una de las clases para funcionar como un *punto de comunicación* con Jess. La forma en la que sea llevada a cabo esta comunicación, queda al libre albedrío de cada implementación; ya que la interfaz sólo determina qué parámetros reciben los métodos y los resultados que deberían ser devueltos.

```
public interface Shell {

    public void AgregarHecho(SimpleFact hecho) throws
JessappException;
    public void EliminarHecho(int id) throws JessappException;
    public SimpleFact RecuperarHecho(int id) throws
JessappException;
    public SimpleFact[] RecuperarTodosHechos() throws
JessappException;
    public void AgregarRegla(Rule regla) throws
JessappException;
    public void EliminarRegla(String nombreRegla) throws
JessappException;
    public Rule RecuperarRegla(String nombreRegla) throws
JessappException;
    public Rule[] RecuperarTodasReglas() throws
JessappException;
    public void Reiniciar() throws JessappException;
    public Map<PartesRespuesta, Object> Ejecutar() throws
JessappException;

}
```

El fragmento de código de arriba presenta los métodos establecidos para la interfaz `Shell`. Como puede verse, cada uno de los métodos puede corresponderse con otro método en la interfaz definida para el *Jesslet* (ver capítulo VI). Esto se definió así sólo con los fines de realizar las pruebas de una manera más específica en relación con el *Jesslet*.

IX.1.2 Acceso por el Jesslet

La primera implementación de la interfaz antes presentada es la clase `Jesslet`. la cual interactúa con el servicio web *Jesslet* (ver capítulo VII) para la implementación

del comportamiento de cada método. En realidad, podría verse como una **envoltura** (*wrapper*) del servicio web, ya que, como se mencionó antes, los métodos se corresponden uno a uno.

Para su implementación se recurrió a la herramienta `wsimport`[1], la cual permite importar (generar) las clases necesarias para acceder a un servicio web SOAP, utilizando el documento WSDL correspondiente. Estas clases se corresponden con las involucradas en la interfaz definida para el Jesslet, y sirven para facilitar la invocación de cada uno de los métodos y la creación de los parámetros que estos reciben. Cada clase es *etiquetada* adecuadamente para indicar a cómo ha de ser convertida desde y hacia una representación XML necesaria para ser enviada sobre un mensaje SOAP (ver capítulo 4) Por ejemplo, la clase `Rule` posee las siguientes etiquetas:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "rule", propOrder = {
    "actions",
    "conditions",
    "description",
    "name"
})
public class Rule {
    // Definición de Rule.
}
```

Mediante el parámetro `@XmlType` se indica que esta clase se convertirá en un elemento `<rule>`, y que será compuesto por elementos `<actions>`, `<conditions>`, `<description>`, y `<name>`. Cada uno de los elementos componentes poseen, a su vez, sus propias etiquetas para la conversión a XML.

```
<rule>
    <name><!-- nombre de la regla --></name>
    <description><!-- descripción de la regla
--></description>
    <conditions><!-- condiciones para la activación de regla
--></conditions>
```

1. Se usa para generar los archivos necesarios de JAX-WS para hacer a una servicio web accesible, en base a un documento WSDL.

```
    <actions><!-- acciones del consecuente de la regla
--></actions>
</rule>
```

Todas las clases generadas por `wsimport` pueden ser encontradas bajo el *espacio de nombres* `ar.edu.unse.fce.jesslet` y su subespacio `service`.

IX.1.3 Acceso Directo

Para un acceso *nativo* al shell Jess, se desarrolló la clase `Jess`. Para implementar los métodos de la interfaz `Shell`, esta hace uso de las clases que se encuentran en la biblioteca de Jess (el archivo `jess.jar`), dentro del paquete `jess.*`. El código que se muestra a continuación es un fragmento de las importaciones de esta biblioteca que hace la clase `Jess`.

```
// ...
import jess.Defrule;
import jess.Fact;
import jess.HasLHS;
import jess.JessException;
import jess.RU;
import jess.Rete;
import jess.Value;
import jess.ValueVector;
// ...
```

Por otro lado, las clases generadas por la herramienta `wsimport` fueron adaptadas para poder dar soporte dar soporte al acceso directo a Jess. Esto se hizo debido a que la interfaz `Shell` posee, como parámetros en sus métodos, objetos de tipos generados por esta herramienta. Por ejemplo, la clase `ar.edu.unse.fce.jesslet.Rule` fue dotada de un constructor que admite como parámetros un objeto del tipo `jess.Defrule`, y otro del tipo `jess.Rete`; junto con otro método `generateSource` para la generación de código en lenguaje Jess.

```
// ...
import ar.edu.unse.fce.jesslet.Expression;
import ar.edu.unse.fce.jesslet.Rule;
import ar.edu.unse.fce.jesslet.SimpleFact;
```

```
import ar.edu.unse.fce.jesslet.ValueType;
// ...
```

Para cada clase que aparece importada en el código anterior, se ha programado un método `generateSource`, el cual genera un fragmento de código con las sintaxis de Jess para ser ejecutado sobre la instancia de `jess.Rete` (el motor de inferencias de Jess). Esta instancia puede encontrarse en el atributo privado `engine`. El único caso que no se utiliza `generateSource` para su operación es cuando se necesita agregar un hecho a la memoria de trabajo. Esto se debe a la imposibilidad de recuperar el identificador del hecho de hacerse de esta manera.

IX.1.4 Pruebas

Para llevar a cabo las pruebas sobre las clases `Jess` y `Jesslet` se utilizó la herramienta **JUnit 3** [TAHCHIEV2010] provista dentro del entorno integrado de desarrollo **Netbeans**. Todas las clases de prueba del proyecto se encuentran contenidas en el directorio test.

Para asegurar que tanto `Jess` como `Jesslet` devuelven exactamente los mismos resultados, se creó una clase abstracta denominada `ShellTest` en el espacio de nombres `ar.edu.unse.fce.jessapp`. Esta clase define un conjunto de métodos test* que utilizan un atributo del tipo `Shell` (la interfaz) para ejecutar el conjunto de pruebas. De esta manera, estas pruebas se establecen a un nivel de contrato de interfaz, independientemente de la forma en la que esta sea implementada. Para probar cada implementación en sí, fueron programadas dos clases hijas de `ShellTest`: `JessTest` y `JessletTest`. Ambas clases implementan los métodos `setUp` y `tearDown`, propios de la clase abstracta `junit.framework.TestCase` (clase base de un caso de prueba en JUnit), en donde se crea la instancia de `Shell` para ser usadas en los métodos test*.

La Figura 9.2 presenta el diagrama de las clases creadas para las pruebas, junto con estos métodos `test*` presentes en `ShellTest`.

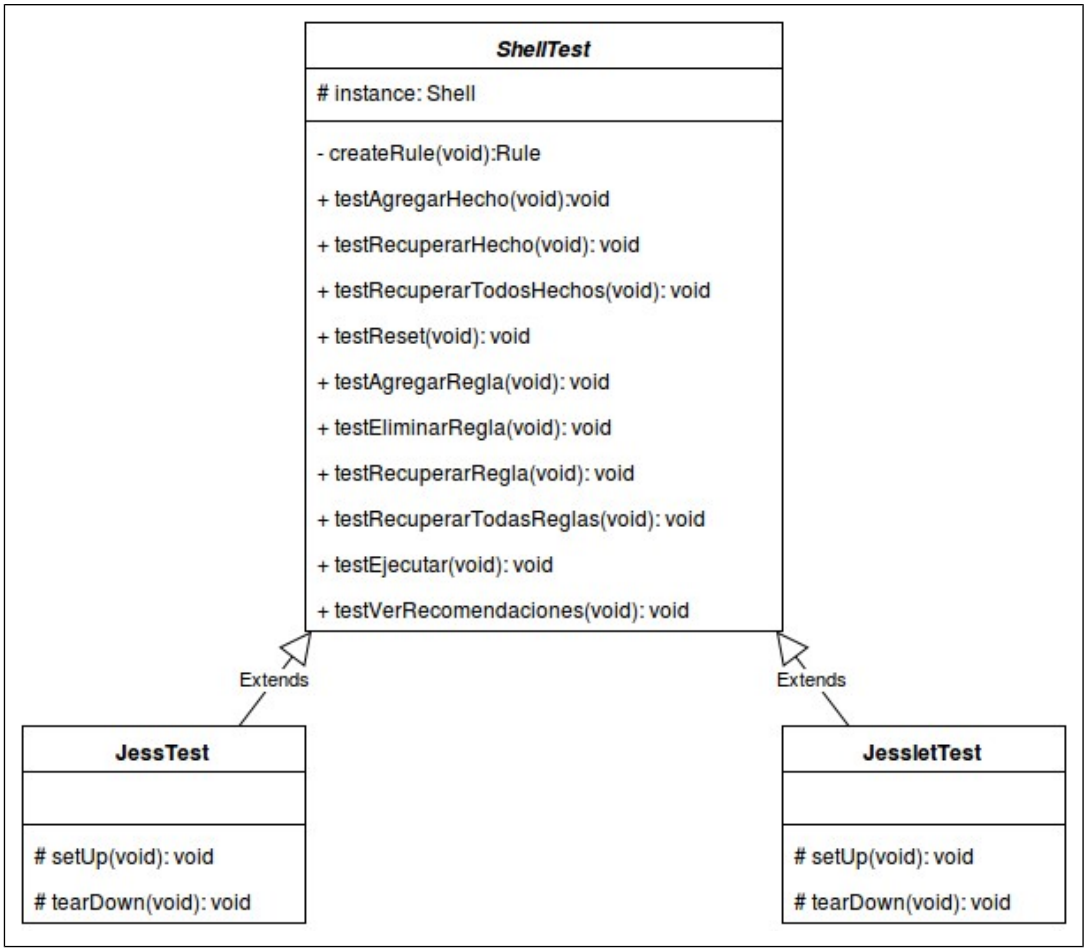


Figura 9.2. Clases de Pruebas de Acceso a Jess

El método `testAgregarHecho` ejecuta pruebas sobre la función `AgregarHecho` de la interfaz `Shell`, el cual inserta un hecho en la memoria de trabajo. Para esto, genera tres hechos (objetos del tipo `SimpleFact`), y los utiliza como parámetros para esta función. `testRecuperarHecho` prueba el método `RecuperarHecho`, el cual agrega hechos de la misma manera que el anterior, y luego los recupera para determinar si esta última operación es llevada a cabo correctamente. De manera semejante, `testRecuperarTodosHechos` carga un conjunto de hechos en la memoria del motor de inferencias, para luego recuperarlos a todos y determinar si han sido guardados correctamente.

Para las pruebas de los métodos relativos a las reglas de inferencias (`AgregarRegla`, `EliminarRegla`, `RecuperarRegla`, y `RecuperarTodasReglas`) se programó una función denominada `createRule`, la

cual genera un instancia de la clase Rule. Se procedió de esta forma debido a que la creación de un objeto de este tipo es un proceso relativamente largo, en el cual deben crearse las condiciones (`FactCondition`) de activación y las acciones (`Action`) a ser realizadas al ser disparada. Cada condición posee, a su vez, un conjunto de una o más expresiones. Por otro lado, cada acción puede tratarse de una creación de un hecho (`AssertAction`), la eliminación de uno (`RemoveAction`), o la presentación de un mensaje (`RecommendAction`). La creación de todos estos objetos se hace en `createRule`. Tanto `testAgregarRegla`, `testEliminarRegla`, `testRecuperarRegla`, como `testRecuperarTodasReglas` utilizan este método privado para la creación del objeto Rule a ser pasado como parámetro de la prueba.

Con `testReset` se testea la invocación del método Reiniciar, el cual limpia la memoria de trabajo, y crea el *hecho inicial 0*. Cabe notar que este método de reinicio no sólo es invocado en esta prueba, sino que aparece también en las pruebas de funciones de manejo de hechos antes mencionadas.

Finalmente, los métodos `testEjecutar` y `testVerRecomendaciones` ejecutan el sistema experto definido a través de `Ejecutar`. La diferencia entre estos es que la segunda prueba está centrada en las recomendaciones devueltas por el sistema tras la ejecución, mientras que la primera sólo controla que el método devuelva un resultado, cualquiera sea. `testVerRecomendaciones` invoca a `createRule`, ya que la regla que este genera imprime una recomendación.

IX.1.5 Métricas de Cohesión y Acoplamiento

Una vez establecidas las implementaciones, tanto para el acceso nativo como para el acceso a través del servicio web, se procedió a llevar a cabo las mediciones de la **cohesión (LCOM4)** y del **acoplamiento (CBO)** (ver capítulo 6) sobre las clases `Jess` y `Jesslet`.

IX.1.5.1 Cohesión

En primer lugar se realiza el cálculo del valor de **LCOM4** de la clase de acceso nativo `Jess`. El diagrama que se presenta en la Figura 9.3 muestra los métodos y

atributos de esta clase y sus relaciones. Como puede verse, todas las funciones se encuentran interconectadas a través del atributo `engine`, lo cual da como resultado un valor LCOM4 igual a 1.

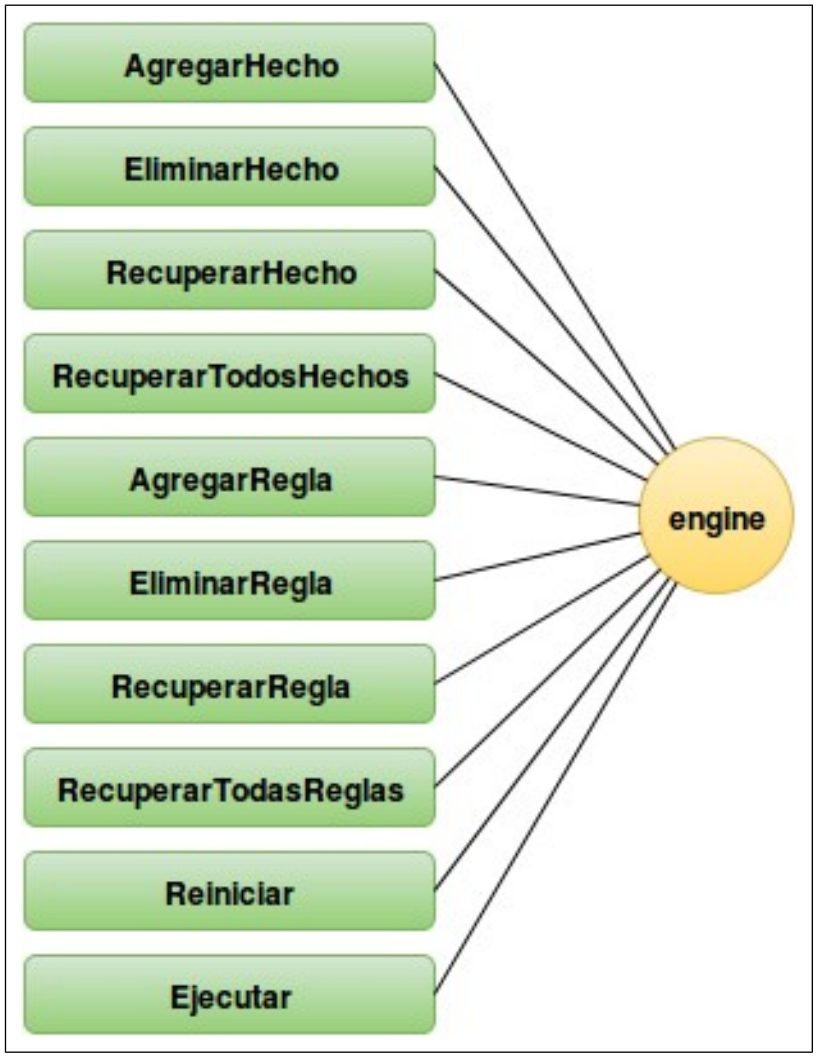


Figura 9.3. LCOM4 de la Clase Jess

Análogamente, una situación similar puede apreciarse al calcular esta métrica sobre la clase `Jesslet`. En la siguiente Figura se presentan las relaciones de los métodos y atributos, con lo que se obtiene un valor LCOM4, nuevamente, igual a 1.

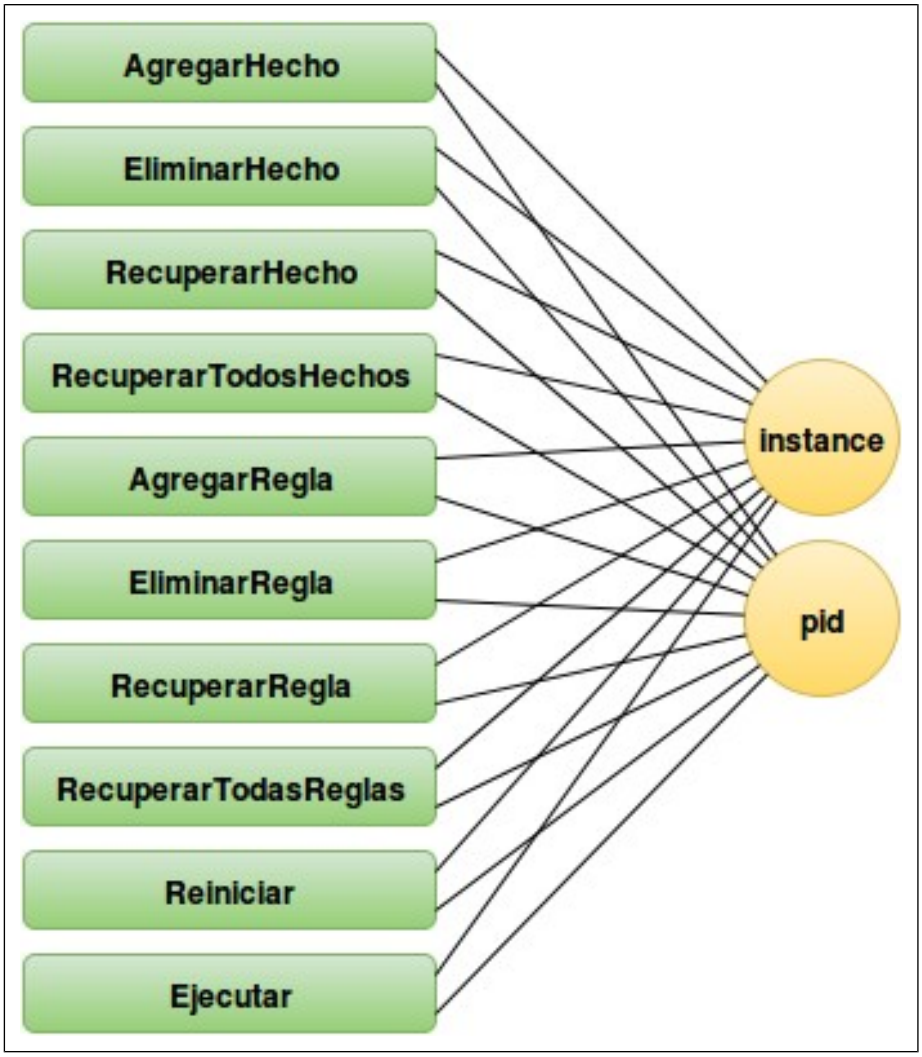


Figura 9.4. LCOM4 de la Clase Jesslet

Esta no varianza de la métrica en ambas clases puede ser una consecuencia directa del hecho de que ambas clases implementan la misma interfaz (`Shell`). Al tener un conjunto de métodos públicos bien definidos, el objetivo de la clase queda perfectamente identificado, razón por la cual resultaría difícil que LCOM4 arroje un valor distinto a 1. Esta es sólo una conjetura, y se encuentra basada en la simple observación del caso en cuestión.

Algo que cabe destacar es que, habiéndose obtenido un valor 1 en el cálculo sobre `Jess`, era esperable obtener el mismo resultado sobre la clase `Jesslet`.

IX.1.5.2 Acoplamiento

Para el cálculo del acoplamiento de una clase se utiliza la métrica **CBO**, la cual contabiliza los **acoples** hacia o desde otras clases. Se entiende por acople una llamada a un método o acceso a un atributo. Cada clase se contabiliza sólo una vez, por lo que si existe una que es accedida o invocada más de una vez, sólo contará como 1.

Una manera sencilla de determinar las clases que utilizan `Jess` y `Jesslet` es mediante las sentencias de importación al principio de cada clase. Para el primer caso, se tiene el fragmento siguiente:

```
// ...
import ar.edu.unse.fce.jessapp.config.PropertiesManager;
import ar.edu.unse.fce.jessapp.exceptions.JessappException;
import ar.edu.unse.fce.jesslet.Expression;
import ar.edu.unse.fce.jesslet.Rule;
import ar.edu.unse.fce.jesslet.SimpleFact;
import ar.edu.unse.fce.jesslet.ValueType;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
import jess.Defrule;
import jess.Fact;
import jess.HasLHS;
import jess.JessException;
import jess.RU;
import jess.Rete;
import jess.Value;
import jess.ValueVector;
// ...
```

Hasta este punto, pueden ser contadas **18 clases acopladas**, sin embargo, dentro del código también pueden ser encontrados accesos e invocaciones a `Shell`, `StringArrayWriter` y `PartesRespuesta`, lo cual arroja un total de **21**. Estas clases no aparecen en la parte de las importaciones debido a que se encuentran en el mismo espacio de nombres (`ar.edu.unse.fce.jessapp`) que la clase `Jess`.

En la clase `Jesslet`, por otro lado, se encuentra el siguiente fragmento:

```
// ...
import ar.edu.unse.fce.jesslet.Rule;
import ar.edu.unse.fce.jesslet.RuleArray;
import ar.edu.unse.fce.jesslet.SimpleFact;
import ar.edu.unse.fce.jessapp.exceptions.JessappException;
import ar.edu.unse.fce.jesslet.Response;
import ar.edu.unse.fce.jesslet.service.JessletFault;
import ar.edu.unse.fce.jesslet.service.JessletService;
import ar.edu.unse.fce.jesslet.service.JessletService_Service;
import java.util.HashMap;
import java.util.Map;
// ...
```

A estas **10 clases** utilizadas, se les adiciona la interfaz `Shell` y la clase `PartesRespuesta`, para un total de **12**. Como puede apreciarse, existe una considerable disminución en el número de clases referenciadas entre la implementación directa y la realizada a través del *Jesslet*. Este descenso era lo esperable para esta métrica, debido a que el servicio web permite la abstracción de las bibliotecas, clases y configuraciones específicas de *Jess*. Esto queda evidenciado más claramente al notar que la única diferencia entre las sentencias de importación de clases de ambas implementaciones son las sentencias en donde se especifican clases dentro de la biblioteca del shell (`jess.*`).

IX.2 Repositorio

La aplicación *Jessapp* puede ser encontrada en la dirección web https://bitbucket.org/rgmiranda/jessapp. Dentro del directorio `src` se encuentra el código fuente para la aplicación con las dos clases usadas para las pruebas: `Jess` y `Jesslet`. Por otro lado, en la carpeta `test` se encuentran ubicadas las *pruebas de unidades* (clases `*Test`) sobre la interfaz `Shell`. Como se mencionó arriba, se utiliza *JUnit 3* para realizar la ejecución de las pruebas, en el entorno de desarrollo *NetBeans*.

IX.3 Resumen

En este capítulo se ha presentado el desarrollo de la aplicación *Jessapp*, la cual tiene el solo objetivo de servir como un ambiente de pruebas para el análisis de las métricas propuestas para el análisis del *Jesslet*. A través de esta, se realizaron los

cálculos correspondientes a la **cohesión** y el **acoplamiento** en una implementación de Jess dentro de otro sistema, tanto de manera **nativa**, como **a través del Jesslet**. Con los resultados obtenidos, se pudo apreciar una evidente disminución en el acoplamiento cuando el acceso se hace a través del Jesslet, mientras que la cohesión se mantuvo sin alteraciones.

Capítulo X Conclusiones

Para el último paso de este trabajo se expondrán distintas conclusiones en lo relativo a varios aspectos del mismo, a saber: el **objetivo general**, los **objetivos específicos**, **cuestiones pendientes** que pueden servir de guía para investigaciones futuras, y, finalmente, una **conclusión final** en donde se hacen apreciaciones finales del trabajo.

X.1 Objetivo General

El objetivo general de este proyecto es **tender a facilitar el uso e implementación de las prestaciones de los sistemas expertos dentro de otros sistemas y/o plataformas**. En base a los resultados obtenidos con el desarrollo del *Jesslet*, y al correcto funcionamiento de las aplicaciones clientes para las pruebas, es posible concluir que este objetivo **ha sido cumplimentado**.

Este trabajo supuso la aleación de dos tecnologías relativamente maduras en el campo de la informática como lo son los *servicios web* y los *sistemas expertos*. Sin embargo, su combinación resultó en un software con diversas posibilidades de aplicación en diferentes campos y disciplinas, con la ventaja de poder ser insertado de manera sencilla en los sistemas del mundo interconectado contemporáneo. Prueba de ello es el sistema **PyJesslet**. El **sistema de recomendación de vacunas SEVAC**, implementado mediante este último, es un claro ejemplo de una solución mediante un sistema experto a un problema concreto. El cliente **Chrolie** utiliza este sistema y sus recomendaciones de inmunizaciones para brindar una herramienta complementaria a las labores de los encargados de la aplicación de vacunas en las instituciones de salud.

X.2 Objetivos Específicos

El primer objetivo específico establecido, el de **eliminar el acoplamiento no deseado entre Jess y otros sistemas al ser usado como componente**, quizás fue demasiado ambicioso en un principio, debido a que el acoplamiento es necesario debido a que los componentes deben comunicarse. Sin embargo, considerando la primera de las

dos hipótesis formuladas en el capítulo VI para este trabajo, es posible afirmar que la misma fue validada. De acuerdo con la métrica **CBO** (ver capítulo VI) obtenida en la *fase de experimentación* (ver capítulo IX), para los casos propuestos, el acoplamiento ha disminuido de **21 unidades** para el acceso directo al shell Jess, a **12 unidades** para el caso de implementación mediante el Jesslet, con lo cual se está cumpliendo con el objetivo de diseño que es el de mantener el nivel del acoplamiento lo mas bajo posible. Esta variación, resulta sensiblemente significativa para mejorar la mantenibilidad, la reutilización, y la detección y corrección de errores de software. Vale destacar que se trata de solo un caso de prueba, por lo que generalizar esta cualidad para todas las aplicaciones puede resultar altamente especulativo y hasta ingenuo. Sin embargo conviene recordar que, conceptualmente hablando, uno de los objetivos fundamentales de los servicios web es la **abstracción** de la tecnología específica para el acceso a una funcionalidad, por lo que una mejora en la métrica de acoplamiento es lo esperable.

El segundo objetivo específico buscaba **incrementar la cohesión de Jess al integrarse dentro de otro sistema**. En relación con esto y considerando ahora la segunda de las dos hipótesis formuladas, es posible afirmar que la misma no fue validada. Para realizar la medición de la *cohesión* se utilizó la métrica **LCOM4** (ver capítulo VI) sobre los mismos componentes tratados para el cálculo del *acoplamiento*. Las recomendaciones y patrones de diseño indican que el valor *LCOM4* no debe ser distinto de 1, ya que este determina el número de *incumbencias* que una clase posee. Tanto el cálculo realizado sobre la clase que accede directamente a Jess como la que lo hace mediante el Jesslet han devuelto este valor, con lo cual la hipótesis de que los servicios web alteran de alguna manera la cohesión no puede ser verificada. No obstante, puede sugerirse un motivo por el cual puede haberse presentado esta situación.

En este caso puntual, la métrica *LCOM4* aplicada tiene el objetivo principal de encontrar la cantidad de ocupaciones (*concerns*) que una clase posee. Y para ello, cuenta la cantidad de conjuntos de métodos interrelacionados dentro de la misma. Con la definición de un contrato (`Interface`) para el servicio web, se consigue que las clases que la implementen posean un conjunto de métodos preestablecidos. La única forma de

que la cohesión varíe de una implementación respecto de otra es que se adicionen nuevos métodos sin relación alguna con los métodos ya definidos en la interfaz.

X.3 Limitaciones y Aspectos Pendientes

Tal vez uno de los mayores retos que presentó la creación del Jesslet fue la abstracción del lenguaje Jess y la correcta representación de las estructuras propias de ese entorno (por ejemplo, reglas, hechos, e invocaciones de funciones) en un mensaje SOAP. Este *shell* define un lenguaje mediante el cual puede realizarse un número ilimitado de acciones como invocar funciones, crear reglas y hechos, definir clases, e importar clases desde bibliotecas Java externas. Todo esto implica la formulación de un nuevo lenguaje completo, lo cual atentaría contra el espíritu y la esencia del proyecto: **abstraer**.

Por estos motivos, y a los fines de aislar y separar los conceptos propios de Jess, tuvieron que realizarse algunos ajustes sobre lo que el servicio permite realizar. Jess permite la definición de dos tipos de hechos: los *hechos ordenados* y los *hechos no ordenados* (ver capítulo III). Para el Jesslet, este conjunto de hechos que es posible manejar tuvo que limitarse a los **hechos ordenados con un único valor** (ver capítulo VI). A futuro, debe considerarse el soporte a los restantes hechos, pero hay que considerar que esto impacta directamente en la definición de las reglas de inferencias. La decisión de limitar los hechos se hizo principalmente debido a la heterogeneidad de los patrones que pueden aparecer en los antecedentes de las reglas de inferencias.

El conjunto de acciones en el consecuente de las reglas de inferencias también tuvo que ser restringido. En esta parte de una regla puede ejecutarse cualquier función en Jess. Ya que se pretendía mantener el foco en el manejo de los hechos en la memoria de trabajo y las recomendaciones para el operador, las únicas acciones que pueden ser especificadas en una regla en el Jesslet son las de **creación de un hecho**, **eliminación de un hecho**, y **presentación de un mensaje** para el usuario (ver capítulo VI). En desarrollos sucesivos, puede ser conveniente ampliar este conjunto para darle al operador mayores opciones a la hora plantear una solución experta. Una de estas acciones puede ser el uso y asignación de valor a *variables* en Jess.

A medida que se avanzaba con el desarrollo se observó que una propiedad de los servicios web sobre SOAP limitaba cierto funcionamiento de los sistemas expertos (ver capítulo VI). Un servicio web en SOAP espera a que un cliente invoque alguno de sus métodos, en donde todas las invocaciones son independientes entre sí (ver capítulo IV). Empero, en muchos sistemas expertos es necesaria una interacción con el operador para la solicitud de datos para la inferencia. Esto no es posible en la versión actual del *Jesslet* debido a que siempre es el cliente quien inicia la solicitud, nunca el servidor. Para un correcto funcionamiento, el sistema debe tener cargados todos los datos antes de la ejecución, ya que no tendrá manera de conseguirlos hasta que este no haya concluido. En otras palabras, se rige bajo el principio de un **universo cerrado** (ver capítulo II). Esta dificultad, teóricamente, puede ser atacada en el futuro mediante el concepto de **sockets**, el cual permite el intercambio de información entre dos programas en dos vías.

Otro detalle que merece ser notado es que la comparación de las métricas ha sido realizada sobre una plataforma y un lenguaje particular: Java. Como se explicó en la etapa de experimentación, su elección no se debió a algo arbitrario, sino a que el shell Jess se encuentra escrito en este lenguaje, lo que permite su acceso como una biblioteca desde otros programas en la misma plataforma. Cabría investigar cuál es el impacto del uso del Jesslet en otras plataformas, de ser posible la implementación directa de la biblioteca de Jess.

Por último, uno de los objetivos tácitos del trabajo era el de **dar a conocer los avances y resultados a través de redes sociales y blogs**. Por desgracia, no fue posible llevar a cabo estas actividades en la forma que se hubiese esperado. Si bien este documento, junto con todos los sistemas que se describen en él, se encuentran bajo una licencia pública y es posible descargarlos desde sus repositorios, la afluencia de entradas en los blogs y publicaciones relacionadas en las redes sociales no ha alcanzado el nivel o la calidad deseable. Vale mencionar que esta última consideración posee un carácter altamente subjetivo. Al tratarse de un objetivo no definido, no están establecidos los parámetros para su verificación, por lo que el enunciar que solo ha sido cumplimentado parcialmente no resulta ser más que una apreciación personal.

X.4 Conclusión Final

Como resultado de la investigación y desarrollo realizados, es posible concluir que el proyecto ha cumplido con los objetivos perseguidos. Además, se ha conseguido profundizar el conocimiento sobre el shell Jess y sus aplicaciones. Esto ha hecho que se conozcan sus limitaciones y proponer formas para mejorarlo. Y el resultado de esas propuestas es el Jesslet; un prototipo funcional de servicio web que abstrae del shell Jess sus funcionalidades, y permite que éstas sean fácilmente incorporables en otros entornos.

Sin embargo, este trabajo se encuentra aún con diversas oportunidades para su mejora. Por un lado, como se mencionó anteriormente, el acceso mediante *SOAP* al servicio web limita en parte las respuestas y la posibilidad de interacción de este con las aplicaciones clientes. Una aproximación para atacar este problema sería la implementación del *Jesslet* mediante *sockets*, los cuales permiten un intercambio más fluido y bidireccional de la información.

Por otro lado, de ser utilizado el *Jesslet* en un ámbito real y de producción, puede que resulte necesario ampliar o revisar el conjunto de reglas que este puede soportar, y, sobre todo, las acciones que pueden aparecer en el consecuente. No obstante, esto implicaría un rediseño de las estructuras de datos fundamentales para la transferencia de reglas y hechos desde el servicio a sus clientes y viceversa. Asimismo puede sugerirse una reestructuración interna para permitir que el servicio pueda utilizar más de un *shell*.

Resulta importante destacar que este proyecto estuvo enmarcado dentro del **Programa de Becas de Estímulo a las vocaciones científicas** (EVC CIN) en el período 2012. Además, se ha participado en el **XVIII Congreso Argentino de Ciencias de la Computación** (CACIC) mediante la presentación de un *paper* homónimo a este trabajo. En octubre y noviembre del 2012, se realizaron exposiciones mediante la modalidad póster sobre el Jesslet en las **VII Jornadas de Ciencia y Tecnología de Facultades de Ingeniería del NOA**, y las en **Jornadas de Becarios UNSE** respectivamente. Finalmente, en octubre del 2017, se participó de la **I Jornada de**

Vinculación con el Medio y Difusión de Resultados de Investigación y Desarrollos en Informática.

Bibliografía

1. [BRIAND2000] Lionel C.Briand, Jürgen Wüst, John W.Daly y D.Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. 2000. Journal of Systems and Software, Vol: 51, No: 3, PP: 245-273.

2. [BOUGUETTAYA2014] Athman Bouguettaya, Quan Z. Sheng , Florian Daniel. Advanced Web Services . Springer 2014. ISBN 978-1-4614-7535-4.

3. [CERAMI2002] Ethan Cerami. Web Services Essentials. O'Reilly 2002. ISBN 0-596-00224-6.

4. [CHANG2010] Chung C Chang, Shih-Chieh Hsieh . A Wireless LAN Problem Diagnosis Expert System Based on Web Services. 2010. International Symposium on Parallel and Distributed Processing with Applications.

5. [CHIDAMBER1991] Shyam R. Chidamber y Chris F. Kemerer. Towards a Matrics Suite for Object Oriented Design. 1991. Sloan School of Management, MIT.

6. [DEMARCO2002] Tom DeMarco. Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency. 2002. Broadway Books. ISBN-13: 978-0932633613

7. [EARL2005] Thomas Earl. Service-Oriented Architecture: Concepts, Technology, and Design. 2005. Prentice Hall. ISBN: 0131858580

8. [FEIGENBAUM1982] Edward A. Feigenbaum. Knowledge Engineering for the 1980s. 1982. Stanford University.

9. [FRIEDMAN2003] Ernest Friedman-Hill. Jess in Action: Ruled-Based Systems in Java. Manning 2003. ISBN 1-930110-89-8.

10. [FRIEDMAN2008] Ernest Friedman-Hill. Jess The Rule Engine for the Java Platform. Sandia National Laboratories.

11. [GARCIA2004] Ramón García Martínez, Paola Britos. Ingeniería de Sistemas Expertos. 2004. Nueva Librería. ISBN:9789871104154.

12. [GIARRATANO1998] Joseph Giarratano, Gary Riley. Sistemas Expertos: Principios y Programación. 1998. Thomson. ISBN-13: 978-0878353354.

13. [HITZ1995] Measuring coupling and cohesion in object oriented systems, Proceedings of the International Symposium on Applied Corporate Computing, 1995, pp. 25-27.

14. [HO2005] K. K. L. Ho, M. Lu. Web-based expert system for class schedule planning using JESS. 2005. 2005 IEEE International Conference on Information Reuse and Integration.

15. [ISO24765]: ISO/IEC/IEEE. ISO/IEC/IEEE 24765:2010. Systems and software engineering -- Vocabulary. 2010.

16. [ISO8601] ISO. Data elements and interchange formats - Information interchange - Representation of dates and times. 2004.

17. [JAIN2008] Babita Jain. M, Amit Jain, M.B Srinivas. A Web based Expert System Shell for Fault Diagnosis and Control of Power System Equipment. 2008. International Conference on Condition Monitoring and Diagnosis.

18. [JANZEN2005] David Janzen, Hossein Saiedian. Test-Driven Development: Concepts, Taxonomy, and Future Direction. 2005. IEEE Computer Society.

19. [KALIN2013] Martin Kalin. Java Web Services: Up and Running. O'Reilly 2013. ISBN 978-1-449-36511-0.

20. [KHADIR2009] Mohamed Tarek Khadir, Sihem Klai. A web-based fault diagnostic maintenance system for steam turbines based on domain ontology and expert system. 2009. 2009 International Conference on Multimedia Computing and Systems.

21. [MESTIZO2008] Sonia Lilia Mestizo Gutiérrez, Alejandro Guerra Hernández, Ramón Parra Loera. Desarrollo de un Centro de Ayuda Inteligente mediante el uso

de Tecnologías de Internet. Maestría en Inteligencia Artificial 2008. Veracruz, México.

22. [NAKAI2004] Kenta Nakai, Dr. Minoru Kanehisa. Expert system for predicting protein localization sites in gram-negative bacteria. 2004. Proteins: Structure, Function, and Bioinformatics.

23. [PRESSMAN2010]: Roger Pressman. Ingeniería del Software: Un Enfoque Práctico. 2010. McGraw Hill. ISBN: 978-607-15-0314-5. Cap: 8, PP: 193.

24. [SANMARTIN2013] Miguel San Martín, Edward Wong, Steven Lee. The Development of the MSL Guidance, Navigation, and Control System for Entry, Descent, and Landing. 2013. 36th Annual AAS Guidance and Control Conference, Breckenridge, Colorado, February 1-6, 2013.

25. [SHATNAWI2010] Raed Shatnawi. A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems. IEEE

26. [SHORE2008] James Shore, Shane Warden. The Art of Agile Development. 2008. O'Reilly Media, Inc.. ISBN: 978-0-596-52767-9.

27. [SNELL2001] James Snell, Doug Tidwell , Pavel Kulchenko . Programming Web Services with SOAP . O'Reilly 2001. ISBN 0-596-00095-2.

28. [SQLITE] SQLite. https://www.sqlite.org/

29. [STEVENS1974]: W. P. Stevens, G. J. Myers y L. L. Constantine. Structured Design. 1974. IBM Systems Journal, Vol: 13, No: 2, PP: 115-139, 1974, DOI: 10.1147/sj.132.0115

30. [SUN1997] Java Code Conventions. Sun Microsystems 1997.

31. [TAHCHIEV2010] Petar Tahchiev, Felipe Leme, Vincent Massol, Gary Gregory. JUnit in Action Second Edition. 2010. Manning. ISBN 9781935182023.

32. [TRAPPEY2009] Charles V. Trappey, C. S. Liu, Amy J. C. Trappey. Develop Patient Monitoring and Support System Using Mobile Communication and Intelligent

Reasoning. 2009. Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics.

33. [VELICER1999] Wayne F. Velicer, James O. Prochaska, Jeffrey M. Bellis. An expert system intervention for smoking cessation. 1999. US National Library of Medicine.

34. [W3CHTTP11] W3C. Hypertext Transfer Protocol - HTTP/1.1. 10 Status Code Definitions.

35. [W3CSOAP] SOAP Specifications. W3C.

36. [W3CWSDL] Web Services Description Language (WSDL) 1.1 Specifications. W3C.

37. [W3SCHOOLSWS] Web Services Tutorial. W3Schools.

38. [WELLS2009] Don Wells. Extreme Programming. 2009.

39. [ZHANG2000] Wenzhu Zhang, Brian T. Chait. ProFound: An Expert System for Protein Identification Using Mass Spectrometric Peptide Mapping Information. 2000. American Chemical Society. Transactions on Software Engineering, vol.36, no. 2, pp. 216-225, March/April 2010, doi:10.1109/TSE.2010.9.

Anexo A - El Lenguaje Jess

Utilizando una sintaxis similar a la de Lisp, Jess define un lenguaje mediante el cual es posible no sólo definir reglas de inferencia, sino también la creación de programas funcionales con cierto grado de complejidad. A esto se le suma el acceso a las librerías de Java, con lo cual las posibilidades para el desarrollo con Jess se vuelven infinitas. Los componentes elementales del lenguaje Jess son: los *símbolos*, los *números*, las *cadenas* y los *comentarios*.

Los **símbolos** son similares a los identificadores en el lenguaje Java, pero admiten un conjunto más amplio de caracteres. Junto con los alfanuméricos, un símbolo puede contener estos caracteres: `A-Z a-z 0-9 $ * . = + / < > _ ? #`.

Sin embargo, los símbolos poseen algunas restricciones. No puede comenzar con un número, o con `$`, `?` o `=`, y son sensibles a mayúsculas y minúsculas. Existen también símbolos con significados especiales. El símbolo `nil` es análogo a `null` en Java, y los valores `TRUE` y `FALSE` son los valores booleanos. En los ejemplos que aparecen a continuación se puede ver el símbolo `crlf`; este representa un salto de línea dentro de la función `printout`.

```
C3PO foo-fighters _bar numero#13 e=m*c2 quien?
```

Los **números** pueden ser de tres tipos: `INTEGER`, `FLOAT` o `LONG`. El primero se corresponde con el tipo `int` de Java, el segundo al `double`, y el último al `long`.

Las **cadenas** de caracteres están delimitadas por comillas dobles. Dentro de una cadena es posible utilizar la barra invertida `\` para escapar las comillas dobles y nuevas líneas (`\n`). Estas cadenas son del tipo `STRING` en Jess.

```
"Madness!" "¡Hola Mundo!" "- ¿Me dice la hora por favor? \n — 
La hora."
```

Los **comentarios** para describir y documentar el código fuente comienzan con el punto y coma, y se extienden hasta el final de la línea. Pueden aparecer en cualquier lugar del programa ya que son simplemente ignorados.

```
; El número PI. No toda la secuencia irracional, por supuesto.
; Supondremos que con este grado de precisión vamos a andar bien.
(bind ?pi 3.14159265359)
```

Los espacios en blanco no poseen significado alguno fuera de los límites de las comillas dobles. Se pueden utilizar para identar el código y hacerlo más legible a voluntad.

Listas y Funciones

La unidad estructural básica de Jess es la lista. Una lista es un conjunto de elementos delimitados por paréntesis, los cuales pueden ser algunos de los descriptos anteriormente, u otra lista. Una lista sirve para estructurar tanto datos como código. El primer elemento de una lista se llama cabecera, y en la mayoría de los casos tiene un significado especial.

En el lenguaje Jess, todo código es una invocación a una función; y toda función es una lista en donde la cabecera es un símbolo que indica el nombre de la función, y los restantes elementos son los parámetros de la misma. En el extracto siguiente pueden verse algunos ejemplos de llamadas a funciones utilizando la consola de Jess.

```
Jess> (* 2 2)
4
Jess> (bind ?x 256)
256
Jess> ; Un poco de complejidad
(= (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 2)))))))) (** 2 8))
TRUE
Jess> ; y tal vez de simplificación...
(= (* 2 2 2 2 2 2 2 2) 256)
TRUE
Jess> ; Es posible incluso jugar con el espacio
; en blanco para la identación del código.
(bind ?resultado (=
    (* 81 9)
    (** 9 3)))
```

```
TRUE
```

Algo que también puede notarse es el hecho de que todo operador (`*`, `+` , `-`, etc.) es en realidad una función. En Jess no existe el concepto de operador como se lo conoce en otros lenguajes. Además de las funciones nativas del lenguaje, en Jess es posible utilizar la API de Java para incorporar librerías, o bien definir nuevos procedimientos utilizando `deffunction`.

Variables

Una variable es un contenedor que puede mantener un valor único. Las variables en Jess tienen la particularidad de no ser tipadas y no necesitan ser declaradas antes de su uso. El identificador de una variable es un símbolo que comienza con el signo de interrogación (`?`). Como buena práctica, se recomienda evitar el uso de asteriscos (`*`), y utilizar sólo guiones medios (`-`) o bajos (`_`) para separar palabras en estos identificadores.

Los asteriscos están reservados para las variables globales. Una variable global es un tipo especial de variable que se define utilizando la función `defglobal`, y su identificador debe comenzar y finalizar con un asterisco. Al ser creada, una variable global es inicializada mediante un valor determinado. Lo que las diferencia de las variables comunes es que las globales no son eliminadas cuando se invoca la función `reset`, sino que se les reasigna el valor con el que fueron definidas.

```
Jess> (defglobal ?*y* = 256) ; Se asigna 256 en la variable
global ?*y*...
TRUE
Jess> (bind ?x 1024) ; y 1024 en ?x.
1024
Jess> ?xx
1024
Jess> ?*y*
256
Jess> (bind ?*y* 128) ; Se cambia el valor de ?*y*.
128
Jess> ?*y*
128
```

```
Jess> (reset)
TRUE
Jess> ?*y* ; y ?*y* vuelve a su valor original (256) y ?x ha
sido borrada.
256
```

En el fragmento de código que aparece a arriba se pueden ver ejemplos de asignaciones a variables globales y ordinarias. La función `bind` se utiliza para asignar valores. Los ejemplos que se han mostrado hasta entonces han sido realizados ingresando el código directamente en la consola. Esto no resulta para nada conveniente si se tiene un programa medianamente largo para ser ejecutado y se lo tiene que reingresar cada vez que se sale de la consola de Jess. La función `batch` soluciona este problema permitiendo ejecutar un archivo con código fuente en lenguaje Jess sobre el contexto actual.

Dentro de las funciones disponibles por defecto, también existe un conjunto de funciones especiales que permiten crear y recuperar información de las listas como valores. Estas listas se conocen como listas planas. Las funciones para manejar estas listas, por convención, terminan con `$`.

Estructuras de Control

El flujo de los algoritmos creados con Jess se controla mediante funciones de control de flujos especiales como `if`, `for`, `foreach`, `while`, `break`, o `continue`. Esta funciones se utilizan de manera análoga a sus contrapartes en Java.

La función `if` recibe como primer parámetro una expresión booleana. Si el resultado de esta expresión es `TRUE`, se ejecutan las funciones que aparecen justo después del símbolo `then`. Si es `FALSE`, se ejecutan las funciones luego de `else`, si es que hubieran algunos.

```
Jess> (bind ?n (random)); Se asigna un número aleatorio a ?n.
61597
Jess> (if (> ?n 10000) then
    ; Si es mayor que 10000...
    (printout t ?n " es mayor que 10000." crlf)
```

```
    else
    ; sino, necesariamente debe ser menor o igual.
    (printout t ?n " es menor o igual que 10000." crlf))
61597 es mayor que 10000.
```

Para las iteraciones pueden ser utilizadas `for`, `foreach` o `while`. En el fragmento siguiente se puede ver un ejemplo del uso de esta última.

```
Jess> (bind ?n (mod (random) 10))
4
Jess> (bind ?i 0); Índice para la iteración..
0
Jess> (while (< ?i ?n) do
    (printout t "Imprimiendo " ?i " de " ?n crlf)
    (++ ?i))
Imprimiendo 0 de 4
Imprimiendo 1 de 4
Imprimiendo 2 de 4
Imprimiendo 3 de 4
FALSE
```

La función `foreach` se utiliza, por ejemplo, para recorrer listas y realizar acciones sobre cada elemento de estas; y `for` trabaja de manera análoga a Java. La iteraciones pueden ser controladas mediante las funciones `break` y `continue`. La primera obliga a una ciclo a terminar inmediatamente, mientras que `continue` obliga a la iteración a terminar con el elemento actual y comenzar a trabajar con el siguiente. Un ejemplo puede ayudar a aclarar su funcionalidad.

```
Jess> (bind ?lista (create$ uno dos tres cuatro cinco seis)) ;
Se crea una lista...
(uno dos tres cuatro cinco seis)
Jess> (foreach ?item ?lista ; Para cada elemento en la
lista...
    (printout t "Estamos en " ?item "." crlf) ; se lo imprime.
    (if (= ?item cuatro) then ; Pero si se ha llegado al
cuarto...
        (printout t "¡Guarda! Hemos llegado al cuarto. De aquí
no pasamos." crlf)
        (break))) ; se detiene el ciclo con break.
Estamos en uno.
Estamos en dos.
```

```
Estamos en tres.
Estamos en cuatro.
¡Guarda! Hemos llegado al cuarto. De aquí no pasamos.
Jess> ; Y ahora un ejemplo con for y continue...
(for (bind ?i 1) (<= ?i ?n) (++ ?i)
    (if (= ?i 3) then
        (printout t "Mmmmm noup... No podemos imprimir el
tres." crlf)
        ; continue le indica que termine con el elemento
actual y
        ; siga con el siguiente.
        (continue))
    (printout t ?i / ?n crlf))
1/5
2/5
Mmmmm noup... No podemos imprimir el tres.
4/5
5/5
FALSE
```

Nuevas Funciones

La función `deffunction` permite crear funciones nuevas que pueden luego ser usadas como cualquier otra de Jess. El nombre de la nueva función debe ser un símbolo, y cada parámetro especificado una variable (debe comenzar con el signo de interrogación `?`). Es posible definir un comentario como una cadena de caracteres, el cual documentará el propósito de la función. Dentro del cuerpo de la función se pueden definir cualquier número de expresiones, y se utiliza `return` para devolver el resultado.

```
Jess> (deffunction saludar (?nombre)
    "Saluda a la persona."
    (printout t "¡¡Hola " ?nombre "!!" crlf))
TRUE
Jess> (saludar "Yoda")
¡¡Hola Yoda!!
```

Los Hechos

Los hechos son los datos que las reglas de inferencia utilizan para el procesamiento y realización de recomendaciones. El conjunto o colección de hechos se conoce como memoria de trabajo, y Jess cuenta con ciertas funciones que permiten

agregar, quitar y recuperarlos. Internamente, estos hechos son almacenados como listas, en donde la cabecera es el nombre del hecho.

```
(humedad 0.89)
(verde 0 255 0)
(nikola tesla)
```

Utilizando la función `facts` se puede recuperar una lista con todos los hechos presentes en la memoria de trabajo. En un principio, esta no posee hecho alguno, sin embargo al llamar a la función `reset` se crear un hecho inicial por defecto (hecho `0`). Este hecho es muy importante para poder trabajar con reglas que poseen ciertos antecedentes, por lo que se recomienda siempre invocar a esta función antes de cualquier ejecución.

```
Jess> (facts) ; Se lista los hechos. Inicialmente, no hay
hechos.
For a total of 0 facts in module MAIN.
Jess> (reset) ; Pero después de invocar a reset...
TRUE
Jess> (facts) ; aparece el hecho 0 en la memoria de trabajo.
f-0   (MAIN::initial-fact)
For a total of 1 facts in module MAIN.
```

Un punto que resulta conveniente notar es la presencia del prefijo `MAIN::` antepuesto al nombre del hecho en el listado. Este prefijo indica el módulo en donde se encuentra definido este hecho. `MAIN` es el módulo por defecto, y Jess da a posibilidad de definir otros para tener un mejor control de la ejecución del sistema. En los capítulos posteriores se ampliará un poco más este concepto, aunque no en demasiada profundidad debido a que no es una característica utilizada pare el desarrollo del Jesslet.

Existe otra función que permite realizar un seguimiento de los hechos que van apareciendo y desapareciendo de la memoria de trabajo. La función `watch` devuelve mensajes para el usuario reportando diferentes eventos dependiendo de los parámetros que recibe. Si es invocada `(watch facts)`, entonces se presentarán mensajes cada vez que un hecho se agregado o eliminado de la memoria de trabajo. Modificando un poco el ejemplo anterior, podríamos ver lo que ocurre con un `reset`.

```
Jess> (watch facts) ; Ahora Jess informará cada vez que suceda
algo con los hechos
TRUE
Jess> (reset)
 ==> f-0 (MAIN::initial-fact)
TRUE
```

Agregando Hechos

La creación de nuevos hechos se consigue a través de `assert`. Esta función recibe como parámetros cualquier cantidad de hechos, y devuelve el ID del último hecho agregado. En el caso del hecho inicial, este ID es `0` (cero). Los identificadores pueden ser luego utilizados para modificar o eliminar los hechos correspondientes.

```
Jess> (watch facts)
TRUE
Jess> (reset)
 ==> f-0 (MAIN::initial-fact)
TRUE
Jess> (assert (codigo 3301))
 ==> f-1 (MAIN::codigo 3301)
<Fact-1>
Jess> (assert (lenguajes Java Clojure Jess PHP Javascript
Python Perl))
 ==> f-2 (MAIN::lenguajes Java Clojure Jess PHP Javascript
Python Perl)
<Fact-2>
Jess> (facts)
f-0    (MAIN::initial-fact)
f-1    (MAIN::codigo 3301)
f-2    (MAIN::lenguajes Java Clojure Jess PHP Javascript Python
Perl)
For a total of 3 facts in module MAIN.
```

Si ha ocurrido un error en la inserción, entonces la función devuelve `FALSE`. Generalmente, este error indica que el hecho es un duplicado, ya que Jess sólo puede almacenar hechos únicos en la memoria de trabajo.

Eliminando Hechos

Para eliminar hechos individuales se utiliza la función `retract`. Esta función recibe como parámetros los identificadores de los hechos que se quieren remover, o bien los hechos en sí. Siguiendo con los hechos que se tenían desde el ejemplo anterior.

```
Jess> (facts)
f-0    (MAIN::initial-fact)
f-1    (MAIN::codigo 3301)
f-2    (MAIN::lenguajes Java Clojure Jess PHP Javascript Python
Perl)
For a total of 3 facts in module MAIN.
Jess> (retract 1) ; Se quita el hecho con ID 1.
 <== f-1 (MAIN::codigo 3301)
TRUE
Jess> (bind ?h (fact-id 2)) ; Se usa la función fact-id para
recuperar el hecho.
<Fact-2>
Jess> ?h
<Fact-2>
Jess> (retract ?h) ; Se quita el hecho en la variable.
 <== f-2 (MAIN::lenguajes Java Clojure Jess PHP Javascript
Python Perl)
TRUE
Jess> (facts)
f-0    (MAIN::initial-fact)
For a total of 1 facts in module MAIN.
```

El resultado de utilizar `retract` con un identificador o con una instancia de un hecho es exactamente el mismo, pero este último resulta más eficiente y rápido que utilizando el identificador del hecho.

Cuando se quiere reiniciar la memoria de trabajo, Jess las funciones `clear` y `reset` que pueden ser de utilidad. Ambas eliminan los hechos de la memoria de trabajo, pero `clear` va un poco más allá e incluye entre sus objetivos a las reglas de inferencia, variables y funciones definidas por el usuario. Para reiniciar el estado de un sistema sin borrarlo, es necesario utilizar `reset`. Esta función elimina todos los hechos y deja solamente el hecho `0` en memoria.

```
Jess> (assert (lenguajes Java Jess C++ C# Python Clojure))
```

```
<Fact-1>
Jess> (assert (foo bar baz))
<Fact-2>
Jess> (facts) ; Se recuperan los hechos en la memoria de
trabajo.
f-0    (MAIN::initial-fact)
f-1    (MAIN::lenguajes Java Jess C++ C# Python Clojure)
f-2    (MAIN::foo bar baz)
For a total of 3 facts in module MAIN.
Jess> (reset) ; Y se aplica un reset.
TRUE
Jess> (facts) ; Ahora sólo aparece el hecho inicial.
f-0    (MAIN::initial-fact)
For a total of 1 facts in module MAIN.
```

Hechos Por Defecto

Utilizando la función `deffacts` es posible definir un conjunto de hechos iniciales que aparecerán cada vez que se invoque a la función `reset`. Con esto se consigue llevar al sistema a un estado inicial conocido sin la necesidad de insertar uno a uno los hechos mediante un `assert` cada vez que se restablezca la memoria de trabajo.

La función `deffacts` recibe como parámetros el nombre de la lista, un comentario opcional a modo de documentación, y el listado de hechos iniciales.

```
Jess> (deffacts memes "Cantidad de memes"
    (yao-ming 191)
    (troll-face 8098)
    (poker-face 789)
    (forever-alone 437)
    (cereal-guy 90))
TRUE
Jess> (reset)
TRUE
Jess> (facts)
f-0    (MAIN::initial-fact)
f-1    (MAIN::yao-ming 191)
f-2    (MAIN::troll-face 8098)
f-3    (MAIN::poker-face 789)
f-4    (MAIN::forever-alone 437)
f-5    (MAIN::cereal-guy 90)
For a total of 6 facts in module MAIN.
```

Hechos No Ordenados

Los hechos no ordenados son hechos que poseen campos análogos a una tabla en una base de datos. Sin embargo antes de poder crear un hecho de este tipo, primeramente hay que definir su estructura. Para ello se utiliza la función `deftemplate`. Esta recibe como parámetros el nombre de la estructura, junto con los campos de la misma establecidos mediante `slot`.

```
Jess> (deftemplate libro "Un libro"
    (slot titulo)
    (slot autor)
    (slot isbn))
TRUE
```

Cuando se crean hechos no ordenados, los valores de los campos pueden ser especificados en cualquier orden.

```
Jess> (assert (libro (autor "Julio Cortázar") (isbn
9789875780620)
    (titulo Rayuela)))
<Fact-0>
Jess> (facts)
f-0    (MAIN::libro (titulo Rayuela) (autor "Julio Cortázar")
(isbn 9789875780620))
For a total of 1 facts in module MAIN.
```

Si el valor de un campo no es especificado, entonces se la asigna el valor por defecto. Los valores por defecto para cada campo pueden ser especificados al definir cada campo.

```
Jess> (clear)
TRUE
Jess> (deftemplate libro ""
    (slot titulo)
    (slot autor (default Anónimo))
    (slot isbn))
TRUE
Jess> (assert (libro (titulo "Sangre Estival")
    (isbn 9789876831888)))
<Fact-0>
Jess> (facts)
```

```
f-0    (MAIN::libro (titulo "Sangre Estival") (autor Anónimo)
(isbn 9789876831888))
For a total of 1 facts in module MAIN.
```

Los valores contenidos en los hechos no ordenados no son constantes y pueden ser alterados. Para este fin se hace uso de la función `modify`.

```
Jess> (bind ?h (fact-id 0))
<Fact-0>
Jess> (modify ?h (autor "Virginio Tamaro"))
<Fact-0>
Jess> (facts)
f-0    (MAIN::libro (titulo "Sangre Estival") (autor "Virginio
Tamaro") (isbn 9789876831888))
For a total of 1 facts in module MAIN.
```

Esta función recibe como primer parámetro el hecho a modificarse, o su ID. Lo que sigue es un conjunto de pares ordenados de la forma `(<campo>, <nuevo valor>)`.

Hechos Ordenados

Los hechos no ordenados resultan de gran utilidad cuando se requiere organizar los datos dentro en estructuras, pero cuando lo que se necesita es almacenar un único dato, entonces la definición de una estructura para este caso resulta innecesaria y excesiva. Como ya ha sido presentado en alguno de los ejemplos anteriores, Jess posibilita la creación de hechos de la siguiente manera:

```
Jess> (assert (river 35))
<Fact-0>
Jess> (facts)
f-0    (MAIN::river 35)
For a total of 1 facts in module MAIN.
Es posible incluso definir una lista de elementos como valor
de un hecho.
Jess> (assert (proporciones 0.5 0.3 0.2))
<Fact-1>
Jess> (facts)
f-0    (MAIN::river 35)
f-1    (MAIN::proporciones 0.5 0.3 0.2)
```

```
For a total of 2 facts in module MAIN.
```

Todas estas acciones se realizan sin la necesidad de definir una estructura mediante `deftemplate`, siempre que no haya una estructura con el mismo nombre previamente especificada. Esto se debe a que cuando un hecho ordenado es creado por primera vez, Jess crea automáticamente su estructura con el nombre del hecho. Con la función `showdeftemplates` se pueden recuperar todas las estructuras de hechos definidas, y con `ppdeftemplate` se puede obtener un detalle de una estructura determinada.

```
Jess> (show-deftemplates)

(deftemplate MAIN::__clear
   "(Implied)")

(deftemplate MAIN::__fact
   "Parent template")

(deftemplate MAIN::__not_or_test_CE
   "(Implied)")

(deftemplate MAIN::initial-fact
   "(Implied)")

(deftemplate MAIN::proporciones
   "(Implied)"
   (multislot __data))

(deftemplate MAIN::river
   "(Implied)"
   (multislot __data))
FALSE
Jess> (ppdeftemplate river)
"(deftemplate MAIN::river
   \"(Implied)\"
   (multislot __data))"
```

La estructura generada para el hecho posee un único campo multiple `__data`. Este campo se encuentra normalmente oculto cuando los hecho son presentados. En sí, los hechos ordenados son un subconjuto de los hecho no ordenados con un único campo llamado `__data`.

Se recomienda utilizar hechos ordenados cuando se requiera tener información volátil y con poca trascendencia a lo largo del programa. Los hechos no ordenados son preferibles debido a que poseen mayor flexibilidad, seguridad con respecto a los datos que mantienen, tienden a producir menos errores de programación, y tienen un mejor desempeño en términos computacionales.

Las Reglas

Las reglas componen lo que se conoce como base de conocimientos, y estas pueden llevar a cabo acciones basándose en los hechos que se encuentran en la memoria de trabajo. Existen dos tipos de reglas de inferencia: las de **encadenamiento hacia adelante** y las de **encadenamiento hacia atrás**.

En el encadenamiento hacia adelante tiene una forma análoga a la estructura `if ... then` presente en varios lenguajes de programación, en donde la parte `then` (**consecuente**) es ejecutada siempre que se cumplan las condiciones en el `if` (**antecedente**). Las reglas de encadenamiento hacia atrás también poseen esta estructura, con la diferencia de que en estas se trata de satisfacer de manera activa las premisas en la parte de `if`. Para este proyecto sólo serán utilizadas reglas de encadenamiento hacia adelante, por lo que solamente estas serán analizadas en profundidad.

Creación de una Regla

Toda regla en Jess es definida utilizando la función `defrule`. En su versión más simple, es posible escribir una regla sin condiciones y que no realice acciones algunas.

```
Jess> (defrule regla-vacia
    "Esta regla no realiza acción alguna"
    =>
    )
TRUE
Jess> (rules)
MAIN::regla-vacia
For a total of 1 rules in module MAIN.
```

El primer parámetro que recibe `defrule` es el nombre de la regla (`regla-vacia` en el ejemplo), y este debe ser un símbolo. El segundo es una cadena opcional de documentación que describe el propósito de la regla. El símbolo `=>` separa las condiciones (`if`) de las acciones (`then`). La condiciones son un conjunto de patrones que serán comparados con los hechos en la base de hechos, mientras que las acciones son llamadas a funciones que pueden modificar estos hechos, las reglas de inferencia, o enviar mensajes para el usuario.

Paralelamente a lo que sucede con los hechos, la función `watch` también puede ser utilizada para mostrar mensaje cada vez que ocurra algo de interés con las reglas. Utilizando `(watch activations)` se le indica a Jess que muestre un mensaje un registro de activación sea agregado o eliminado. Un registro de activación asocia un conjunto de hechos con una regla, e implica que dicho conjunto cumple las condiciones de la regla por lo que esta debe ser disparada.

Siguiendo el ejemplo anterior:

```
Jess> (watch rules)
TRUE
Jess> (watch activations)
TRUE
Jess> (watch facts)
TRUE
Jess> (reset)
 ==> f-0 (MAIN::initial-fact)
==> Activation: MAIN::regla-vacia :  f-0
TRUE
Jess> (run)
FIRE 1 MAIN::regla-vacia f-0
1
```

Como se puede ver en el ejemplo, la regla se activa luego de que aparece en la memoria de trabajo el hecho inicial creado por `reset`. Este ejemplo pone de manifiesto una de las funcionalidades que posee este hecho inicial: **activar reglas que poseen un conjunto vacío de condiciones**.

La función `rules` (en el primer ejemplo) presenta un listado de las reglas que se han definido, y se puede utilizar `ppdefrule` para presentar el detalle interno de una de estas.

Mediante `(watch rules)` se le indica a Jess que presente un mensaje cuando una regla sea disparada. Se dice que una regla es disparada cuando se ejecutan las acciones en la parte del `then`. La función `run` hace que las reglas activas comiencen a dispararse, y continúa una a una hasta que no existan reglas activas. Ninguna regla puede ser disparada antes de invocar a esta función.

La siguiente regla es un ejemplo más complejo en donde aparecen un antecedente junto con ciertas acciones en el consecuente.

```
Jess> (defrule cargar-bateria-baja
    "Si la batería del celular está baja, recargarla."
    ?bat <- (bateria-baja)
    =>
    (printout t "Cargando batería..." crlf)
    (retract ?bat) )
TRUE
```

En este punto es conveniente remarcar nuevamente que **todo lo que aparece en el antecedente no son funciones**. Se trata de patrones que serán aplicados sobre los hechos. Debido a la forma en la que trabajan los lenguajes de programación y su analogía con la estructura `if ... then`, se tiende a pensar en expresiones y condiciones booleanas. Jess no intenta evaluar a `TRUE` o `FALSE` el antecedente, sino que trabaja con los patrones para obtener el conjunto de hechos que satisfaga las condiciones y de lugar a la creación de un registro de activación.

En el siguiente ejemplo se utiliza `(watch all)` para conseguir que Jess presente un mensaje para cada evento de interés en las reglas, hechos y activaciones.

```
Jess> (watch all)
TRUE
Jess> (defrule cargar-bateria-baja
    "Si la batería del celular está baja, recargarla."
    ?bat <- (bateria-baja)
```

```
    =>
    (printout t "Cargando batería..." crlf)
    (retract ?bat)
    )
MAIN::cargar-bateria-baja: +1+1+1+t
TRUE
Jess> (reset)
 ==> Focus MAIN
 ==> f-0 (MAIN::initial-fact)
TRUE
Jess> (assert (bateria-baja))
 ==> f-1 (MAIN::bateria-baja)
==> Activation: MAIN::cargar-bateria-baja :  f-1
<Fact-1>
Jess> (run)
FIRE 1 MAIN::cargar-bateria-baja f-1
Cargando batería...
 <== f-1 (MAIN::bateria-baja)
 <== Focus MAIN
1
```

La regla es activada cuando se crea el hecho `(bateria-baja)` y se dispara una vez que se invoca a la función `run`. Lo que ocurre a continuación está dado por las acciones en el consecuente. En primer lugar se presenta el mensaje para el usuario, y a continuación se elimina el hecho de la memoria de trabajo. Es de esta manera como las reglas pueden modificar la memoria de trabajo: agregando o quitando hechos en sus acciones.

El número `1` que aparece al final de la ejecución es el número total de reglas que han sido disparadas.

Restricciones y Filtros de los Hechos

La mayor parte de los hechos que aparecen en los sistemas expertos no se tratan de hechos vacíos (como `(barteria-cargada)`), sino que poseen datos adicionales. Y los patrones en las condiciones de las reglas pueden utilizar estos datos como un filtro para especificar un subconjunto de hechos. Estos filtros se conocen como restricciones, puesto que restringen los valores que puede contener un hecho en un patrón, y pueden pertenecer a alguno de los siguientes tipos:

1. Literales
2. De variable
3. Conectivas
4. De función predicado
5. De retorno de función

Debido a que para el desarrollo de este trabajo sólo se dará soporte a los hechos ordenados, los desarrollos conceptuales y ejemplificaciones estarán orientados exclusivamente hacia este tipo de hechos.

Restricciones Literales

Pueden establecerse valores literales como restricciones dentro de los patrones. De esta manera, sólo se verificarán con estos patrones los hechos que contengan esos valores en las posiciones correspondientes. En el ejemplo de abajo se especifica un patrón que se cumple cuando existe un hecho `(lenguaje Python)` en la memoria de trabajo.

```
Jess> (watch all)
TRUE
Jess> (defrule regla-literal
    (lenguaje Python)
    =>
    (printout t "Se cumple el valor literal." crlf))
MAIN::regla-literal: +1+1+1+1+t
TRUE
Jess> (reset)
 ==> Focus MAIN
 ==> f-0 (MAIN::initial-fact)
TRUE
Jess> (assert (lenguaje Java))
 ==> f-1 (MAIN::lenguaje Java)
<Fact-1>
Jess> (assert (lenguaje "Python"))
 ==> f-2 (MAIN::lenguaje "Python")
<Fact-2>
Jess> (assert (lenguaje Python))
 ==> f-3 (MAIN::lenguaje Python)
==> Activation: MAIN::regla-literal :  f-3
```

```
<Fact-3>
Jess> (facts)
f-0    (MAIN::initial-fact)
f-1    (MAIN::lenguaje Java)
f-2    (MAIN::lenguaje "Python")
f-3    (MAIN::lenguaje Python)
For a total of 4 facts in module MAIN.
```

A partir del ejemplo de arriba se debe hacer hincapié en un punto. Las restricciones literales obligan una correspondencia exacta del valor dado; no se realizan conversiones de datos. Es por eso que al crearse el hecho `(lenguaje "Python")` la regla no es activada, pero sí cuando aparece en la memoria de trabajo `(lenguaje Python)`. `"Python"` es una cadena de caracteres (`STRING`), mientras que `Python` es un símbolo (`SYMBOL`).

```
Jess> (jess-type "Python")
STRING
Jess> (jess-type Python)
SYMBOL
```

Restricciones de Variables

Es posible definir una variable en lugar de un valor literal para los valores de los hechos en los patrones. Una variable es asignada con el valor que posea el hecho en la posición en la que esta aparezca.

```
Jess> (defrule regla-variables
    "Esta regla tiene variables como restricciones."
    (hecho ?foo ?bar)
    =>
    (printout t "Las variables son " ?foo " y " ?bar "."
crlf))
```

Esta regla se activa cada vez que aparece un hecho con valores asignados en sus dos primeras posiciones, por ejemplo `(hecho 1 2)`, `(hecho uno dos)`. Estos valores se asignan a las variables `?foo` y `?bar` en sus respectivas posiciones, y estas se encuentran disponibles para ser utilizadas en las acciones presentes en el consecuente. Siguiendo el ejemplo anterior:

```
Jess> (assert (hecho uno dos))
<Fact-0>
Jess> (assert (hecho tres cuatro cinco))
<Fact-1>
Jess> (run)
Las variables son uno y dos.
1
```

Como puede verse, sólo activan a la regla los hechos que contienen sólo dos valores. Si poseen un número distinto, cualquiera sea, no se verificarán con el patrón. El valor uno se asigna a `?foo`, y dos a `?bar`.

Una variable puede aparecer más de un patrón en la misma regla, con la restricción de que para cada aparición debe tener asignado exactamente el mismo valor.

```
Jess> (defrule regla-variables
    "Otro ejemplo de variables en las restricciones."
    (hecho ?foo ?foo)
    =>
    (printout t "Tenemos un hecho con el valor " ?foo
        " en sus dos primeras posiciones." crlf))
TRUE
Jess> (reset)
TRUE
Jess> (assert (hecho c3p0 c3p0))
<Fact-1>
Jess> (assert (hecho uno dos))
<Fact-2>
Jess> (assert (hecho "" ""))
<Fact-3>
Jess> (run)
Tenemos un hecho con el valor  en sus dos primeras posiciones.
Tenemos un hecho con el valor c3p0 en sus dos primeras
posiciones.
2
```

Es posible restringir un valor en un hecho sin establecer una variable a la cual vincularla utilizando un signo de interrogación (`?`) en lugar de un identificador. En general esto es útil sólo cuando se requiere especificar que cierto hecho contiene algunos valores, sin importar cuáles sean estos.

Restricciones Conectivas

El sólo hecho de limitar los valores de los hechos a literales únicos no es suficiente para la representación del conocimiento experto en las reglas de inferencia. Las restricciones conectivas pueden percibirse como una mejora sobre los patrones de las literales que se presentaron anteriormente, en donde pueden utilizarse los conectores `&` (conjunción), `|` (disyunción), y `~` (negación).

Una restricción que sea precedida por el tilde `~` concordará con opuesto a lo que esta restricción originalmente lo hubiese hecho.

```
Jess> (defrule regla-conectiva
    "Esta regla sólo se activa si hay algún hecho que no
    contenga el valor Python en su primera posición."
    (lenguaje ~Python)
    =>
    (printout t "Tenemos uno que no es Python." crlf))
TRUE
Jess> (assert (lenguaje Java))
<Fact-4>
Jess> (reset)
TRUE
Jess> (assert (lenguaje Java))
<Fact-1>
Jess> (assert (lenguaje Python))
<Fact-2>
Jess> (assert (lenguaje "Python"))
<Fact-3>
Jess> (run)
Tenemos uno que no es Python.
Tenemos uno que no es Python.
2
```

Es posible utilizar la conjunción y la disyunción para armar restricciones simples.

```
Jess> (defrule regla-conectiva
    (protocolo http|ftp)
    =>
    (printout t "Trabajando sobre HTTP o FTP." crlf))
TRUE
Jess> (reset)
TRUE
Jess> (assert (protocolo ftp))
```

```
<Fact-1>
Jess> (run)
Trabajando sobre HTTP o FTP.
1
```

También es posible vincular una variable mediante el conector de la conjunción.

```
Jess> (defrule regla-conectiva
    (provincia ?p&~Corrientes)
    (ciudad "Santos Lugares")
    =>
    (printout t "Tenemos una ciudad llamada Santos Lugares "
    "en una provincia " ?p crlf))
TRUE
Jess> (reset)
TRUE
Jess> (assert (ciudad "Santos Lugares"))
<Fact-1>
Jess> (assert (provincia Formosa))
<Fact-2>
Jess> (run)
Tenemos una ciudad llamada Santos Lugares en una provincia
Formosa
1
```

Se debe notar que el orden de precedencia para estos patrones es la negación, seguido de la conjunción y la disyunción, y no es posible la utilización de paréntesis para modificar esto.

Restricciones de Función Predicado

A pesar de la aparente flexibilidad que brindan las restricciones literales y conectivas, su alcance puede verse limitado al intentar obtener patrones más complejos en donde deban realizarse llamadas a funciones o cálculos sobre los valores. Por ejemplo, supóngase que se tienen en la memoria de trabajo hechos que contienen los datos personales de una persona. Entre estos datos se encuentra el de su fecha de nacimiento. Si en alguna regla se requiere que el antecedente trabaje con la edad de la persona, este cálculo sería imposible de realizar utilizando solamente las restricciones hasta aquí vistas. Para estos casos, Jess dispone de las restricciones de función predicado, mediante las cuales puede expresarse casi cualquier restricción que pueda ser necesaria.

Una función predicado es una función que devuelve un valor booleano (`TRUE` o `FALSE`), y puede ser establecida como restricción en los patrones mediante los dos puntos (`:`). Si se quiere utilizar el valor del campo en cuestión, debe ser asignado con una variable.

```
Jess> (defrule regla
    "Esta regla sólo será activada si el nombre de
    la persona tiene más de cuatro caracteres."
    (nombre ?n&:(> (str-length ?n) 4))
    =>
    (printout t ?n " tiene más de 4 caracteres." crlf))
TRUE
Jess> (reset)
TRUE
Jess> (assert (nombre "Dominic"))
<Fact-1>
Jess> (assert (nombre "Harold"))
<Fact-2>
Jess> (assert (nombre "Elias"))
<Fact-3>
Jess> (assert (nombre "Root"))
<Fact-4>
Jess> (run)
Dominic tiene más de 4 caracteres.
Harold tiene más de 4 caracteres.
Elias tiene más de 4 caracteres.
```

```
2
```

La potencia de las funciones de predicado radica en el hecho de que pueden ser agrupadas en estructuras más complejas mediante conjunciones (`and`) y disyunciones (`or`).

```
Jess> (defrule mayores-solamente
    "Esta regla sólo permite el paso de personas
    entre los 35 y los 50 años."
    (edad ?e&:(and
        (>= ?e 35)
        (<= ?e 50)))
    =>
    (printout t ?e " años." crlf))
TRUE
Jess> (assert (edad 25))
<Fact-0>
Jess> (assert (edad 49))
<Fact-1>
Jess> (run)
49 años.
1
```

Restricciones de Retorno de Función

En ciertas situaciones es necesario que el valor en un hecho concuerde con el valor devuelto por una función. Supóngase que, en una condición de una regla, se necesitan dos edades de tal manera que una sea exactamente tres veces la otra. De repente, con lo visto hasta aquí, se podría plantear algo como lo siguiente:

```
Jess> (defrule triple-edad
    "Esta regla se dispara si hay dos edades tal que
    una sea tres veces la otra."
    (edad ?e1)
    (edad ?e2&:(=
        ?e2 (* ?e1 3)))
    =>
    (printout t ?e2 " = " ?e1 "x3." crlf))
TRUE
```

Si se agregan los hechos necesarios y se ejecuta, se tiene lo siguiente.

```
Jess> (assert (edad 21))
<Fact-0>
Jess> (assert (edad 7))
<Fact-1>
Jess> (run)
21 = 7x3.
1
```

Cuando se quiere restringir un valor a lo retornado por una función, esta es precedida por un signo igual (=). El condicional de la regla puede ser reescrito de la siguiente manera.

```
Jess> (defrule triple-edad
    "Esta regla se dispara si hay dos edades tal que
    una sea tres veces la otra."
    (edad ?e1)
    (edad =(* ?e1 3))
        =>
    (printout t (* ?e1 3) " = " ?e1 "x3." crlf))
TRUE
```

Lo que consigue de esta manera es simplificar los patrones al no escribir la función `eq`. El ejemplo también tiene la particularidad de que la variable `?y` no es utilizada en su restricción, por lo que la aplicación de esta restricción resulta idónea. En sí, cuando Jess es ejecutado, esta nueva forma se convierte hacia la función `eq`, por lo que no hay diferencia en cuanto al rendimiento de una u otra forma.

Asignación de Patrones

Para poder ser utilizados en las acciones del consecuente, los hechos que concuerdan en el antecedente deben ser asignados a variables. Para esto se utiliza el operador `<-` como aparece en el siguiente fragmento.

```
Jess> (defrule regla
    ?hecho <- (password 123456)
    =>
    (printout t "Hay una contraseña demasiado sencilla." crlf)
    (retract ?hecho))
TRUE
Jess> (reset)
```

```
TRUE
Jess> (watch facts)
TRUE
Jess> (assert (password admin))
 ==> f-1 (MAIN::password admin)
<Fact-1>
Jess> (assert (password 123456))
 ==> f-2 (MAIN::password 123456)
<Fact-2>
Jess> (run)
Hay una contraseña demasiado sencilla.
 <== f-2 (MAIN::password 123456)
1
```

Así, el hecho `(password 123456)` queda asignado en la variable `?hecho`, para luego ser quitado de la memoria mediante la función `retract`.

Los Elementos Condicionales

Así como es posible asignar variables y aplicar funciones en los patrones para la restricción de los valores en los hechos, Jess también cuenta con ciertas funcionalidades que permiten expresar relaciones más complejas entre los hechos. Se tratan de los elementos condicionales, y permiten modificar los patrones al agruparlos en estructuras lógicas.

Muchos de estos elementos condicionales poseen exactamente el mismo nombre que funciones usadas como funciones predicado, lo que puede prestarse a confusión. Por ello hay que remarcar que no se tratan de las mismas funciones. Por ejemplo, la función `and` que aparece en un predicado es una función **booleana** que devuelve `TRUE` o `FALSE`, mientras que el elemento condicional `and` se verifica si se cumple un conjunto de patrones de hechos.

Los elementos condicionales incluidos en Jess son:

- `and` – verifica varios patrones al mismo tiempo
- `or` – verifica patrones alternativos
- `not` – se verifica si no se cumple patrón alguno dentro de un conjunto

- `exists` – se verifica si al menos un patrón dentro de un conjunto se verifica
- `test` – se cumple si una función no devuelve FALSE
- `logical` - permite definir hechos como consecuencias de otros

El Elemento Condicional `and`

Una regla de inferencia contiene en su condición un conjunto de cero o más patrones. Cada patrón define un conjunto de hechos, y todos estos patrones deben cumplirse para que la regla se active. El elemento condicional `and` funciona de esta manera, y puede verse como la intersección de los conjuntos generados por estos patrones. De hecho, toda condición de una regla es envuelta dentro de un `and` tácito.

```
Jess> (defrule regla-and
    (and (edad ?e&:(>= ?e 18))
        (acreditado))
    =>
    (printout t "Está acreditato y es mayor de edad." crlf))
TRUE
```

Hay que notar que la regla del ejemplo de arriba funcionaría exactamente de la misma manera que si se hubiese omitido el `and`. El elemento condicional, en este caso, sólo fue agregado a los fines de presentación. La potencia de estos radica en las posibilidades de combinación con otros, como el `or` o el `not`.

El Elemento Condicional `or`

Este elemento se cumple si uno o más de los patrones que incluye se verifica. Si se cumple más de uno de los patrones, el elemento `or` se verificará más de una vez también.

```
(defrule regla-or
    ?f <- (or (role administrator)
        (username root))
    =>
    (printout t "El usuario está autorizado." crlf))
```

Si con esta regla aparecen los hechos `(username root)` y `(role administrator)`, entonces esta se activaría y dispararía dos veces, y la variable `?f` quedaría asignada con cualquiera de los dos hechos que provoquen la activación.

```
Jess> (reset)
TRUE
Jess> (assert (username root))
<Fact-1>
Jess> (assert (role administrator))
<Fact-2>
Jess> (run)
El usuario está autorizado.
El usuario está autorizado.
2
```

Como puede verse, con la utilización del `or` en la regla del ejemplo se obtiene un funcionamiento análogo al de tener dos reglas. Internamente, Jess convierte este elemento condicional en `n` reglas de inferencia, una por cada patrón que posee.

El Elemento Condicional `not`

Este elemento condicional se utiliza para controlar la no existencia de un conjunto de uno o más patrones dentro de la memoria de trabajo.

```
Jess> (defrule sin-alfiles-negros
    (not (alfil negro))
    =>
    (printout t "No hay alfiles negros en el tablero." crlf))
TRUE
```

Debido al funcionamiento de este elemento condicional, no se pueden definir variables para ser asignadas con el hecho. Es posible, sin embargo, utilizar variables dentro de un patrón `not`, siempre que sólo sean utilizadas dentro del mismo patrón.

El `not` es evaluado ante tres situaciones:

1. Cuando se crea un hecho que concuerda con uno de los patrones que incluye, fallando en este caso.

2. Cuando se elimina un hecho que concuerda con uno de los patrones incluidos, verificándose esta vez.
3. Cuando un patrón inmediato anterior dentro del condicional de la regla es evaluado.

Si el elemento condicional `not` está primero dentro de un grupo de patrones, entonces el patrón `(initial-fact)` es antepuesto para cumplir la función de este patrón anterior. En este punto puede notarse la importancia de la llamada a la función `reset` previo a la ejecución, debido a que esta crea el hecho inicial, garantizando el correcto funcionamiento de los patrones con `not`.

Como un ejemplo más complejo, se podría tener la siguiente regla:

```
Jess> (defrule enroque
    ?t <- (pieza torre blanco h 1)
    ?r <- (pieza rey blanco e 1)
    (not (pieza ? ? f|g 1))
    =>
    (printout t "Realizamos el enroque." crlf)
    (retract ?t)
    (retract ?r)
    (assert (pieza torre blanco f 1))
    (assert (pieza rey blanco g 1)))
TRUE
```

Aquí se establecen las condiciones que tienen que darse en un tablero de ajedrez para que sea posible realizar un enroque. Por supuesto, falta saber si tanto el rey como la torre involucrados han sido movidos o no, pero a los fines de la simplicidad, se obviará esa condición. El elemento condicional `not`, en este caso, se verificará cuando no exista ninguna pieza del tipo color que sea, en las posiciones `F1` y `G1`.

El Elemento Condicional `exists`

El elemento condicional `exists`, de manera análoga al `not`, puede abarcar uno o más patrones, y se verifica si se cumplen uno o más de estos. Este elemento condicional es útil cuando se requiere disparar una regla sólo una vez, aunque puedan existir muchos hechos en la memoria de trabajo que podrían activarla. Usar `(exists (hecho))` es lo mismo que `(not (not (hecho))`.

El Elemento Condicional `test`

Este elemento posee una particularidad que lo distingue del resto: sus parámetros no se tratan de otros patrones, sino que es una expresión o función booleana. El resultado determina si el patrón se cumple o falla. Un patrón `test` falla si y sólo si la función devuelve `FALSE`.

La evaluación del `test` es igual al `not`, por lo que, nuevamente, es importante el uso del `reset` previo a la ejecución. Una función o expresión que aparezca en el `test` puede ser igualmente escrita utilizando restricciones de predicados, con la diferencia que con la primera opción, ante una evaluación booleana demasiado larga, el `test` puede mejorar la legibilidad del código fuente.

De esta manera, la regla:

```
Jess> (defrule mayores
    (persona ?nombre ?apellido ?edad)
    (test (> ?edad 17))
    =>
    (printout t ?nombre " " ?apellido " es mayor." crlf))
TRUE
```

Equivale a la siguiente:

```
Jess> (defrule mayores
    (persona ?nombre ?apellido ?edad&:(> ?edad 17))
    =>
    (printout t ?nombre " " ?apellido " es mayor." crlf))
TRUE
```

El Elemento Condicional `logical`

Existen situaciones en donde un hecho es una consecuencia directa de otro. Por ejemplo, si se activa un interruptor de una lámpara, se espera que el foco se encienda. Y a la inversa, si el interruptor es desactivado, la lógica indica que el foco debe apagarse. Este tipo de relaciones se llaman dependencias lógicas, y una forma de representarlas en Jess es la siguiente:

```
Jess> (defrule foco-encendido-interruptor
    (interruptor 1)
    =>
    (assert (foco 1)))
TRUE
Jess> (defrule foco-apagado-interruptor
    (not (interruptor 1))
    ?f <- (foco 1)
    =>
    (retract ?f))
TRUE
```

Sin embargo, es posible utilizar el elemento `logical` para conseguir los mismos resultados de manera más concisa. El anterior ejemplo puede simplificarse de esta manera:

```
Jess> (defrule foco-interruptor
    (logical (interruptor 1))
    =>
    (assert (foco 1)))
TRUE
```

De esta manera, se obtendría el siguiente funcionamiento:

```
Jess> (watch facts)
TRUE
Jess> (reset)
 ==> f-0 (MAIN::initial-fact)
TRUE
Jess> (assert (interruptor 1))
 ==> f-1 (MAIN::interruptor 1)
<Fact-1>
Jess> (run)
 ==> f-2 (MAIN::foco 1)
```

```
1
Jess> (retract 1)
 <== f-1 (MAIN::interruptor 1)
 <== f-2 (MAIN::foco 1)
TRUE
```

Restricciones en la Representación de las Reglas

Debido a que la creación de un servicio web que brinde todas las posibilidades que provee Jess para la representación de las reglas de inferencia requeriría un desarrollo complejo, para el Jesslet se realizarán ciertas restricciones en este sentido.

En primer lugar, para las restricciones en los patrones de hechos, sólo será posible la utilización de las de función de predicado. El motivo de esto es que mediante estas restricciones es posible representar cualquiera de las restantes. Un función de predicado brinda la flexibilidad suficiente para poder trabajar con diversas expresiones y patrones, aunque en detrimento, por supuesto, de la simplicidad.

Por otro lado, sólo podrán ser utilizados los elementos condicionales `and` , `not` y `exists`. Como se mencionó arriba, el `or` puede ser traducido como varias reglas de inferencia, y los elementos `test` y `logical` pueden ser representados sin problemas mediante restricciones de función de predicado. La omisión de estos elementos condicionales tiene el fin de evitar volver innecesariamente más compleja al contrato del servicio web y al Jesslet en sí.

Anexo B - SOAP

SOAP (*Simple Object Access Protocol*) es un protocolo basado en XML e independiente de la plataforma, para la comunicación entre aplicaciones. La especificación define solamente un formato para el envío y la recepción de mensajes, junto con un conjunto de reglas para traducir tipos de datos específicos de la aplicación y la plataforma a representaciones en texto XML y viceversa [W3SCHOOLSWS, SNELL2001].

En SOAP, las aplicaciones se comunican mediante mensajes XML. Esta forma de comunicación se caracteriza por su gran flexibilidad, ya que XML es independiente de plataforma, lenguaje de programación o sistema operativo alguno.

Mensajes SOAP

Un mensaje SOAP consiste en un **sobre** (`Envelope`) que contiene una cabecera y un cuerpo. La **cabecera**, la cual es opcional, lleva información referente a cómo debe ser procesado el mensaje, los cuales pueden incluir configuraciones de direccionamiento y entrega del mensaje, u opciones de autenticación y autorización. El **cuerpo** contiene el mensaje en sí, y siempre debe aparecer en el mensaje. Cualquier información que pueda ser representada en XML debe ir en el cuerpo.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-
encoding">
    <soap:Header>
        <!-- Cabecera -->
    </soap:Header>
    <soap:Body>
        <!-- Cuerpo del mensaje -->
        <soap:Fault>
            <!-- Error en SOAP -->
        </soap:Fault>
    </soap:Body>
```

```
</soap:Envelope>
```

El elemento `Envelope` es el elemento raíz del mensaje, y define al documento XML como tal. El espacio de nombres `http://www.w3.org/2001/12/soap-envelope` define la sintaxis para armar un mensaje SOAP. En esta URL se puede encontrar la especificación de un esquema (XSD) para uno de estos documentos XML.

Mediante el atributo `encodingStyle` se definen los tipos de datos usados en el documento. Hay que notar que un mensaje SOAP no posee una codificación por defecto.

Cabecera y Cuerpo

Si el elemento `Header` está presente, entonces debe ser el primero dentro del mensaje. Su objetivo es comunicar información contextual relativa al procesamiento del mensaje SOAP. En los elementos de la cabecera pueden aparecer tres atributos: `actor`, `mustUnderstand`, y `encodingStyle`. Estos definen como el receptor deberá procesar el mensaje.

Desde que el mensaje parte desde el emisor, hasta llegar al receptor, puede pasar por diferentes intermediarios. Este conjunto de intermediarios se conoce como **camino del mensaje**, y cada uno de ellos se denomina **actor**. La construcción del mensaje no está definida en la especificación SOAP. Aunque existen ciertas extensiones que pretenden cubrir este vacío, no existe una estándar o utilizada de manera general. Sin embargo, SOAP sí especifica un mecanismo que permite identificar qué partes del mensaje están dirigidas a actores específicos. El atributo `actor` se utiliza para dirigir la cabecera de un mensaje a un actor determinado, y su valor debe ser un identificador único (URI). Todo intermediario que no se corresponda con este, deberá ignorar la cabecera.

Mediante `mustUnderstand` se puede especificar si el procesamiento del encabezamiento es obligatorio u opcional para el receptor. Si toma el valor `1`, entonces este debe necesariamente reconocer el elemento, de otra manera todo el proceso falla.

El atributo `encodingStyle` se usa para definir la manera en la que serán codificadas en XML las estructuras de datos nativas de la aplicación y la plataforma utilizadas en el documento. Puede aparecer en cualquier elemento SOAP, y se aplicará sobre ese elemento y sus descendientes.

El **cuerpo** de un mensaje SOAP contiene el mensaje en sí para el destinatario final. Los elementos inmediatos descendientes pueden estar implementados mediante espacios de nombres.

Errores

El elemento `Fault` se utiliza para informar errores dentro de los mensajes SOAP, y posee los siguientes subelementos:

- `<faultcode>` - Código para identificar el error.
- `<faultstring>` - Una descripción legible del error.
- `<faultactor>` - Información sobre qué causó la falla.
- `<detail>` - Información del error específica de la aplicación.

En el espacio de nombre `http://www.w3.org/2001/06/soap-envelope` se definen cuatro tipos estándar de fallas, que se detallan en el tabla que aparece a continuación.

Código	Causa
`VersionMismatch`	Hay un espacio de nombre inválido para el elemento `Envelope`.
`MustUnderstand`	Un elemento inmediato descendiente del `Header` con el atributo `mustUnderstand` en 1 no fue procesado.
`Client`	El mensaje estaba mal formado o tenía información errónea.
`Server`	Un problema con el servidor impidió el correcto procesamiento de mensaje.

Cabe notar que estos códigos pueden ser extendidos hacia otros tipos de fallas más granulares y pormenorizados, mientras se mantiene la compatibilidad con los tipos de

fallas estándar. Por ejemplo, el código de error `Server.Database` deriva de `Server`. La notación separada por punto indica que el código de la izquierda es más general que el de la derecha.

WSDL

Al invocar un procedimiento en un componente, se requiere conocer el nombre de la función, los parámetros que recibe, y lo que se espera devuelva una vez que concluya. WSDL (*Web Services Description Language*) es un lenguaje basado en XML para describir servicios web y su acceso. Los documentos WSDL permiten a las herramientas de generación de código simplificar la creación de clientes de los servicios web, y forman una parte fundamental en el proceso de descubrimiento de los mismos.

Algunas de las ventajas que tiene la utilización de WSDL son:

1. Facilita la creación y mantenimiento de servicios web al proveer de un enfoque más estructurado para la definición de las interfaces y contratos.
2. Facilita el acceso a los servicios al reducir la programación requerida por los clientes.
3. Permite implementar cambios, los cuales, gracias al descubrimiento dinámico, impactan en los clientes de manera automática.

Sin embargo, WSDL no está completo. De momento no posee soporte para el versionamiento, de manera que los clientes no tienen manera de saber si se está trabajando con una descripción distinta. Esto puede producir serios problemas si las interfaces son modificadas.

El Documento WSDL

La descripción de un servicio web especifica la interfaz abstracta a través de la cual un cliente se comunica con el proveedor del servicio, así como también detalles específicos sobre cómo dicha interfaz ha sido implementada. Para ello, se definen cuatro aspectos: tipos de datos, mensajes, interfaces y servicios.

La estructura básica de un documento WSDL se presenta en el código siguiente:

```
<?xml version="1.0"?>
<definitions>
    <types>
        <!-- Definición de tipos de datos a ser usados en el
servicio. -->
    </types>
    <message>
        <!-- Información a ser transferida. -->
    </message>
    <portType>
        <!-- Conjunto de operaciones. -->
    </portType>
    <binding>
        <!-- Especificación de protocolos y formato de datos.
-->
    </binding>
    <service>
        <!-- Especificación de servicios. -->
    </service>
</definitions>
```

Un servicio (`<service>`) es un conjunto de puertos. La definición abstracta de un puerto (`<portType>`) describe al servicio web, y se compone de una colección de operaciones (`<operation>`) que establecen el intercambio ordenado de mensajes (`<message>`). Los mensajes están integrados por una o más partes (`<part>`) de un tipo determinado, y cada una de estas puede verse como los parámetros en una función. El tipo de dato utilizado por las partes de los mensajes puede precisarse dentro del mismo documento (`<types>`), o bien mediante un estándar de tipos de datos, como los esquemas XML. Mediante la definición específica del puerto (`<binding>`) se determinan los protocolos usados por este.

Tipos de Datos

La interoperabilidad entre aplicaciones ejecutándose en diferentes sistemas operativos y plataformas encuentra su principal dificultad en la representación de los tipos de datos. Diferentes lenguajes de programación no sólo poseen diferentes definiciones sobre sus tipos de datos primitivos, sino también cómo son expresados al ser enviados a través de la red.

A los fines de posibilitar una interoperabilidad entre plataformas, debe existir un mecanismo mediante el cual el servidor y el cliente acuerden en la utilización de datos comunes y su representación. En WSDL, el método principal para definir estas estructuras de datos compartidas es la especificación de esquema de XML (*XML Schema*) del W3C. Sin embargo, vale aclarar que, al describir un servicio, es posible utilizar cualquier otra forma de definición de tipos de datos. Cualquiera sea, tanto el proveedor del servicio como sus clientes deben acordar y manejarla de igual manera; de otra manera, la descripción del servicio resulta inútil. Es por este motivo que los autores del WSDL recomiendan utilizar los esquemas de XML: no dependen de plataforma alguna.

El objetivo de los esquemas es definir una clase de documentos XML. Si los datos a ser enviados en los mensajes pueden ser representados mediante XML, entonces estos esquemas pueden utilizarse para describir las reglas que definen a estos datos.

Profundizar en los esquemas de XML escapa de los fines de este trabajo, sin embargo, entenderlos es de vital importancia si desea crear servicios complejos que intercambien estructuras de datos complejas.

La Interfaz y la Implementación

La **interfaz** de un servicio web no difiere demasiado de la de una clase en la programación orientada a objetos. Se tienen los mensajes de entrada (conjunto de parámetros enviados para una operación), mensajes de salida (conjunto de valores devueltos por la operación), y mensajes de errores (el conjunto de condiciones de error que pueden darse al se invocada una función). En WSDL, a esta interfaz se la define en el elemento `<portType>`.

Con WSDL también es posible especificar la **implementación** de la interfaz dada dentro del elemento `<binding>`. Esta implementación puede dividirse en dos partes: los *protocolos* y el *servicio*.

Los **protocolos** determinan, entre otras cosas, la forma de comunicación que usarán los protocolos, y esto puede ser del tipo RPC (`rpc`) o documento (`document`). RPC (*Remote Procedure Call*) indica que los mensajes SOAP cumplirán el convenio

RPC SOAP, mientras que una comunicación como documento establece que los mensajes SOAP transportarán XML arbitrario.

El **servicio** establece las operaciones involucradas, la *acción SOAP* (valor de la cabecera `SOAPAction` cuando se utilice como transporte HTTP) de cada uno, y los parámetros de entrada y valores devueltos.

Anexo C - Reglas de Producción del *SEVAC*

En este anexo se presentan las **reglas de producción** obtenidas como resultado de la etapa de **formalización** del **SEVAC** (ver capítulo 8).

BCG

```
IF NOT(bcg-aplicada = TRUE) AND edad < 5 años THEN
    SET aplicar-bcg = TRUE

IF aplicar-bcg = TRUE AND edad < 7 dias THEN
    SET reportar-esquema-normal = TRUE

IF aplicar-bcg = TRUE AND NOT(edad < 7 dias) AND edad < 1 año
THEN
    SET reportar-bcg-esquema-alternativo = TRUE

IF aplicar-bcg = TRUE AND NOT(edad < 1 año) THEN
    SET reportar-bcg-fuera-esquema = TRUE
```

Hepatitis B

```
IF edad < 1 mes AND NOT( hb-1-aplicada = TRUE ) THEN
    SET aplicar-hb-1-recien-nacido = TRUE

IF edad > 11 años AND NOT( hb-1-aplicada = TRUE ) THEN
    SET aplicar-hb-1-mayores = TRUE

IF hb-1-aplicada = TRUE AND NOT( hb-2-aplicada = TRUE ) THEN
    SET esquema-hb-2-completo = FALSE

IF esquema-hb-2-completo = FALSE AND edad > 11 años
    AND DIFF( fecha-nacimiento, fecha-inmunizacion-hb-1 ) < 1
mes THEN
        SET chequear-dpthb-2 = TRUE

IF NOT( hb-2c-aplicada = TRUE ) AND NOT( dpthb-2-aplicada =
TRUE ) THEN
    SET esquema-dpthb-2-completo = FALSE

IF chequear-dpthb-2 = TRUE AND esquema-dpthb-2-completo =
FALSE THEN
```

```
    SET continuar-esquema-dpthb-2 = TRUE

IF DIFF( fecha-nacimiento, fecha-inmunizacion-hb-1 ) >= 11
años
    AND esquema-hb-2-completo = FALSE THEN
        SET chequear-diferencia-hb-1 = TRUE

IF DIFF( fecha-inmunizacion-hb-1, fecha-actual ) > 1 mes
    AND chequear-diferencia-hb-1 = TRUE THEN
        SET continuar-esquema-hb-2 = TRUE

IF chequear-diferencia-hb-1 = TRUE AND continuar-esquema-hb-2
= TRUE THEN
    SET aplicar-hb-2 = TRUE

IF aplicar-hb-2 = TRUE AND NOT( edad >= 12 años) THEN
    SET reportar-hb-2-normal = TRUE

IF aplicar-hb-2 = TRUE AND edad >= 12 años AND NOT( edad >= 18
años ) THEN
    SET reportar-hb-2-fuera-esquema = TRUE

IF aplicar-hb-2 = TRUE AND edad >= 18 años THEN
    SET reportar-hb-2-alternativo = TRUE

IF hb-2c-aplicada = TRUE OR dpthb-2-aplicada = TRUE THEN
    SET esquema-dpthb-2-completo = TRUE

IF hb-1-aplicada = TRUE AND hb-2-aplicada = TRUE THEN
    SET esquema-hb-2-completo = TRUE

IF edad < 12 horas AND aplicar-hb-1-recien-nacido = TRUE THEN
    SET reportar-hb-1-normal = TRUE

IF NOT( edad < 12 horas ) AND aplicar-hb-1-recien-nacido =
TRUE THEN
    SET reportar-hb-1-fuera-esquema = TRUE

IF NOT( hb-3c-aplicada = TRUE ) AND NOT( dpthb-3-aplicada =
TRUE ) THEN
    SET esquema-dpthb-3-completo = FALSE

IF esquema-dpthb-2-completo = TRUE
    AND esquema-dpthb-3-completo = FALSE
    AND NOT( hb-3-aplicada = TRUE ) THEN
        SET completar-esquema-dpthb-3 = TRUE

IF esquema-hb-2-completo = TRUE
    AND DIFF( fecha-inmunizacion-hb-2, fecha-actual ) > 1 mes
```

```
    AND NOT( hb-3-aplicada = TRUE ) THEN
        SET completar-esquema-hb-3 = TRUE

IF completar-esquema-dpthb-3 = TRUE OR completar-esquema-hb-3
= TRUE THEN
    SET aplicar-hb-3 = TRUE

IF aplicar-hb-3 = TRUE AND NOT( edad >= 12 años ) THEN
    SET reportar-hb-3-normal = TRUE

IF edad >= 12 años AND aplicar-hb-3 = TRUE AND NOT( edad >= 18
años ) THEN
    SET reportar-hb-3-fuera-esquema = TRUE

IF edad >= 18 años AND aplicar-hb-3 = TRUE THEN
    SET reportar-hb-3-alternativo = TRUE

IF aplicar-hb-1-mayores = TRUE AND NOT( edad >= 12 años ) THEN
    SET reportar-hb-1-dosis-pediatrica = TRUE

IF aplicar-hb-1-mayores = TRUE
    AND edad >= 12 años
    AND NOT( edad >= 18 años ) THEN
        SET reportar-hb-1-fuera-esquema = TRUE

IF aplicar-hb-1-mayores = TRUE AND edad >= 18 años THEN
    SET reportar-hb-1-adultos = TRUE
```

Sabin

```
IF edad >= 2 meses AND NOT( edad >= 4 meses ) THEN
    SET tiene-2-meses = TRUE

IF edad <= 18 años AND NOT( edad >= 2 años ) THEN
    SET tiene-12-18-anios = TRUE

IF tiene-2-meses = TRUE AND NOT( sabin-1-aplicada = TRUE )
THEN
    SET aplicar-sabin-1 = TRUE

IF tiene-12-18-anios = TRUE AND NOT( sabin-1-aplicada = TRUE )
THEN
    SET aplicar-sabin-1-fuera-esquema = TRUE

IF edad >= 4 meses AND NOT( edad >= 6 meses ) THEN
    SET tiene-4-meses = TRUE
```

```
IF sabin-1-aplicada = TRUE
    AND DIFF( sabin-1-fecha-inmunizacion, fecha-actual ) >= 1
mes
    AND tiene-4-meses = TRUE
    AND NOT( sabin-2-aplicada = TRUE ) THEN
        SET aplicar-sabin-2 = TRUE

IF sabin-1-aplicada = TRUE
    AND DIFF( sabin-1-fecha-inmunizacion, fecha-actual ) >= 1
mes
    AND tiene-12-18-anios = TRUE
    AND NOT( sabin-2-aplicada = TRUE ) THEN
        SET aplicar-sabin-2-fuera-esquema = TRUE

IF edad >= 6 meses AND NOT( edad >= 18 meses ) THEN
    SET tiene-6-meses = TRUE

IF sabin-2-aplicada = TRUE
    AND tiene-6-meses = TRUE
    AND DIFF( sabin-2-fecha-inmunizacion, fecha-actual ) >= 1
mes
    AND NOT( sabin-3-aplicada = TRUE )
        SET aplicar-sabin-3 = TRUE

IF sabin-2-aplicada = TRUE
    AND tiene-12-18-anios = TRUE
    AND DIFF( sabin-2-fecha-inmunizacion, fecha-actual ) >= 1
mes
    AND NOT( sabin-3-aplicada = TRUE ) THEN
        SET aplicar-sabin-3-fuera-esquema = TRUE

IF edad >= 18 meses AND NOT( edad >= 2 años ) THEN
    SET tiene-18-meses = TRUE

IF sabin-3-aplicada = TRUE
    AND tiene-18-meses = TRUE
    AND DIFF( sabin-3-fecha-inmunizacion, fecha-actual ) >= 6
meses
    AND NOT( sabin-4-aplicada = TRUE ) THEN
        SET aplicar-sabin-4 = TRUE

IF sabin-3-aplicada = TRUE
    AND tiene-12-18-anios = TRUE
    AND DIFF( sabin-3-fecha-inmunizacion, fecha-actual ) >= 6
meses
    AND NOT( sabin-4-aplicada = TRUE ) THEN
        SET aplicar-sabin-4-fuera-esquema = TRUE

IF sabin-4-aplicada = TRUE
```

```
    AND edad >= 5 años
    AND edad <= 6 años
    AND edad-inmunizacion-sabin-4 < 2 años
    AND NOT( sabin-99-aplicada = TRUE ) THEN
        SET aplicar-sabin-99 = TRUE
```

Quíntuple (Pentavalente)

```
IF edad >= 2 meses AND edad < 5 años THEN
    SET tiene-edad-dpthb = TRUE

IF tiene-edad-dpthb = TRUE AND NOT( dpthb-1-aplicada = TRUE )
THEN
    SET aplicar-dpthb-1 = TRUE

IF aplicar-dpthb-1 = TRUE AND NOT( edad >= 4 meses ) THEN
    SET reportar-dpthb-1-normal = TRUE

IF aplicar-dpthb-1 = TRUE AND edad >= 4 meses THEN
    SET reportar-dpthb-1-alternativo = TRUE

IF reportar-dpthb-1-normal = TRUE
    OR reportar-dpthb-2-normal = TRUE
    OR reportar-dpthb-3-normal = TRUE THEN
        SET reportar-dpthb-esquema-normal = "Aplicación
Óptima"

IF reportar-dpthb-1-alternativo = TRUE
    OR reportar-dpthb-2-alternativo = TRUE
    OR reportar-dpthb-3-alternativo = TRUE THEN
    SET reportar-dpthb-esquema-alternativo = "Aplicación
Esquema Alterantivo

IF dpthb-1-aplicada = TRUE
    AND tiene-edad-dpthb = TRUE
    AND edad >= 4 meses
    AND DIFF( fecha-inmunizacion-dpthb-1, fecha-actual ) >= 1
meses
    AND NOT( dpthb-2-aplicada = TRUE ) THEN
        SET aplicar-dpthb-2 = TRUE

IF aplicar-dpthb-2 = TRUE AND NOT( edad >= 6 meses ) THEN
    SET reportar-dpthb-2-normal = TRUE

IF aplicar-dpthb-2 = TRUE AND edad >= 6 meses THEN
    SET reportar-dpthb-2-alternativo = TRUE
```

```
IF tiene-edad-dpthb = TRUE
    AND dpthb-2-aplicada = TRUE
    AND edad >= 6 meses
    AND DIFF( fecha-inmunizacion-dpthb-2, fecha-actual ) >= 1
meses
    AND NOT( dpthb-3-aplicada = TRUE ) THEN
        SET aplicar-dpthb-3 = TRUE

IF aplicar-dpthb-3 = TRUE AND edad <= 18 meses THEN
    SET reportar-dpthb-3-normal = TRUE

IF aplicar-dpthb-3 = TRUE AND NOT( edad <= 18 meses ) THEN
    SET reportar-dpthb-3-alternativo = TRUE
```

Cuádruple

```
IF edad >= 15 meses
    AND edad <= 18 meses
    AND NOT( dpth-99-aplicada = TRUE )
    AND dpthb-3-aplicada = TRUE
    AND DIFF( fecha-inmunizacion-dpthb-3, fecha-actual ) >= 6
meses THEN
        SET aplicar-dpth-99 = TRUE
```

Neumococo Conjugada (13 Valente)

```
IF edad >= 2 meses AND edad < 18 meses THEN
    SET tiene-edad-neumo = TRUE

IF tiene-edad-neumo = TRUE AND NOT( neumo-1-aplicada = TRUE )
THEN
    SET aplicar-neumo-1 = TRUE

IF tiene-edad-neumo = TRUE
    AND neumo-1-aplicada = TRUE
    AND DIFF( fecha-inmunizacion-neumo-1, fecha-actual ) >= 2
meses
    AND NOT( neumo-2-aplicada = TRUE ) THEN
        SET aplicar-neumo-2 = TRUE

IF tiene-edad-neumo = TRUE
    AND neumo-2-aplicada = TRUE
    AND DIFF( fecha-inmunizacion-neumo-2, fecha-actual ) >= 2
meses
    AND NOT( neumo-3-aplicada = TRUE ) THEN
```

```
        SET aplicar-neumo-3 = TRUE
```

Triple Viral

```
IF edad = 12 meses AND NOT( srp-1-aplicada = TRUE ) THEN
    SET aplicar-srp-1 = TRUE

IF srp-1-aplicada = TRUE
    AND edad >= 5 años
    AND edad <= 6 años
    AND NOT( srp-2-aplicada = TRUE )
    AND NOT( sr-0-aplicada = TRUE ) THEN
        SET aplicar-srp-2 = TRUE

IF edad = 11 años
    AND NOT( srp-2-aplicada = TRUE )
    AND NOT( sr-0-aplicada = TRUE ) THEN
        SET aplicar-srp-2-esquema-alternativo = TRUE
```

Doble Viral

```
IF edad >= 18 años
    AND NOT( srp-2-aplicada = TRUE )
    AND NOT( sr-0-aplicada = TRUE ) THEN
        SET aplicar-sr-0 = TRUE
```

Triple Bacteriana Celular

```
IF edad >= 5 años
    AND edad <= 6 años
    AND dpth-99-aplicada = TRUE
    AND NOT( dpt-99-aplicada = TRUE ) THEN
        SET aplicar-dpt-99 = TRUE
```

Triple Bacteriana Acelular

```
IF edad = 11 años
    AND dpt-99-aplicada = TRUE
    AND NOT( dpta-99-aplicada = TRUE ) THEN
        SET aplicar-dpta-99 = TRUE
```

Hepatitis A

```
IF edad = 12 meses
    AND NOT( ha-0-aplicada = TRUE ) THEN
        SET aplicar-ha-0 = TRUE
```

VPH

```
IF edad = 11 años AND NOT( vph-1-aplicada = TRUE ) THEN
    SET aplicar-vph-1 = TRUE

IF edad = 11 años
    AND vph-1-aplicada = TRUE
    AND DIFF( fecha-inmunizacion-vph-1, fecha-actual ) >= 1
meses
    AND NOT( vph-2-aplicada = TRUE ) THEN
        SET aplicar-vph-2 = TRUE

IF edad = 11 años
    AND vph-2-aplicada = TRUE
    AND DIFF( fecha-inmunizacion-vph-2, fecha-actual ) >= 3
meses
    AND NOT( vph-3-aplicada = TRUE ) THEN
        SET aplicar-vph-3 = TRUE
```