Tesis de Licenciatura

“Estrategias automáticas de interrogatorio médico”

Tesista: Carlos E. Ferro
Director: Dan Rozenfarb

Marzo / 2009

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

ceferro@ciudad.com.ar
drozenfa@ciudad.com.ar

Resumen

Este trabajo comenzó con una propuesta de investigación sobre mejoras para una aplicación ya existente, desarrollada en Smalltalk. Esta aplicación se llama ExpertCare y se trata de una herramienta para soporte de decisión en un servicio telefónico de despacho de ambulancias.

ExpertCare analiza los síntomas reportados por los pacientes y sugiere preguntas que apunten a otros síntomas, hasta consolidar un diagnóstico presuntivo y establecer si la llamada requiere una ambulancia y/o atención médica urgente.

La aplicación cuenta con una base de conocimiento de síndromes, definidos en términos de síntomas. También utiliza un sistema de reglas para determinar qué nuevas preguntas deberían hacerse. Pero la construcción y el mantenimiento de ese sistema de reglas es compleja y costosa, además de requerir mucha colaboración por parte de los expertos del dominio. Por estas razones nos propusimos desarrollar estrategias automáticas que pudieran operar sobre la base de conocimiento y decidir cuáles serían las mejores preguntas para alcanzar el diagnóstico presuntivo.

Para esto, tuvimos que construir un "laboratorio virtual" donde pudiéramos simular llamadas telefónicas de pacientes y sesiones de interrogatorio y diagnóstico. En ese ambiente desarrollamos y probamos varias estrategias, midiendo y mejorando su performance hasta alcanzar valores que las hacen utilizables en la aplicación.

Abstract

The present work started as a research on improvements for an existing Smalltalk application. This application, called ExpertCare, is a decision support tool for telephonic medical triage and ambulance dispatch.

ExpertCare analyzes symptoms reported by patients and suggests questions pointing to new symptoms, in order to make a presumptive diagnosis and assess if the call requires an ambulance and/or urgent attention.

The application has a knowledge base of syndromes defined in terms of symptoms and a system of rules to determine which questions should be asked. But construction and maintenance of such a system of rules is complex and costly, requiring intensive domain expert collaboration; therefore, we aim at developing an automatic strategy which can operate upon the knowledge base and decide which questions are best asked in order to achieve the presumptive diagnosis.

For that purpose, we needed to build a "virtual laboratory" where we can simulate patient calls and diagnostic sessions. In this environment we developed and tested several strategies, measuring and enhancing the performance until we reached values that grant their usage in the main application, in real conditions.

Agradecimientos

Este es un trabajo tan largamente demorado que ha sido como una deuda pendiente por la mayor parte de mi vida adulta (hasta ahora). Desde que empecé mi primer proyecto de tesis hasta el momento actual, casi diez años pasaron.
En un tiempo tan largo, se acumularon muchas cosas para agradecer y muchas personas y grupos a las que debería agradecerles. Estoy condenado de antemano a la incompletitud, la omisión y la parcialidad. Por lo tanto, mencionaré sólo los obvios y grandes grupos genéricos, para disminuir el impacto.
A mis compañeros de trabajo, de los que todavía aprendo día a día.
A quienes fueron mis compañeros de cátedra en las materias de Objetos, con quienes empezamos a recorrer un camino fascinante e infinito, plagado de discusiones interesantes y enriquecedoras. Y especialmente a Dan, compañero, amigo y ahora director de tesis.
A mi familia (especialmente mi esposa e hija), por demostrar que no todo es trabajo y academia, y darle un sentido al resto.
A mis amigos, que se encargaron periódica y persistentemente de pincharme para que no olvidara que tenía que terminar la tesis.
A toda la gente del Departamento de Computación (docentes, no-docentes y alumnos), que siempre dio un buen ámbito y estímulo para el trabajo académico.

Índice

Introducción

Problema inicial

Partimos de una aplicación existente llamada ExpertCare. Es un sistema experto que guía a un operador que dialoga telefónicamente con un paciente, en la obtención de un diagnóstico médico presuntivo y sugiere una conducta a seguir.

Se procede haciendo preguntas a la persona que llama, para determinar la presencia de diversos síntomas e ir avanzando hacia la confirmación de un diagnóstico que indique un síndrome concreto.

El objetivo de este trabajo es utilizar tecnología de objetos e inteligencia artificial para implementar estrategias que mejoren la conducción de este interrogatorio, y llegar a una confección automática o semiautomática de los interrogatorios, a partir de la información de la base de conocimientos del sistema.

Se cuenta con una implementación realizada en un ambiente de objetos. Esta implementación tuvo un período prolongado de uso, con resultados satisfactorios. Se utilizó para cubrir una población de 1.500.000 personas en un área urbana grande.

Esta implementación fue realizada por una empresa pequeña, dedicando pocos recursos a ello. El resultado es una combinación ad-hoc de técnicas diversas para superar las dificultades concretas del problema.

La aplicación cuenta con una base de conocimiento de *síntomas* y de *síndromes*. Un síndrome tiene una definición que es una expresión lógica sobre los síntomas que la caracterizan, por ejemplo, (‘fiebre alta’ y no ‘vómitos’) o ‘diarrea’. Esta base de conocimiento cuenta con un certificado de la Facultad de Medicina de la UBA, aseverando que refleja el estado actual del arte en la práctica médica.

Las reglas guían el interrogatorio, dictaminando qué preguntar ante cada situación. Este conocimiento no se encuentra en los libros pues los interrogatorios no están prescriptos, salvo en casos excepcionales.

Para dar una idea del tamaño y complejidad del problema, la base de conocimiento de esta implementación cuenta con alrededor de 700 síndromes, y unos 2400 síntomas. Estos elementos son obtenidos esencialmente de libros de medicina, con cierto procesamiento para formalizarlos y adaptarlos al formato de representación utilizado. Podría decirse que esta componente es la del conocimiento “enciclopédico”.

Hay también un gran número de *reglas* para la conducción de los interrogatorios (aproximadamente 3200). El conjunto de reglas conforma aproximadamente el 50% de la cantidad de objetos en la base de conocimiento. Es decir, que podríamos considerar que representa el 50% del tamaño o del contenido. Pero por otro lado, hemos comprobado que representa el 80% de la complejidad del sistema, y el 90% del costo de construcción y mantenimiento.

Todo el trabajo arrancó con una gran pregunta: ¿Sería suficiente con la información capturada en la base de conocimiento para conducir el interrogatorio? Esta pregunta nace del deseo de prescindir, en la medida de lo posible, de las reglas de interrogatorio.

El uso de reglas confeccionadas manualmente por los expertos constituyó y constituye uno de los problemas principales para el desarrollo y mejora de la aplicación. La fuente de estas reglas son los médicos (de diferentes especialidades) y su experiencia personal. Esto trae inmediatamente el problema de que, en el panorama médico de Argentina prácticamente no hay interrogatorios ordenados, consensuados ni rigurosos. Además no suelen ser eficientes ni completos, pero sí proclives a error dada su alta improvisación y manualidad. Por otra parte, los médicos y su tiempo son un recurso costo-

so y difícil de manejar. Se trata de un dominio en el que acceder a los conocimientos de los expertos es complicado, y además las consecuencias de un error pueden impactar directamente en la salud (o la vida) de un paciente.

Por las características de la aplicación, además, es difícil revisar el funcionamiento del conjunto de reglas, ya que el "resultado" es un conjunto muy variado de interrogatorios que surgen de la interacción de esas reglas con las respuestas posibles de los pacientes.

Las reglas no sólo son costosas de confeccionar, revisar y poner a punto: también son una traba para mejoras y modificaciones. Por ejemplo, realizar una adaptación para un servicio pediátrico, que parecería una pequeña modificación de la base de conocimiento, sacando los síndromes que no apliquen a la población infantil y tal vez agregando algunos más, requeriría la revisión completa de las reglas y posiblemente una nueva construcción de otro conjunto de reglas para los interrogatorios, muy diferente del original.

Este trabajo no está motivado por los requerimientos de un cliente particular, aunque sí tenemos la expectativa de mejorar la aplicación actual en varios aspectos: mantenibilidad, adaptabilidad, economía y simplicidad.

Tenemos la posibilidad de medir las mejoras, ya que la versión actual está en funcionamiento y podemos comparar nuestros resultados con los del producto ExpertCare. Las mejoras se verían si logramos una disminución consistente en la cantidad de preguntas necesaria para resolver una consulta, disminuyendo el tamaño de la base de conocimiento (al eliminar la mayoría de las reglas de interrogatorio).

Otro objetivo que tenemos es ver si es posible inducir una guía para el interrogatorio a partir de las definiciones, sin necesidad de reglas explícitas sobre las preguntas. Nos parece que en la mayoría de los dominios donde hay conocimiento experto, sería más fácil conseguir definiciones de los conceptos, normalmente consensuadas y relativamente fáciles de encontrar y codificar a partir de la bibliografía especializada, en tanto que los procedimientos de interrogatorio tendrían mayor grado de informalidad, mayor dependencia de un grupo particular de expertos y sus prácticas y mucho mayor costo de adquisición y mantenimiento.

Lamentablemente, encontramos bastante bibliografía sobre sistemas basados en reglas, pero ninguna con un enfoque como éste.

Enfoque de la solución

Los primeros pasos estuvieron encaminados a construir un pequeño *framework* de clases y objetos que nos permitiera pensar, estudiar e implementar distintas estrategias automáticas para conducir el interrogatorio. Una vez logrado esto, utilizando el mismo *framework* podríamos compararlas y mejorarlas.

Las estrategias automáticas cuentan con distinto grado de información, obtenida de la base de conocimientos de ExpertCare. Además de la información básica de los síndromes y su definición en términos de síntomas, utilizamos información adicional, relativamente fácil de obtener, expresada como atributos de los síndromes y síntomas.

Esta información es utilizada por las estrategias o heurísticas automáticas para guiar el interrogatorio, prescindiendo de las costosas reglas. Las sucesivas preguntas se deducen en función de la información de la base de conocimientos y de las respuestas de un paciente.

Para los síndromes, tenemos un atributo sumamente importante para las decisiones del sistema, que es la *gravedad* o criticidad. Está dividida en tres valores en orden descendente: rojo, amarillo o verde. Se consideran como urgencia que justifica enviar

una ambulancia los rojos y amarillos, y es un criterio de parada de la sesión: si se encuentra un diagnóstico completo de un síndrome con esa gravedad, se termina.

Otro atributo de los síndromes es la *frecuencia* o incidencia: alta, media o baja. Esto puede utilizarse en las decisiones de qué preguntar, teniendo en cuenta que podríamos considerar más importante decidir entre síndromes de alta frecuencia que considerar los de baja.

Otro atributo de los síndromes que sirve para establecer relaciones entre ellos es el *sistema* que afectan. Puede ser sistema digestivo, circulatorio, respiratorio, etc. Este atributo también puede ser de ayuda para orientar el cuestionario, tratando de decidir un sistema afectado y luego trabajar con los síntomas de ese sistema.

Pueden surgir otros atributos de este tipo, por ejemplo la *fisiopatología*, otra tipificación de los síndromes, que puede relacionar algunos síntomas característicos de determinadas patologías.

Aunque no es estrictamente un atributo, tenemos disponible alguna información adicional sobre los síntomas, que refleja ciertas *implicaciones* entre ellos. Por ejemplo, 'Dolor abdominal alto' implica 'Dolor abdominal', y también lo hace 'Dolor abdominal bajo', y los tres pueden aparecer en las expresiones características de los síndromes. La respuesta afirmativa a 'Dolor abdominal' no nos permite concluir un valor positivo de 'Dolor abdominal bajo' ni de 'Dolor abdominal alto', pero sin embargo tenemos más evidencia a favor de ellos que de un síntoma no preguntado cualquiera.

Beneficios esperados

La automatización del procedimiento de interrogatorios nos permitiría

- optimizar los interrogatorios a los fines de realizar menos preguntas y poder determinar en menos tiempo la gravedad de la llamada.
- tener mayor cobertura y mejor utilización de la base de conocimientos
- contar con un proceso de decisión racional y verificable
- facilitar el mantenimiento de la base de conocimientos y el sistema mismo
- explicar el razonamiento para obtener un diagnóstico presuntivo.

Creemos que esto puede lograrse utilizando un enfoque científico de la cuestión, estudiando y analizando las características de la base de conocimientos actual.

Mejorar esta aplicación (ExpertCare) nos daría la posibilidad de que herramientas así se utilicen más ampliamente, con perspectivas de mejorar la atención de la salud de la población.

Su difusión también podría ayudar a introducir aplicaciones de inteligencia artificial similares, basadas en estos principios y técnicas, en otros dominios.

También creemos que es importante realizar esto con tecnologías de objetos, ampliamente usadas y aceptadas por la comunidad de programadores y "el mercado", a diferencia de las herramientas específicas de inteligencia artificial. El trabajo debería producir un framework de objetos para conceptualizar y expresar distintas estrategias de interrogatorio.

ExpertCare – Implementación actual

Ejemplo de Funcionamiento

La modalidad estándar de funcionamiento de ExpertCare es la siguiente:

En un call-center hay varios puestos de trabajo con el sistema. Cuando un operador recibe un llamado de un paciente, le solicita sus datos personales y los síntomas

que motivaron el llamado. Al ingresar esta información, ExpertCare propone nuevas preguntas que el operador transmite al paciente ingresando a su vez las respuestas correspondientes.

Junto a las nuevas preguntas se propone una lista de diagnósticos presuntivos ordenados por grado de certeza, teniendo en cuenta toda la evidencia ingresada hasta el momento.

La principal diferencia con otros sistemas del mercado es que el interrogatorio es basado en reglas, y no es arbóreo. El sistema analiza -todo el tiempo- la totalidad de la información ingresada, y vuelve a decidir en función de las reglas activadas cuál es la próxima pregunta, sin la necesidad de seguir un camino preestablecido. Para ilustrar mejor esto veamos un ejemplo de su utilización que no puede ser resuelto con un algoritmo tradicional arbóreo.

Supongamos que llama la madre de una paciente con dolor en el abdomen por debajo del ombligo.

Al tomar los datos personales la madre refiere el nombre de la paciente, el sexo femenino y una edad de 30 años. Se carga esta información como se muestra en la imagen 1.

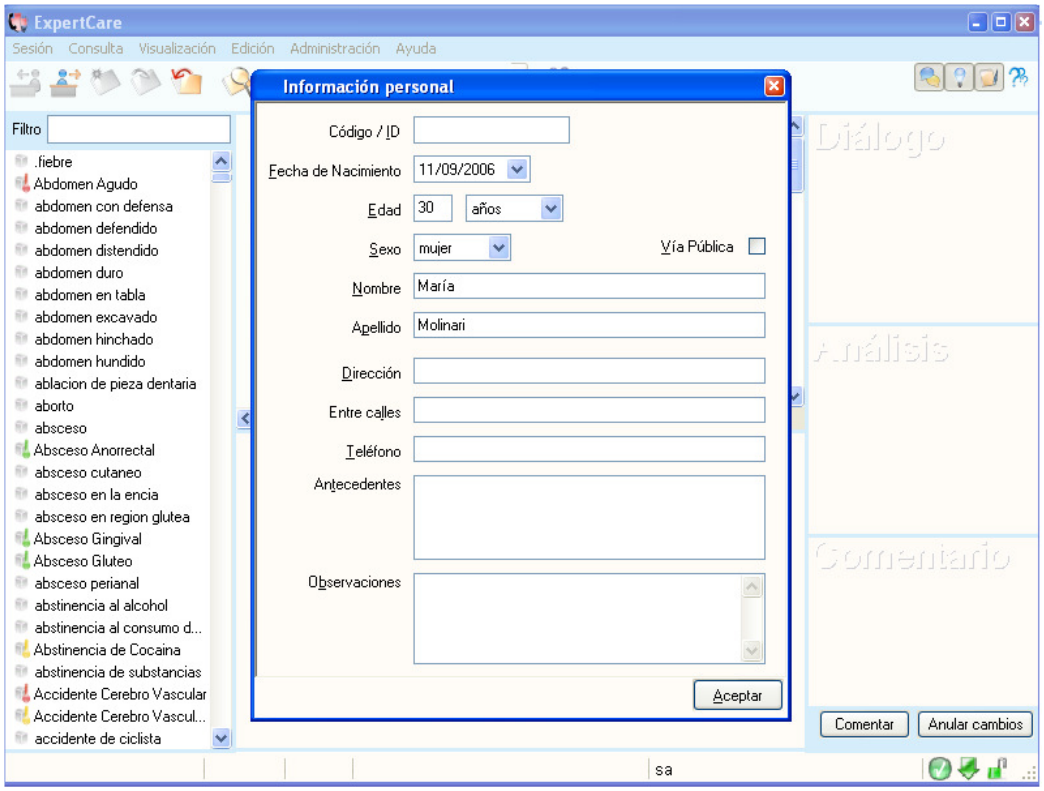


Imagen 1

Luego se ingresa el referido dolor abdominal inferior.

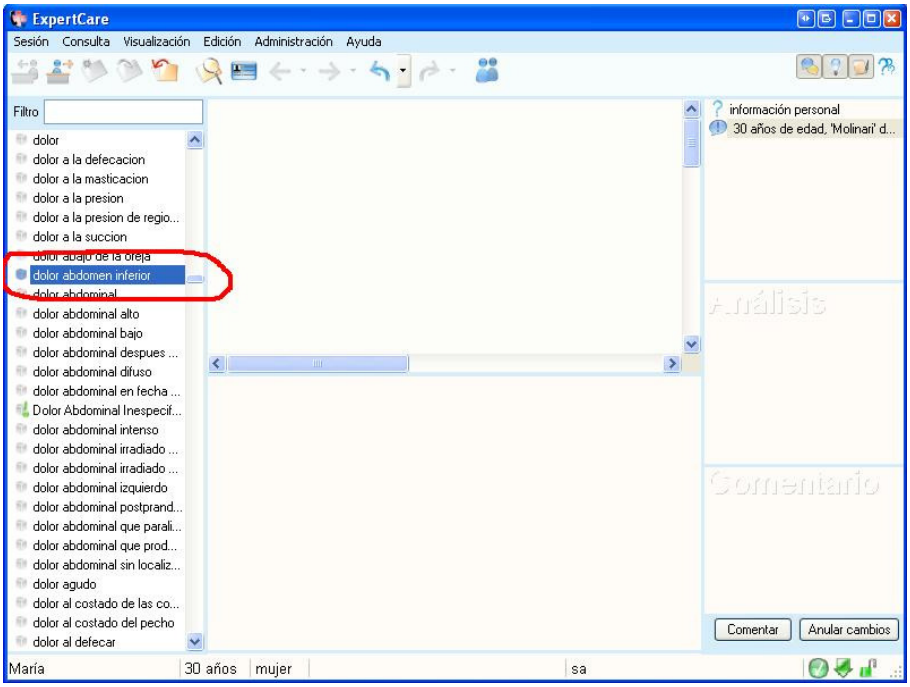


Imagen 2

Ante el conocimiento de que la paciente es mujer, con 30 años de edad y tiene dolor abdominal inferior, ExpertCare sugiere, en base a su conocimiento acumulado, preguntar por un abdomen en tabla, con el propósito de descartar una emergencia. En este ejemplo, la madre dice que no, lo cual se ingresa como vemos en la imagen 3.

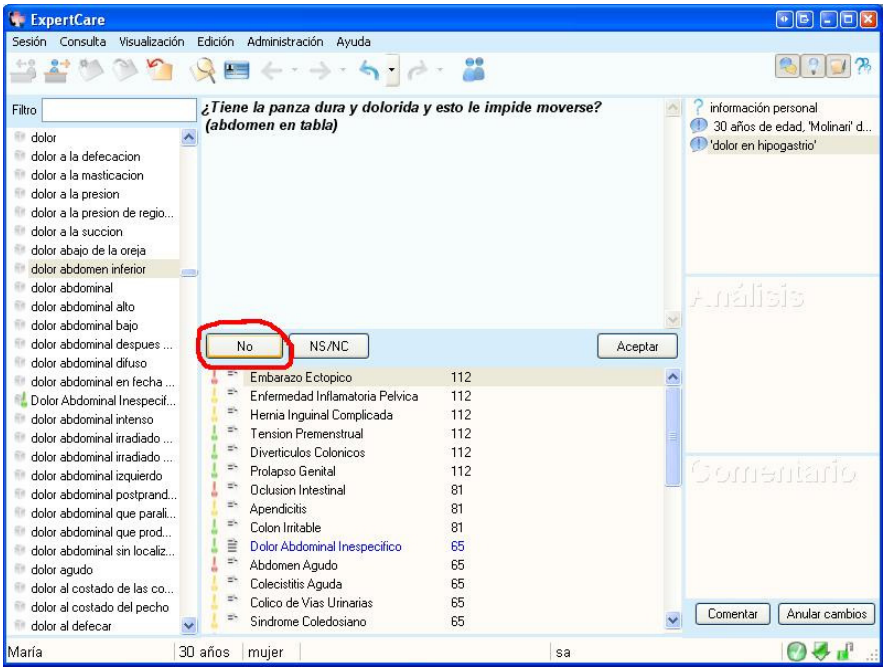


Imagen 3

Ahora el sistema considera que corresponde preguntar si existe un atraso menstrual. Mientras este diálogo ocurre, ExpertCare muestra una lista valorizada de potenciales diagnósticos presuntivos en el panel inferior. En la imagen 4 hemos señalado con flechas algunos diagnósticos relacionados con una mujer en edad fértil.

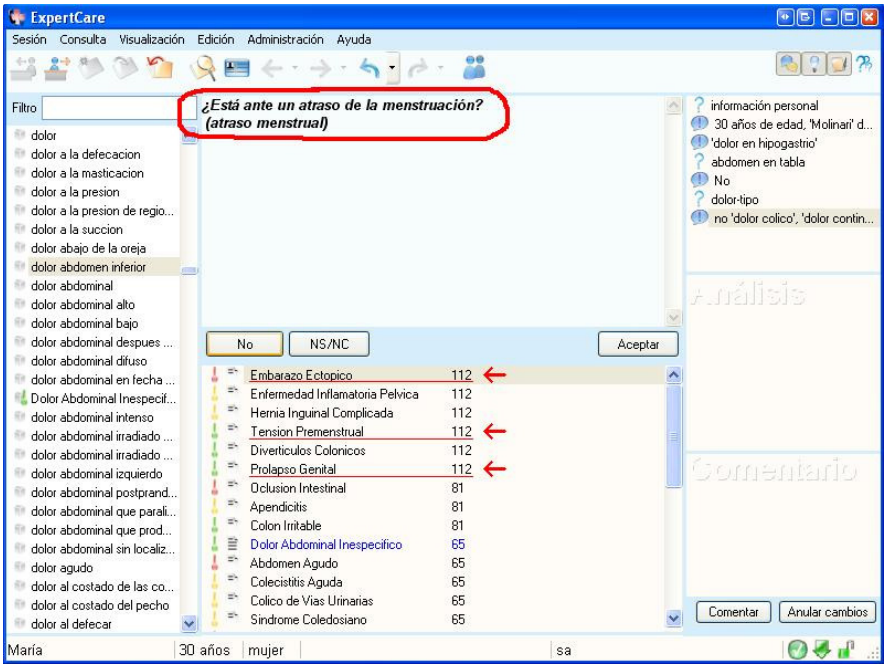


Imagen 4

Ante la pregunta del operador sobre si la paciente tiene atraso menstrual, la madre se da cuenta de un error suyo. La paciente dolorida es su hija, de 9 años. La madre entonces manifiesta su equivocación, lo cual es ingresado en el sistema.

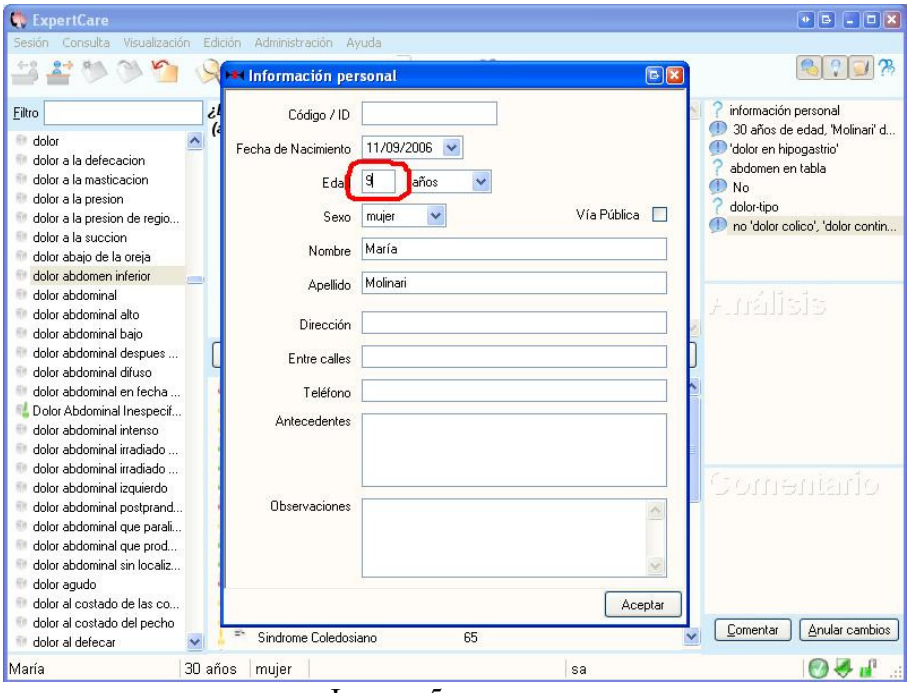


Imagen 5

El sistema toma la nueva evidencia y recalcula todo. No es necesario comenzar el interrogatorio nuevamente, como se haría en un algoritmo arbóreo clásico. En la imagen podemos observar como se cambia la pregunta de atraso menstrual por la de fiebre. Pero además se recalcula la lista de diagnósticos, que en este caso produce la desaparición de los diagnósticos señalados anteriormente.

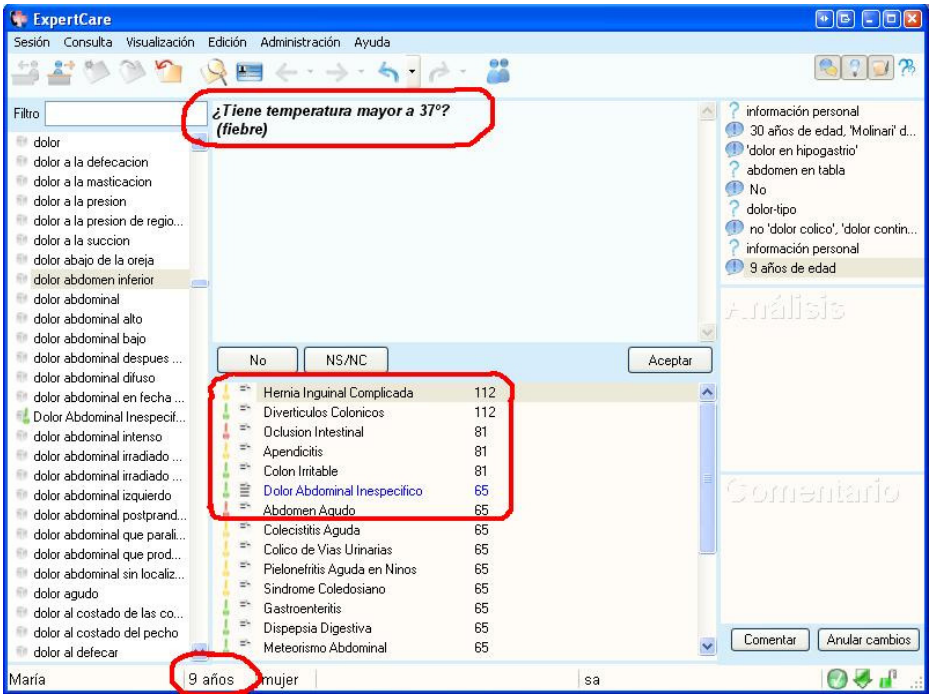


Imagen 6

Composición

ExpertCare está compuesto principalmente por un motor inteligente y una base de conocimiento médico (*Knowledge Base*, o *KB* para abreviar) que contiene las diferentes patologías (síndromes), los síntomas y las reglas que guían los interrogatorios.

Actualmente la base de conocimiento se compone de los siguientes elementos:

Tipo de elemento	Cantidad
Reglas	3.209
Síntomas	2.383
Síndromes	673
Otros	157
Total	6.422

Estos pueden agruparse básicamente en dos componentes:

A. Síntomas, Síndromes y otros

Estos elementos son obtenidos esencialmente de libros de medicina, con cierto procesamiento para formalizarlos y adaptarlos al formato de representación utilizado. Podría decirse que esta componente es la del conocimiento "enciclopédico".

B. Reglas

El conjunto de reglas conforma aproximadamente el 50% de la cantidad de objetos en la base de conocimiento. Es decir, que podríamos considerar que representa el 50% del tamaño o del contenido. Pero por otro lado, hemos comprobado que representa el 80% de la complejidad del sistema, y el 90% del costo de construcción y mantenimiento[1].

Las reglas guían el interrogatorio, dictaminando qué preguntar ante cada situación. Este conocimiento no se encuentra en los libros pues los interrogatorios no están prescriptos, salvo en casos excepcionales. La fuente de estas reglas son los médicos y su experiencia personal.

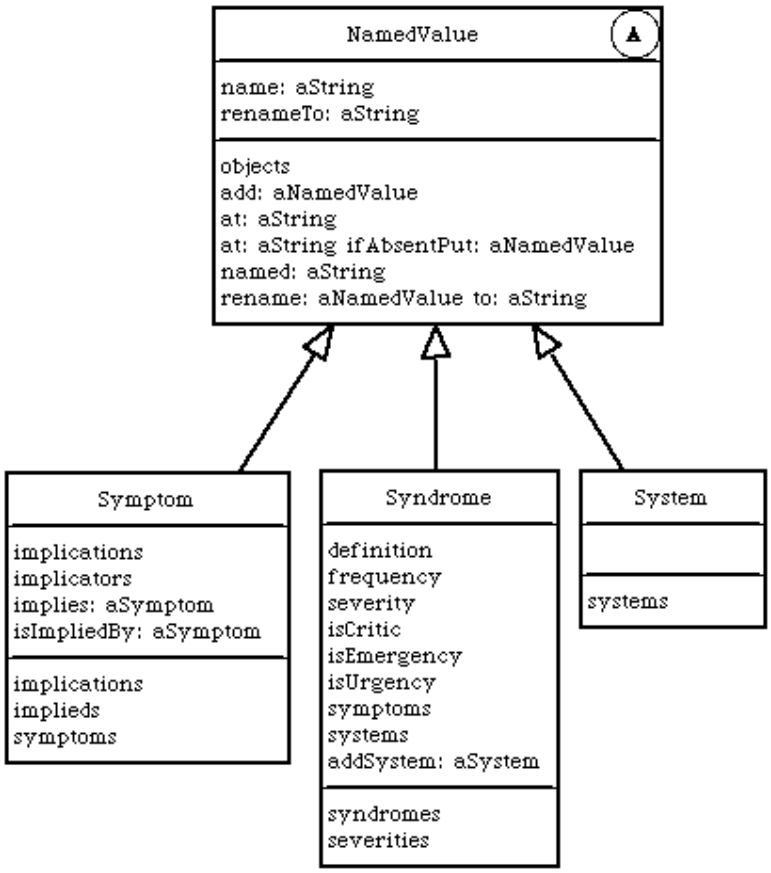
Desarrollo

Representación de los conceptos básicos

El primer paso obvio para encarar un framework era representar los conceptos más básicos ya encontrados en la descripción del problema: los síntomas y los síndromes, con todos sus atributos.

El tamaño relativamente pequeño (menos de 5000 objetos principales) nos alentaba a tenerlos todos en memoria, así que pensamos en una clase que nos permitiera guardarlos en una variable de la clase, para mantenerlos vivos frente al *garbage collector*. Como lo más importante y característico que tienen es el nombre, decidimos llamar a la clase genérica `NamedValue`, dotándola además de protocolo para acceder a las instancias por su nombre.

[1] Los costos de mantenimiento y construcción fueron calculados durante la experiencia de uso del software en Córdoba.



La implementación de casi todo este protocolo es trivial, la mayoría son accesos directos a variables de instancia. Sin embargo, hay un par de cosas que vale la pena aclarar.

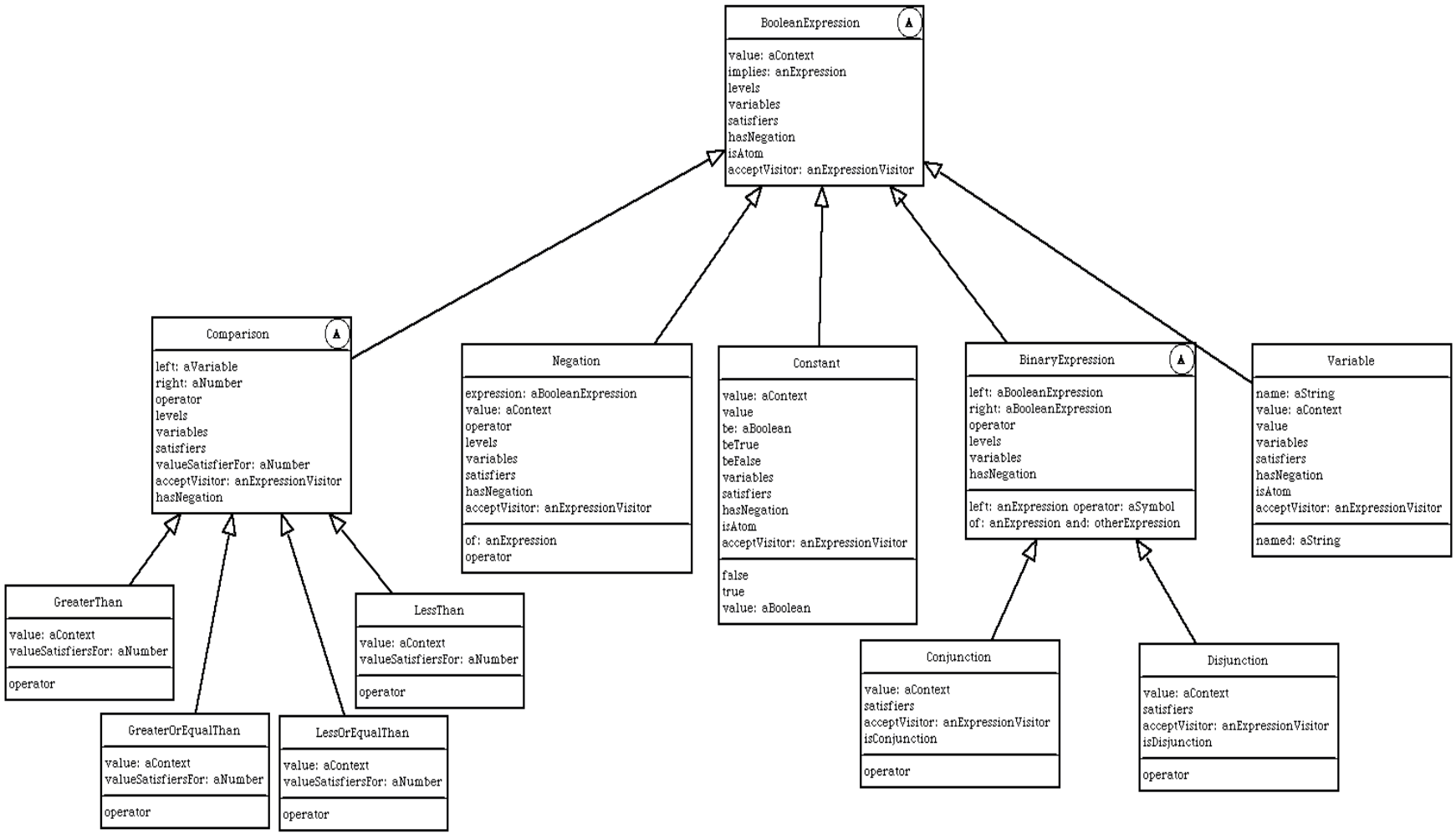
En cada clase, la variable `objects` se maneja como un diccionario, de ahí el protocolo de `#at:` y `#at:ifAbsentPut:`. En lugar de `#at:put:` se usa el `#add:`, para agregar un objeto en la clave de su nombre. Las subclases tienen un acceso con nombre más significativo (`symptoms`, `syndromes` y `systems`).

En los síntomas, se agrega todo un protocolo para manejar las implicaciones entre síntomas. Nuevamente, el conjunto de todas las implicaciones se guarda en variables de la clase, indicando quién implica a quién (`implications`) y quién es implicado por quién (`implieds`). Esto es una decisión para simplificar las búsquedas y clausuras de la implicación en uno u otro sentido.

En los síndromes hubo que utilizar una colección para indicar los sistemas, ya que en algunos casos, un síndrome afecta a más de un sistema.

Expresiones lógicas

Para completar este grupo básico de objetos, necesitamos representar las expresiones lógicas que usamos como definiciones de los síndromes. Para esto, trabajamos con la siguiente jerarquía:


BooleanExpression
value: aContext
implies: anExpression
levels
variables
satisfiers
hasNegation
isAtom
acceptVisitor: anExpressionVisitor
Comparison
left: aVariable
right: aNumber
operator
levels
variables
satisfiers
valueSatisfierFor: aNumber
acceptVisitor: anExpressionVisitor
hasNegation
Negation
expression: aBooleanExpression
value: aContext
operator
levels
variables
satisfiers
hasNegation
acceptVisitor: anExpressionVisitor
of: anExpression
operator
Constant
value: aContext
value
be: aBoolean
beTrue
beFalse
variables
satisfiers
hasNegation
isAtom
acceptVisitor: anExpressionVisitor
false
true
value: aBoolean
BinaryExpression
left: aBooleanExpression
right: aBooleanExpression
operator
levels
variables
hasNegation
left: anExpression operator: aSymbol
of: anExpression and: otherExpression
Variable
name: aString
value: aContext
value
variables
satisfiers
hasNegation
isAtom
acceptVisitor: anExpressionVisitor
named: aString
GreaterThan
value: aContext
valueSatisfiersFor: aNumber
operator
LessThan
value: aContext
valueSatisfiersFor: aNumber
operator
GreaterOrEqualThan
value: aContext
valueSatisfiersFor: aNumber
operator
LessOrEqualThan
value: aContext
valueSatisfiersFor: aNumber
operator
Conjunction
value: aContext
satisfiers
acceptVisitor: anExpressionVisitor
isConjunction
operator
Disjunction
value: aContext
satisfiers
acceptVisitor: anExpressionVisitor
isDisjunction
operator

Esta jerarquía sirve para una representación muy simplificada de las expresiones del cálculo proposicional, enriquecidas con las comparaciones. Aunque no es una representación completa ni óptima, resulta suficiente para trabajar con las definiciones de los síndromes y no es uno de nuestros objetivos principales, por lo que no profundizamos más en el tema.

El protocolo principal, además de los accesos elementales para la construcción de expresiones, es el de evaluación. El mensaje principal es `#value: aContext`, donde entendemos `aContext` como un contexto de evaluación, algo parecido a una valuación parcial, representado por un simple diccionario donde se encuentran variables y sus valores de verdad.

Las variables con las que trabajamos se caracterizan únicamente por su nombre y están asociadas uno a uno con los síntomas. El nombre establece esta relación, ya que es el mismo nombre que el del síntoma[2]. En una sesión, por ejemplo, cuando el paciente informa que sufre Dolor Abdominal Bajo, se crea una variable con ese nombre y se coloca en el contexto de evaluación, con un valor de verdad `true`.

Aparte de algún protocolo muy sencillo de *testing* (`#isAtom`, `#isConstant`, `#hasNegation`, etc.) hay otro protocolo importante para este trabajo, y es el relacionado con la noción operativa de *satisfiers*. Definimos como *satisfier* de una expresión a un conjunto mínimo de variables (con sus correspondientes valores de verdad), que hacen verdadera la evaluación de una expresión. El mensaje `#satisfiers` devuelve una colección de todos esos conjuntos. Para dar los ejemplos más básicos, estos son los *satisfiers* de las expresiones elementales:

Expresión	**Satisfiers**
A	**A = true**
no A	**A = false**
A y B	**(A = true, B = true)**
A o B	**(A = true)** **(B = true)**

Es fácil ver que en los únicos casos donde aparece más de un conjunto de *satisfiers*, es cuando hay una disyunción involucrada.

Como dijimos, nuestras expresiones se extienden con comparaciones entre el valor de un *síntoma cuantificado* y un número fijo, por ejemplo "Frecuencia Cardíaca" > 90[3]. También fue necesario extender el concepto de *satisfier* para estos casos, usando un *rango* de valores numéricos, en lugar de un valor booleano. El *satisfier* de la expresión anterior es la variable con un valor dentro del intervalo (90, +infinito).

El protocolo de las expresiones se completa con el mensaje de *double-dispatch* necesario para usar *visitors* que puedan recorrer las expresiones: `#acceptVisitor: anExpressionVisitor`.

Nos referimos a un Visitor en el sentido del conocido patrón de diseño [GHJV/95], y utilizamos tres de ellos: `PrintingExpressionVisitor` para tener

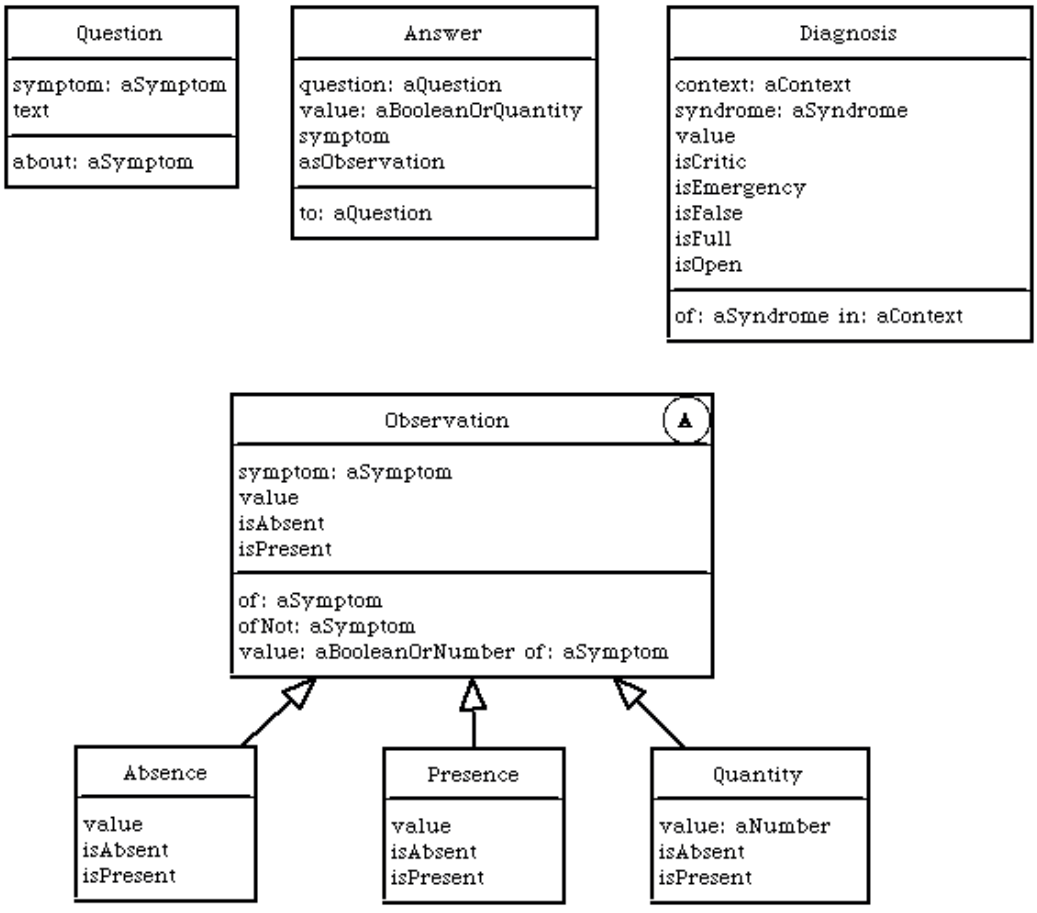
[2] Este es un punto dudoso del diseño, ya que es el lugar donde la jerarquía de las expresiones booleanas toma contacto con el dominio. Para disminuir ese contacto, uso el nombre del síntoma en lugar del síntoma, ya que así al menos podría reutilizar la jerarquía con otros objetos nombrados.

[3] Los síntomas cuantificados llevan asociada alguna unidad de medida, pero para simplificar, omití las unidades y asumí que los valores siempre se trabajan en la misma unidad.

una buena representación textual de las expresiones (sin paréntesis superfluos), `SolvingExpressionVisitor` para resolver y simplificar parcialmente expresiones (por ejemplo A y B se puede reducir a B si el valor de A se conoce como true) y `SyndromeExpansionVisitor` para resolver un detalle técnico de las definiciones de los síndromes: a veces aparece un síndrome en la definición de otro, y lo que hace este visitor es equivalente a una expansión sintáctica que coloca la definición del síndrome en lugar de su referencia, para tener todas las definiciones únicamente en términos de los síntomas[4].

Representación de conceptos de diagnóstico y sesión de ExpertCare

Hay un segundo grupo de clases, no tan simples y directas como los síntomas y síndromes, relacionados con el modo de funcionamiento de ExpertCare y la idea de una sesión de diagnóstico que se lleva a cabo con preguntas y respuestas.



[4] Este es el único caso en que una variable se liga a algo que NO es un síntoma. En este caso, en las definiciones aparece un síndrome, y la variable quedaría ligada al síndrome. Para evitar esto, durante la importación de la base de conocimiento utilizamos este `visitor` que mediante la expansión garantiza el uso correcto de las variables.

La más sencilla, `Question`, tiene poco uso y es utilizada sólo para representar una de las preguntas que haría el operador. La pregunta está siempre relacionada con un síntoma particular. Y para no dejar estas preguntas sin respuesta, `Answer` modela la respuesta a una pregunta concreta. Esta respuesta tiene un valor que puede ser booleano (un síntoma está o no está presente según la descripción del paciente) o numérico, para el caso de síntomas cuantificados.

La respuesta está muy emparentada con y se traduce en una `Observation`, la observación sobre presencia, ausencia o cantidad de un síntoma. Cabe aclarar que esta distinción entre pregunta, respuesta y observación se derivó de la jerarquía existente en la aplicación, donde estos conceptos tienen usos distintos. En este trabajo esa distinción no resultó necesaria y estas tres clases se podrían haber resumido en una, simplificando el diseño. No encaramos esa refactorización porque tampoco fue necesaria para nuestros objetivos.

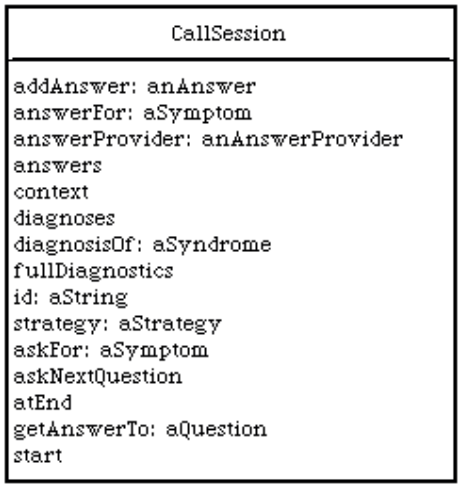
Completa este grupo la clase `Diagnosis`, que representa el diagnóstico de un síndrome concreto en el contexto de una sesión. Su `value` es el resultado de evaluar la definición del síndrome en dicho contexto, pudiendo entonces estar completamente confirmado (`isFull`) si la expresión de la definición evalúa a verdadero, completamente descartado[5] (`isFalse`) si la expresión de la definición evalúa a falso, o en algún estado intermedio (`isOpen`) si no se puede evaluar completamente la expresión con la información (valores de variables, respuestas del paciente) actual del contexto.

El resto del protocolo no requiere mayor explicación porque son accesos triviales a los colaboradores de cada objeto.

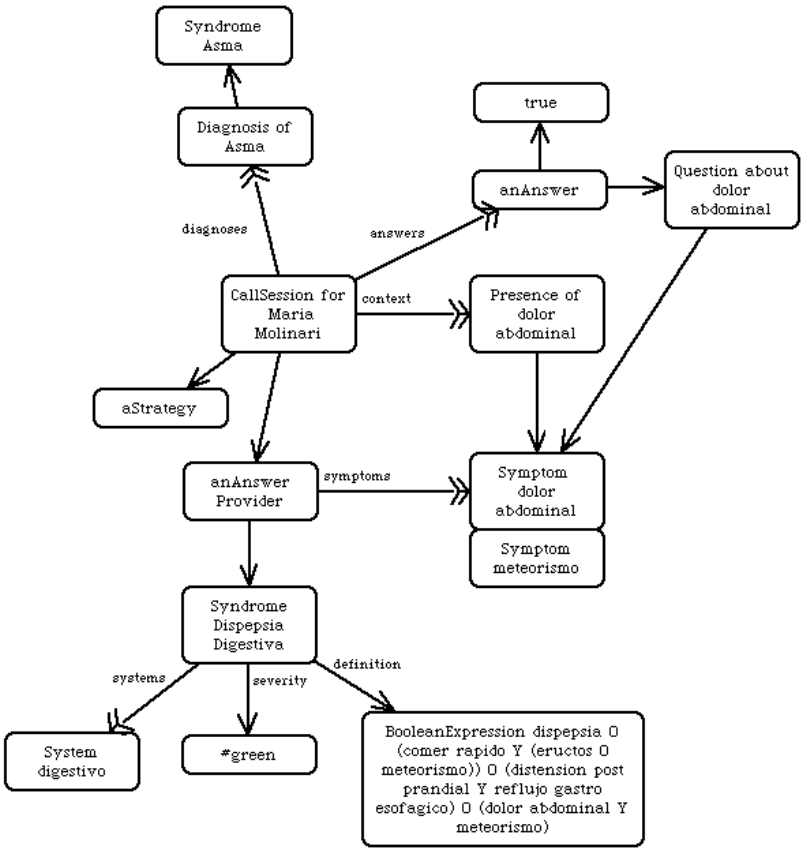
La sesión de diagnóstico aglutina todos estos objetos y opera con ellos, para representar la interacción, el diálogo y el intercambio de preguntas y respuestas entre el operador y el paciente. A la vez, la sesión de ExpertCare va proporcionando al operador información sobre los diagnósticos y la evidencia que los apoya, en la forma de ordenamiento por puntajes en la lista de síndromes.

La sesión con la que trabajamos aquí, sin embargo, no cumple con todas esas funciones, sino que incorpora algunos objetos de automatización (de los que hablaremos en la próxima sección) para simular al paciente y al operador, con el fin de poder probar el funcionamiento de distintas estrategias para dirigir el interrogatorio.

[5] En realidad, nunca se descarta nada completamente en *ExpertCare*, ya que cada nueva respuesta hace que se arme un nuevo contexto y se reevalúe completamente toda la situación. Así, un diagnóstico podría ser falso en un momento, y ante nueva información volver a un estado intermedio o resultar confirmado.



El siguiente diagrama de instancias nos permite mostrar las formas en que se relacionan estos elementos:



Auxiliares de Automatización

En el diagrama de objetos ya pudimos ver que, además de los objetos ya mencionados, la sesión cuenta con un `AnswerProvider` y una `Strategy`. Ahora explicaremos qué papel cumplen los mismos en nuestro modelo.

El `AnswerProvider` es un objeto que simula ser un paciente, el interlocutor de la sesión. Se lo inicializa con un síndrome y un conjunto de síntomas (se elige, del conjunto de los *satisfiers* de la definición del síndrome, uno en particular) y su responsabilidad principal es responder preguntas, mediante el protocolo `#answerTo: aQuestion`.

Por supuesto, es un interlocutor exageradamente simplificado, que omite muchas complicaciones que se presentan en una sesión real, hablando con una persona. El `AnswerProvider` nunca vuelve atrás sobre sus respuestas, nunca se contradice ni se equivoca, no puede entender mal lo que se pregunta ni desconocer la respuesta, ni tiene una psique que le haga sesgar las respuestas. Muchas de estas complicaciones escapan al alcance de este trabajo; algunas de ellas hemos atacado posteriormente y otras quedan anotadas en el *Future Work*.

El criterio del `answerProvider` para responder una pregunta es verificar si el síntoma de la pregunta se corresponde con alguno de los "suyos", y en ese caso da la respuesta preprogramada que confirma que presenta ese síntoma, dando una respuesta negativa en caso contrario.

Hay casos en los que un síntoma se encuentra negado en la definición del síndrome, por ejemplo ("vómitos" Y NO "fiebre alta"). En estos casos el `answerProvider` actúa dando la respuesta adecuada, esto es, negativa para "fiebre alta".

Otro caso especial es el de los síntomas cuantificados, donde como ya dijimos, los *satisfiers* se expresan como un rango de valores aceptados. En estos casos el `answerProvider` responde con un valor dentro del rango, elegido en el momento de su inicialización, si el síntoma es uno de los propios. Pero para el caso en que no lo es, dar una respuesta negativa implicaría saber cuál es el rango de valores a que apunta la pregunta. Como esa información no está disponible, adoptamos una convención entre el `AnswerProvider` y la sesión, respondiendo un número que se interpreta siempre como fuera del rango. En este caso, pudimos usar un número negativo, ya que todos los valores utilizados en las definiciones son positivos. Esta convención queda expresada en el método `AnswerProvider>>noQuantity`.

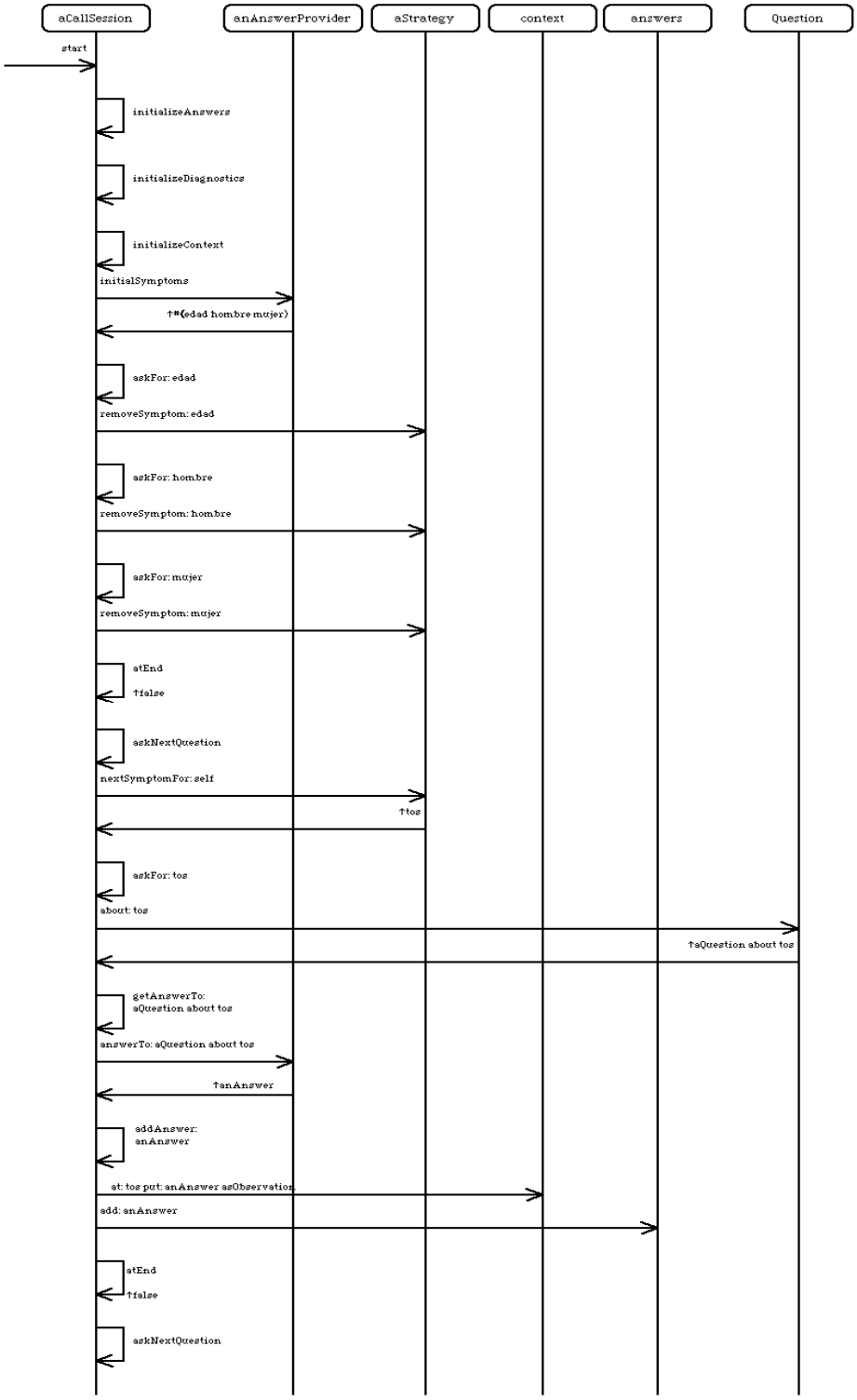
La última clase que nos falta ver de este conjunto que compone la parte básica de nuestro framework, es la que condensa el objetivo del presente trabajo también: la estrategia que dirige el interrogatorio de una sesión.

Usamos una clase abstracta, llamada simplemente `Strategy`, para definir el protocolo que se usa para la comunicación y colaboración entre la estrategia y la sesión. El mensaje principal para esto es `CallSession>>nextSymptomFor: aCallSession`. Este se define en la clase abstracta como *método abstracto* [GHJV/95], quedando a cargo de cada una de las subclases concretas dar una implementación.

El protocolo se completa con `#atEnd`, para que la estrategia pueda indicar que terminó su trabajo, `#symptoms`, `#removeSymptom:` y `#symptoms:` para manejar internamente la colección de síntomas sobre los que la estrategia puede preguntar, que se mantiene en la variable de instancia llamada `symptoms`.

Comunicación entre los elementos del framework

Una vez configurada la sesión con su *answerProvider* y su *strategy*, se le envía el mensaje `#start`, que inicia el ciclo de preguntas y respuestas hasta llegar a una condición de finalización.



En el diagrama podemos verificar que la comunicación principal de la sesión con la *strategy* es en la pregunta `#nextSymptomFor: self`, aunque también le indica `#removeSymptom:`(que no se ve en el diagrama, para no agregar más nivel de detalle), para sacarlo de la colección de síntomas que se pueden preguntar. Otro mensaje, que no se ve, es en el `#atEnd`, donde también se pregunta si la estrategia terminó.

La comunicación principal con el *answerProvider* es en el momento de responder la pregunta, en el método `#getAnswerTo: aQuestion` se le envía `#answer: aQuestion`. Pero hay otra comunicación inicial, para pedirle los `#initialSymptoms`. Esto representa la información con la que se inicia la sesión, datos básicos como el nombre del paciente, el sexo o la edad. Algunos de estos datos son síntomas y participan en las definiciones de algunos síndromes, por ejemplo, ciertas enfermedades son únicamente femeninas. Esto se resuelve en la sesión de ExpertCare completando un formulario al inicio de la sesión, pero aquí lo simulamos con los `#initialSymptoms` del *answerProvider*. Más adelante agregamos también uno o dos síntomas, ya que el paciente que llama siempre comienza mencionando algún síntoma.

En la etapa de inicio de la sesión, se inicializan tres cosas: *answers* con una colección vacía, *context* con un diccionario vacío y *diagnoses* de una manera especial. La sesión genera un diagnóstico por cada síndrome, indicando su propio contexto como contexto de evaluación del diagnóstico. Cabe aclarar que la separación entre *answers* y *context*, nuevamente, se debe a una adopción temprana de características que resultaban relevantes en la implementación de ExpertCare, pero que no reportan ningún beneficio en nuestro modelo, donde resultaron una redundancia inútil (que no nos molestamos tampoco en remover).

El ciclo que se ve en el diagrama continúa hasta que la sesión responde `#atEnd` con *true*, indicando que se cumple alguna de las condiciones del criterio de parada. Analicemos pues este criterio, a través de la implementación de `#atEnd`.

```
CallSession>>atEnd
    | full candidates |
    strategy atEnd ifTrue: [^true].
    full := diagnoses select: [:d | d isFull].
    full isEmpty ifTrue: [^false].
    (full anySatisfy: [:d | d isCritic]) ifTrue:
[^true].
    candidates := diagnoses
    select: [:d | d isCritic and: [d isOpen]].
    candidates isEmpty ifTrue: [^true].
    ^(self shouldContinueWith: candidates) not
```

Lo primero que se chequea es, como ya dijimos, si la estrategia indica que se terminó. Eso querría decir que no tiene más síntomas que tenga sentido preguntar, por lo que la sesión se termina.

Luego se revisan los diagnósticos, seleccionando los que estén completamente validados (*isFull*) en el contexto de la sesión. Si no hay ninguno, quiere decir que ningún diagnóstico está confirmado y no tenemos suficiente información como para concluir nada interesante, por lo que se retorna *false* para que la sesión continúe.

El siguiente paso es ver si alguno de los diagnósticos confirmados por las respuestas del paciente, corresponde a una situación de urgencia o emergencia (*isCritic*). Si

hay alguno en estas condiciones, la sesión se termina inmediatamente indicando un diagnóstico presuntivo de urgencia. Esto podría resultar prematuro ya que posterior información podría indicar con mayor certeza otro diagnóstico, pero es un criterio que refleja el riesgo inaceptable de dejar sin atención una emergencia.

Llegado a este punto, el código descartó varias situaciones y conviene repasar en qué condiciones estamos: tenemos algunos diagnósticos completamente confirmados (de lo contrario, la pregunta de `full isEmpty` nos hubiera sacado), pero ninguno de ellos es crítico.

Seleccionamos entonces un nuevo grupo de interés para analizar: los diagnósticos de síndromes críticos, que no están confirmados pero tampoco descartados (`d isCritic and: [d isOpen]`). Si no hay ninguno en estas condiciones, quiere decir que hemos descartado todos los posibles diagnósticos de urgencias, y podemos terminar la sesión. Recordemos que el objetivo primario es determinar si hay una emergencia o no; una vez que se decide que no, se da por terminada la sesión y el operador puede escoger un diagnóstico presuntivo sin necesidad de más preguntas. Cabe aclarar que sabemos que hay algún diagnóstico completo porque, como indicamos en el párrafo anterior, estamos en el caso en que hay diagnósticos confirmados pero ninguno es crítico.

Nos queda la última línea para analizar el caso restante: tenemos en `candidates` un grupo de diagnósticos de síndromes críticos que nuevas preguntas podrían confirmar. Vemos entonces la implementación de `#shouldContinueWith: candidates` para ver cómo se realiza esa decisión no trivial.

```
CallSession>>shouldContinueWith: candidates
    | full |
    full := diagnoses select: [:d | d isFull].
    candidates do: [:d | | syn |
      syn := d syndrome.
      full do: [:diag | (syn canCompleteFrom: diag
syndrome) ifTrue: [^true]]].
    ^false
```

Nuevamente seleccionamos en full los diagnósticos confirmados (sean o no críticos) y recorremos los candidatos (que son críticos), verificando si alguno puede completarse a partir de los diagnósticos confirmados. Decimos que un síndrome puede completarse a partir de otro, si hay una intersección no vacía entre sus *satisfiers*. Por lo tanto, esto es equivalente a verificar si hay síndromes críticos que estén abiertos y puedan completarse a partir de los síntomas observados. Si esto ocurre, conviene continuar la sesión para intentar confirmar o descartar al menos a ese candidato crítico, por lo que devolvemos *true* (y el `#atEnd` devolverá *false*).

Por el contrario, si ninguno de los candidatos se puede completar a partir de los diagnósticos confirmados, devolvemos *false* y terminamos la sesión. Esto es un poco más difícil de explicar, pero se basa en que para llegar a uno de esos síndromes críticos, se necesitaría un nuevo grupo de síntomas completamente separado del que ya tenemos (por la condición de que sus *satisfiers* tendrían que ser completamente disjuntos con el conjunto de síntomas observados), y eso es muy poco probable[6].

[6] Esta es una conclusión que no podemos afirmar con toda generalidad, aunque sí tenemos fundamento para sostenerla en este dominio y con la base de conocimiento estudiada.

Para llevarlo a un ejemplo de la vida real, sería como tener información de vómitos y continuar la sesión para explorar la posibilidad de una enfermedad respiratoria.

Primeras estrategias

Contando ya con los elementos básicos del framework, pudimos implementar algunas estrategias elementales y hacer las primeras pruebas. Esto nos permitió revisar los detalles de construcción, hacer una primera depuración del código y empezar a ganar conocimiento sobre la problemática que queremos resolver.

Recordemos que el objetivo principal de las estrategias es elegir síntomas para preguntar de tal manera que se minimice la cantidad de preguntas necesaria para poder cerrar la sesión con los criterios ya expuestos.

Definida esta métrica principal, tenemos también un objetivo definido por la práctica: deberíamos lograr que los síndromes de severidad `red` se decidan con menos de 3 o 4 preguntas, los de `yellow` con 4 o 5 y tenemos más margen con los síndromes de severidad `green`, donde el objetivo sería 6 o 7 pero se puede extender hasta 12.

Empezamos con las estrategias más simples que pudimos pensar, que prácticamente no utilizan información del contexto más que para no preguntar dos veces el mismo síntoma. Tampoco utilizan información de los síndromes en los que aparecen los síntomas, sino que consideran cada síntoma como elemento independiente del resto y con el mismo valor de información.

SequentialStrategy

Toma cada vez el siguiente síntoma de la colección ordenada (por nombre). Esto se logra implementando sólo dos mensajes:

```
SequentialStrategy>>initialize
    super initialize.
    symptoms := symptoms sortBy: #name
```

```
SequentialStrategy>>nextSymptomFor: aCallSession
    | symptom |
    symptom := symptoms first.
    symptoms remove: symptom.
    ^symptom
```

En la inicialización, se ordena por nombre la colección de síntomas. Esto da un orden arbitrario, ya que el nombre no tiene nada de particular en relación a lo que el síntoma es ni al papel que desempeña en las definiciones de los síndromes. Pero es un orden cómodo y repetible.

Luego, cada vez que se invoca `#nextSymptomFor:`, simplemente toma el primero de la lista de síntomas y lo saca de la misma para no volver a preguntarlo. Recordemos que éste es el mensaje que utiliza cíclicamente la sesión para que la estrategia le indique la próxima pregunta.

RandomStrategy

Toma cada vez un síntoma al azar entre los que aún no fueron preguntados. Su implementación es tan simple como la de la clase anterior:

```
RandomStrategy>>nextSymptomFor: aCallSession
    | symptom |
    symptom := symptoms atRandom.
    symptoms remove: symptom.
    ^symptom
```

Estas dos estrategias son “ciegas”, sirven únicamente para tener cotas de comparación. Las siguientes estrategias que planteamos incorporan algún criterio, aunque muy simple.

MoreSatisfiersStrategy

Considera todos los síndromes que todavía no están completos, toma los conjuntos de síntomas que hacen verdadera su definición (*satisfiers*) y elige el síntoma no preguntado que aparezca más veces en ellos.

```
MoreSatisfiersStrategy>>nextSymptomFor: aCallSession
    | syndromes all symptom |
    syndromes := aCallSession diagnoses
      select: [:d | d isOpen]
      thenCollect: [:d | d syndrome].
    all := Bag new.
    syndromes do: [:syndrome |
      syndrome symptoms[7] do: [:sym |
       (symptoms includes: sym)
            ifTrue: [all add: sym]].
(*) symptom := all max: [:e | all occurrencesOf: e].
    symptoms remove: symptom.
    ^symptom
```

Esta implementación es un poco más compleja que las anteriores, operando con varias colecciones, aunque la idea es simple y es la expuesta en el párrafo anterior.

Primero se recogen en la colección `syndromes` los síndromes que corresponden a diagnósticos aún abiertos en la sesión (los que todavía no tienen suficiente información como para que su definición se resuelva a verdadero o falso).

Luego se inicializa un `bag` llamado `all`, para recoger todos los síntomas aún no preguntados de esos síndromes. Un `bag` es una colección similar a un conjunto, pero que permite tener múltiples ocurrencias del mismo elemento y decir cuántas veces aparece, con el mensaje `#occurrencesOf:`.

Se recorre la colección de síndromes y por cada uno, a su vez, se recorre la colección de síntomas, extraídos de los *satisfiers* de la definición. Si el síntoma está (aún) en la colección de la estrategia, quiere decir que es un síntoma candidato por el que se puede preguntar. En ese caso, se lo agrega a la colección `all`, que así se va completando.

Una vez completo este recorrido por los síndromes y sus síntomas, estamos listos para buscar el resultado: el máximo valor sobre `all`, de la cantidad de ocurrencias

[7] Esto es una simplificación del código real. El mensaje `Syndrome>>symptoms` devuelve un conjunto de observaciones que hacen verdadera la definición del síndrome y en el código se recolectan los síntomas a partir de esas observaciones.

de un elemento. O sea, el síntoma que más aparece en las definiciones de los síndromes candidatos.

LessSatisfiersStrategy

Igual a la anterior, sólo que toma el síntoma que aparezca menos veces.

La implementación es casi igual, sólo que en la línea marcada con (*), en lugar de buscar el máximo se busca el mínimo.

```
(*) symptom := all min: [:e | all occurrencesOf: e].
```

MiddleSatisfiersStrategy

Igual a las dos anteriores, pero toma un síntoma que esté en el medio, entre la máxima y mínima cantidad de apariciones.

La implementación es casi igual, pero reemplazando la línea marcada con (*) en las anteriores por un código que ordena la colección por cantidad de ocurrencias y luego busca el elemento de la posición media.

```
(*) sorted := all asSet asArray
      sortBy: [:e1 :e2 | (all occurrencesOf: e1) <=
(all occurrencesOf: e2)].
    symptom := sorted at: (sorted size // 2 max: 1).
```

Estas tres estrategias siguen ideas exageradamente simplificadas sobre el valor de la información que pueden aportar los síntomas.

MoreSatisfiers opera sobre una premisa que podríamos caracterizar como refutacionista: si un síntoma aparece muchas veces, saber que NO está presente aporta una información importante. Así, esta estrategia escoge el síntoma que encuentre con mayor cantidad de apariciones en el conjunto de candidatos para preguntarlo, con la "esperanza" de que el paciente no lo tenga y reducir de esa manera el conjunto de candidatos.

De un modo simétrico, *LessSatisfiers* se basa en una esperanza similar a la del apostador de carreras: elegir un candidato poco favorecido puede pagar más. Si un síntoma aparece pocas veces, y el paciente lo tiene, ya sólo queda por explorar un conjunto pequeño de candidatos. Por eso, elige el síntoma que menos aparezca entre los síndromes candidatos.

El trío se completó con *MiddleSatisfiers*, en un intento por ver qué pasaba si se considera que los dos extremos son malos y se buscan síntomas por el medio.

MoreCriticSeparationStrategy

Esta es una estrategia bastante elemental, pero difiere de las otras en que sí utiliza información de los atributos de los síndromes. Su implementación es un poco más elaborada que las anteriores:

```
MoreCriticSeparationStrategy>>
nextSymptomFor: aCallSession
    | syndromes symptom |
    symptoms isEmpty ifTrue: [^nil].
    syndromes := aCallSession diagnoses
      select: [:d | d isOpen]
      thenCollect: [:d | d syndrome].
```

```
    symptom := self
      nextSymptomFor: aCallSession
      symptoms: symptoms
      syndromes: syndromes.
    symptoms remove: symptom.
    ^symptom
```

Este método selecciona los síndromes de la sesión que no están completos ni cerrados, y luego invoca al método de decisión sobre ese conjunto.

```
MoreCriticSeparationStrategy>>
nextSymptomFor: aCallSession
symptoms: syms
syndromes: syndromes
    | all symptom critic nonCritic |
    syms isEmpty ifTrue: [^nil].
    all := Bag new.
    critic := Bag new.
    nonCritic := Bag new.
    syndromes do: [:s |
      s symptoms do: [:sym |
        (syms includes: sym) ifTrue: [
          all add: sym.
          s isCritic
            ifTrue: [critic add: sym]
            ifFalse: [nonCritic add: sym]]]].
    symptom := all
      max: [:e | ((critic occurrencesOf: e) -
(nonCritic occurrencesOf: e)) abs].
    ^symptom
```

Aquí se recorren los síndromes previamente seleccionados. Por cada uno de ellos se recorren los síntomas de su definición. Los que aún no han sido preguntados se agregan a las colecciones de críticos o no críticos, según la severidad del síndrome. Finalmente, se elige el síntoma que maximice la diferencia de tamaño entre los dos conjuntos. Se busca con esto encontrar al síntoma que mejor separa los críticos de los no críticos. Un caso ideal sería que un síntoma aparezca en todos los síndromes críticos y ninguno de los no críticos, por ejemplo. Esta heurística busca un síntoma con buen poder de separación.

Cabe aclarar que estas estrategias fueron implementadas en primer lugar por su sencillez, con la idea de usarlas para probar y depurar el framework y una forma de trabajar con las estrategias, y no con la expectativa de que fueran efectivas. Era claro desde un principio que estos enfoques extremos y con muy poco uso de la información no podían dar un buen resultado. Sin embargo, pueden servir como valores de referencia de malos casos.

Primeros ensayos

En esta parte del trabajo tuvimos un concepto guía: el *laboratorio virtual*. Contando con poco conocimiento de la problemática real y de las posibles soluciones, quisimos construir herramientas que nos permitieran explorar la base de conocimientos, analizarla, ensayar distintas cosas y poder comparar fácilmente resultados.

Una práctica normal en estos casos es construir *scripts*, pequeñas piezas de código (generalmente sueltas) que construyen casos de prueba, hacen corridas de lo que se quiere probar y generan archivos de salida que luego se analizan.

Pero trabajar con scripts genera a su vez varios problemas. Por su misma naturaleza son poco estructurados, y generan una mecánica de trabajo desorganizada, poco documentada y donde es difícil reproducir resultados un tiempo después. Guiados por experiencias previas, intentamos organizar un poco más la forma de trabajo aprovechando el framework de SUnit [SUNIT].

SUnit es un framework para construir, organizar y correr repetidas veces un gran número de tests de código unitarios. Introducido como parte fundamental de la metodología ágil eXtremeProgramming (XP), fue portado con éxito a diferentes lenguajes y ambientes de trabajo, aunque la versión original fue SUnit (donde la S es de Smalltalk).

Nuestro propósito no era realmente hacer tests unitarios de código, pero sí realizar gran número de ensayos, cada uno de los cuales podría considerarse unitario. Por otra parte, ya Andrés Valloud mostró en [VAL07] cómo se podían utilizar variantes de SUnit para tareas similares: validación de invariantes de representación y *benchmarking* o comparación de resultados de pruebas. Esto último es muy similar a las pruebas que queríamos desarrollar.

Así construimos una clase de test llamada DiagnosticSessionTest y empezamos a escribir pequeños tests. Mostraremos algunos a modo de ejemplo.

El primero fue éste:

```
testSessionSequential1
    self buildSample1; setup1.
    syndrome severity: #yellow.
    session strategy: SequentialStrategy new; start.
    self asserts1
```

El código del test queda muy sintético, y es porque separamos varias partes y tareas:

1) La construcción de la muestra en

```
buildSample1
    | sym1 sym2 sym3 |
    Symptom initializeObjects; readSample1.
    Syndrome initializeObjects; readSample1.
    Syndrome syndromes do: [:syn | | symptoms |
      symptoms := Symptom symptoms copy.
      sym1 := symptoms atRandom.
      symptoms remove: sym1.
      sym2 := symptoms atRandom.
      symptoms remove: sym2.
      sym3 := symptoms atRandom.
      syn
```

```
        definition: (Conjunction
          of: (Conjunction
            of: (Variable named: sym1 name)
            and: (Variable named: sym2 name))
          and: (Variable named: sym3 name))]
```

Se comienza reinicializando las colecciones de síntomas y síndromes, y leyendo muestras de cada uno, que eran simplemente listas de nombres de síntomas y síndromes tomadas de algunas páginas de Internet, sin relación especial entre sí. Las muestras eran de 123 síntomas y 180 síndromes.

Después de leer los nombres, comienza el verdadero trabajo de construcción, que es armar las definiciones de los síndromes, en términos de los síntomas.

Con el `#do:` se recorren los síndromes (recientemente leídos) y se construye la definición de cada uno escogiendo para ello tres síntomas al azar (`sym1`, `2` y `3`). Las siguientes líneas son correcciones a la muestra que escogen otro síntoma si encuentran una repetición, esto garantiza que tenemos tres síntomas distintos. En la última parte se construyen (con gran número de paréntesis) variables de expresión lógica para los síntomas y se arma una conjunción de las tres. Esta fue la primera y más simple de las pruebas, más adelante trabajamos con otras expresiones, eligiendo operadores al azar, etc.

2) La preparación de la corrida de prueba en

```
setup1
    | ap |
    syndrome := Syndrome syndromes atRandom.
    ap := AnswerProvider having: syndrome.
    session := CallSession new.
    session answerProvider: ap.
    symptoms := ap symptoms
```

Se elige, nuevamente al azar, alguno de los síndromes y se crea un `AnswerProvider` para que simule ser un paciente con ése síndrome. Luego, se arma una sesión y se le coloca el *answerProvider* recién construido. Se guardan en una variable de instancia los síntomas elegidos por el *answerProvider* entre los *satisfiers* de la definición del síndrome, para poder hacer comprobaciones posteriores al final de la prueba.

Separamos esto de la construcción de la muestra para poder variarlos independientemente. Aquí no importa cómo se haya armado la muestra, es simplemente elegir un síndrome y configurar la sesión de cierta manera.

3) Corrida de la prueba, en el mismo test:

```
syndrome severity: #yellow.
session strategy: SequentialStrategy new; start.
```

Se determina que la severidad del síndrome elegido sea `#yellow`, para simular que la situación de la prueba sea una urgencia. Esto podría hacer sido

parte de la preparación de la corrida de prueba, pero preferimos dejar ese detalle aquí, nuevamente, para poder variarlo con independencia del resto.

Se construye la estrategia, en este caso eso es simplemente instanciar la clase de la estrategia secuencial. Se la coloca como estrategia de la sesión y se arranca la prueba con `#start`.

4) Comprobación de resultados, en

```
asserts1
    | dic full o |
    self
      assert: session atEnd;
      assert: session diagnoses size == Syndrome
syndromes size
      description: 'There should be one diagnosis
for each syndrome';
      assert: session answers size <= Symptom
symptoms size
      description: 'It should not have asked for
more symptoms'.
    dic := session context.
    self assert: dic size <= Symptom symptoms size.
    dic keysAndValuesDo: [:k :v |
      o := symptoms detect: [:obs | obs symptom = v
symptom] ifNone: nil.
      self
        assert: ((o isNil
          and: [(v value isBoolean and: [v value
not]) or: [v value < 0]])
          or: [o notNil and: [o isLike: v]])
        description: 'bad symptom value in
context'].
    full := session fullDiagnoses.
    self assert: full hasOneElement.
    full := full anyone.
    self assert: full syndrome == syndrome
description: 'Bad diagnosis'
```

Normalmente los tests unitarios culminan con una serie de aseveraciones (`asserts`) que describen y definen el estado final que uno espera encontrar al final de la corrida de prueba.

Si bien, como ya dijimos, no estamos haciendo un test unitario de código, encontramos útil esto para establecer y verificar varias propiedades.

Sin detenernos mucho en aspectos del código que no tienen interés, las descripciones que acompañan a cada aseveración indican la intención de la comprobación.

Así, indicamos primero que al terminar el test, la sesión debe estar finalizada (`atEnd`). Por construcción, la sesión debería tener tantos diagnósticos como síndromes había, y no puede haber más respuestas que la cantidad de síntomas, ya que el peor caso sería hacer una pregunta por cada síntoma.

Luego se hace una comprobación del contexto de la sesión, verificando por cada uno de sus elementos (un par síntoma-valor que indica si está presente o no) que sea uno de los síntomas elegidos por el answerProvider y tenga el valor que éste indica, o bien uno distinto y tiene valor falso.

Finalmente, se hacen algunas comprobaciones sobre lo que podríamos llamar la respuesta de la sesión, los síndromes que al terminar están completamente verificados. Como configuramos la sesión para tener una única urgencia, debería haber uno solo, y debería ser el que habíamos escogido en la etapa de configuración[8].

Todas estas afirmaciones parecen triviales, y normalmente en los tests unitarios pasa esto. Pero su propósito es comprobar que las cosas que el programador da por sentadas, al correr el código realmente se correspondan con las expectativas.

Con el mismo cuerpo de código de este test y variando únicamente la clase de estrategia a usar, construimos `testSessionRandom1`, `testSessionMoreSatisfiers1`, `testSessionLessSatisfiers1` y `testSessionMiddleSatisfiers1`.

Estos tests son una prueba mínima de funcionamiento, pero por ser una prueba única, de una sola corrida de la sesión, no aportan información sobre el desempeño de cada estrategia.

El siguiente paso natural era montar una iteración para elegir distintos síndromes y correr la sesión un gran número de veces, registrando cierta información de interés en cada corrida. Y así surgió un segundo grupo de tests básicos, que son los mismos que describimos recién, pero dentro de un ciclo:

```
testSessionSequential2
    | file |
    self buildSample1.
    file := File newFile:
'testSessionSequential2.txt'.
    [
      1 to: 1000 do: [:i |
        self setup1.
        syndrome severity: #yellow.
        session strategy: SequentialStrategy new;
start.
        self asserts1.
        self report1On: file line: i]]
      ensure: [file close]
```

Construimos la muestra fuera de la iteración, ya que no hay necesidad de variarla cada vez. Luego abrimos un archivo de texto, que será la salida donde registraremos la información de cada corrida. Luego se enmarca la iteración en un bloque que asegure el cierre del archivo al finalizar.

[8] Podría darse el caso, por la construcción al azar, de que otro síndrome distinto tuviera los mismos síntomas. Por el tamaño de las muestras esto era bastante improbable y no consideramos ese caso como interesante, ya que en una base de conocimientos real no debería darse.

La corrida realiza 1000 iteraciones del cuerpo del `testSessionSequential1`, con el agregado de `#report1On: file line: i`. Ese método es el que registra la información que nos interesa. En este caso, es sólo el número de iteración, el nombre del síndrome, la cantidad de preguntas que hizo la sesión antes de terminar y los nombres de los síntomas elegidos. Esto se coloca en el archivo de texto, separado por tabulaciones y poniendo cada iteración en una línea. Este formato fue elegido porque es muy sencillo y se puede leer inmediatamente con Excel, lo que nos da una manera muy rápida de hacer algunos análisis sobre estos archivos de resultados.

Del mismo modo y variando solamente la clase de estrategia, se construyeron `testSessionRandom2`, `testSessionMoreSatisfiers2`, `testSessionLessSatisfiers2` y `testSessionMiddleSatisfiers2`.

Sobre los archivos de salida y utilizando Excel, empezamos a hacer análisis estadísticos elementales para poder comparar la performance de las distintas estrategias, siempre considerando la cantidad de preguntas como el parámetro fundamental. De cada salida tomamos el total de preguntas realizadas a lo largo de las iteraciones, y el promedio por cada sesión. También analizamos máximo y mínimo, moda y mediana de las cantidades de preguntas, y el desvío estándar del parámetro para tener una idea de la dispersión de los resultados.

En los siguientes gráficos mostramos cómo se desempeñan las distintas estrategias a lo largo de varias pruebas similares.

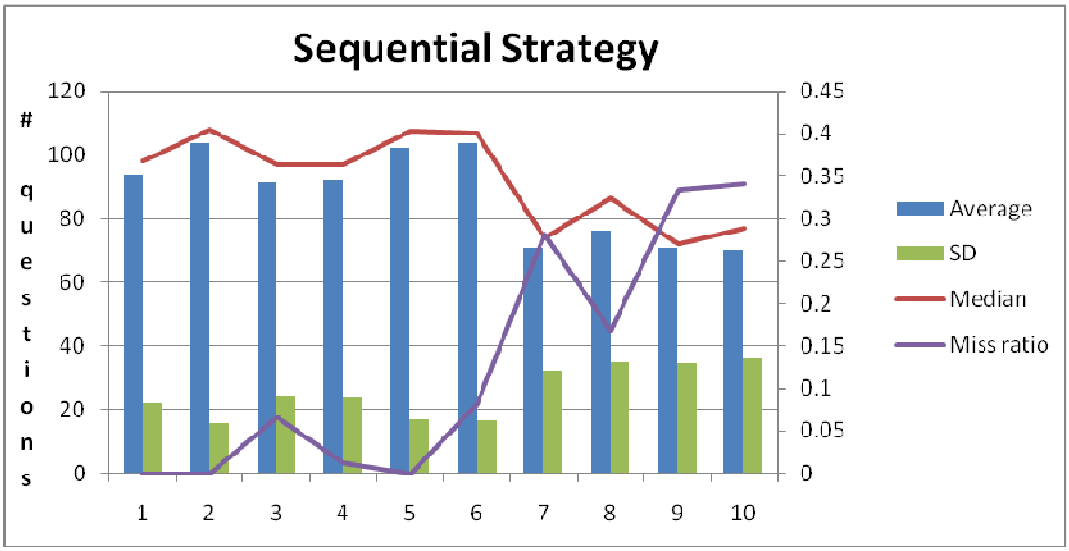
La que acabamos de describir corresponde a la primera columna. La segunda serie de pruebas es muy similar pero utilizando en las definiciones conjunciones de 5 síntomas en lugar de tres.

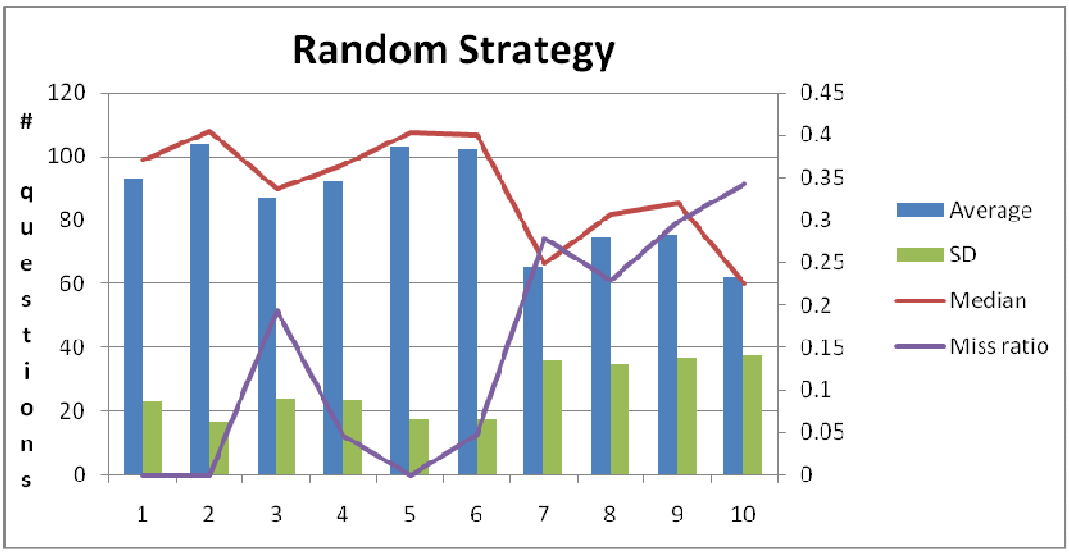
Decidimos luego incorporar negaciones y disyunciones, variando ligeramente la construcción de las definiciones en base a los tests ya construidos.

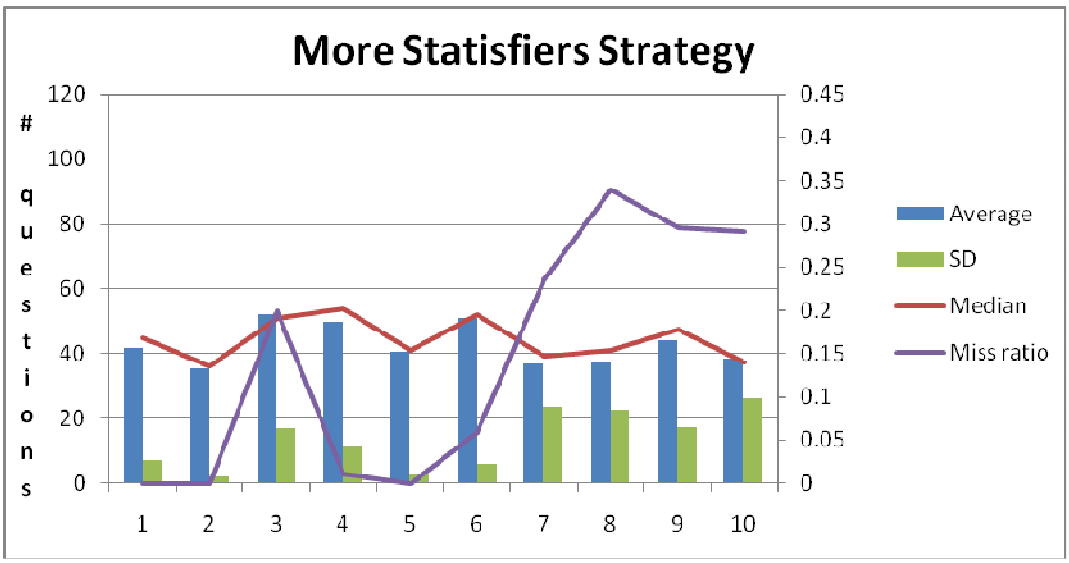
En las series 3 y 4, las definiciones se arman como conjunción de tres síntomas, pero con una probabilidad de 0.3 primero y 0.125 después (hicimos dos series de corridas), un síntoma se niega. Es decir, de la definición original de *A* ^ *B* ^ *C* podemos terminar en *A* ^ *no B* ^ *C*, o en *no A* ^ *B* ^ *no C*, o cualquier combinación excluyendo explícitamente la negación de los tres, que por la construcción del `AnswerProvider`, quedaría como verdadera desde el primer momento. Las probabilidades de 0.3 o 0.125 fueron elegida arbitrariamente, para reflejar que esperamos una cierta cantidad de negaciones pero no que sean mayoritarias. Tomamos un valor bastante alto y otro más moderado para intentar ver cómo variaba la respuesta de las estrategias.

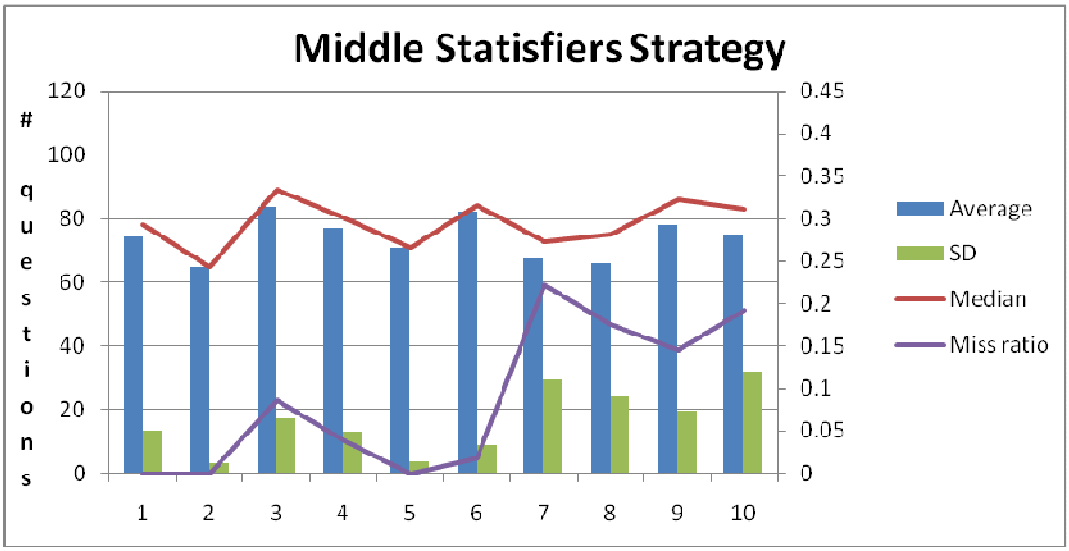
Las series 5 y 6 son equivalentes a las 3 y 4 pero con definiciones de 5 síntomas en lugar de tres.

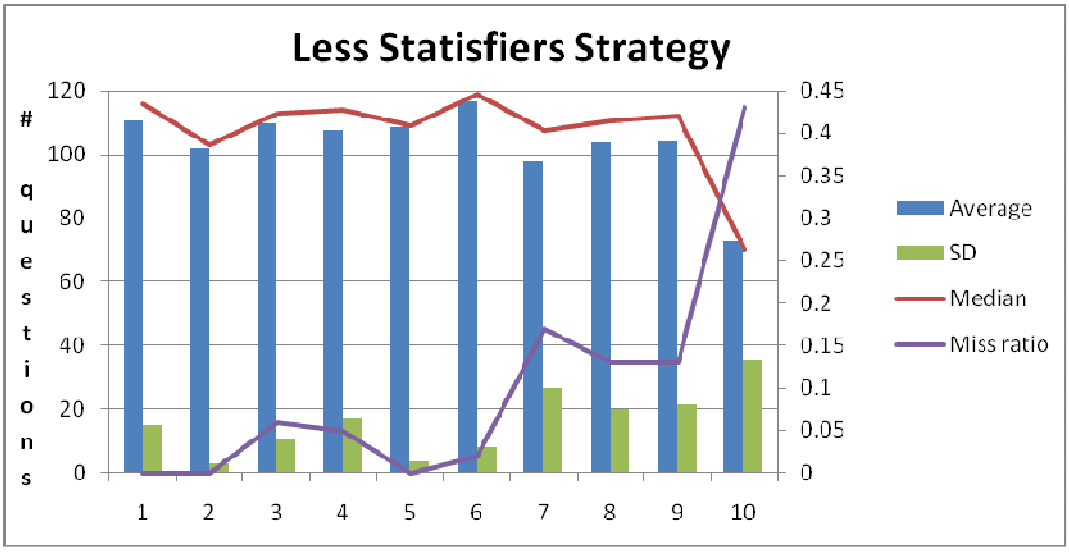
Las series 7, 8, 9 y 10 son análogas a las descriptas, pero afectando la probabilidad de convertir una conjunción en disyunción en lugar de negar el síntoma. Es decir, de *A* ^ *B* ^ *C* podemos pasar a *A* V *B* ^ *C* o bien a *A* ^ *B* V *C* o a *A* V *B* V *C*. Las series 7 y 8 son con definiciones de tres síntomas y las 9 y 10, de cinco. En todos los casos se hicieron corridas con los mismos valores de probabilidad de 0.3 y 0.125.

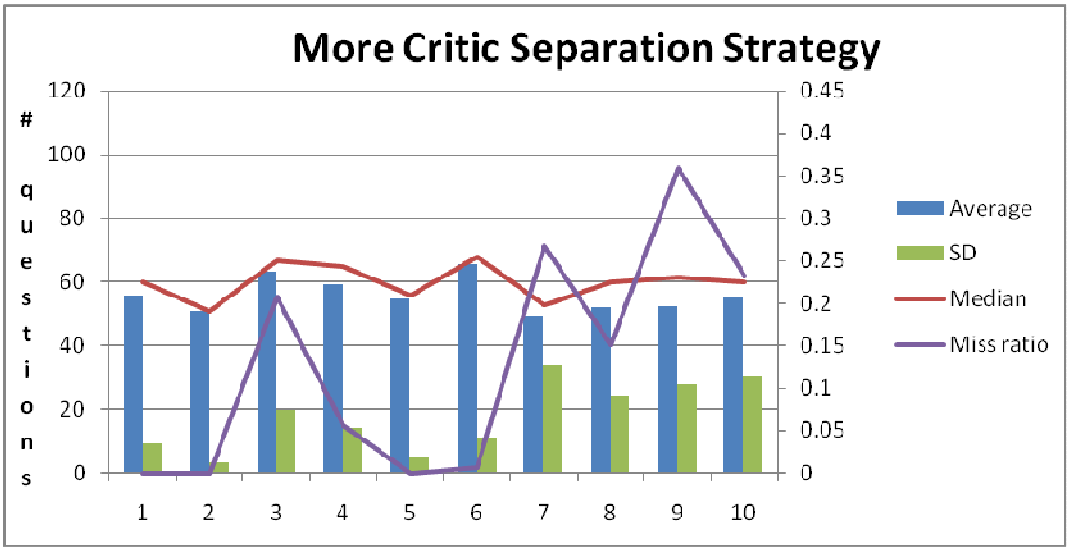

Sequential Strategy
questions
120
100
80
60
40
20
0
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1 2 3 4 5 6 7 8 9 10
Average
SD
Median
Miss ratio

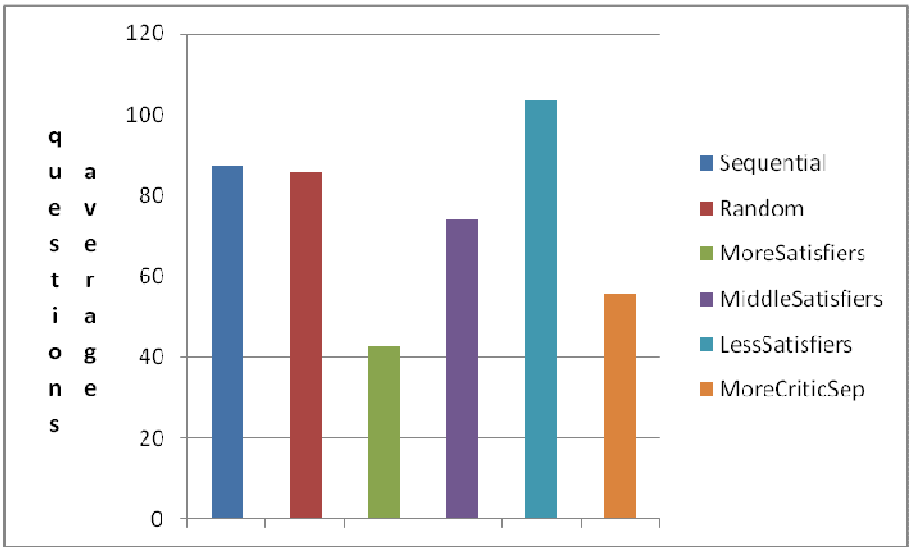

Random Strategy
questions
120
100
80
60
40
20
0
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1 2 3 4 5 6 7 8 9 10
Average
SD
Median
Miss ratio


More Statisfiers Strategy
questions
120
100
80
60
40
20
0
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1 2 3 4 5 6 7 8 9 10
Average
SD
Median
Miss ratio


Middle Statisfiers Strategy
questions
120
100
80
60
40
20
0
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1 2 3 4 5 6 7 8 9 10
Average
SD
Median
Miss ratio


Less Statisfiers Strategy
questions
120
100
80
60
40
20
0
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1 2 3 4 5 6 7 8 9 10
Average
SD
Median
Miss ratio


More Critic Separation Strategy
questions
120
100
80
60
40
20
0
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1 2 3 4 5 6 7 8 9 10
Average
SD
Median
Miss ratio



Estas pruebas elementales sirvieron para empezar a dimensionar la tarea y las complicaciones a enfrentar.

Lo primero notable fue que, a pesar del pequeño tamaño y la simplicidad de las muestras elegidas, la complejidad supera un abordaje tan ingenuo como éste. Realizar mil iteraciones toma un tiempo considerable (entre 10 minutos y varias horas), al punto de que en la estrategia menos eficiente (`LessSatisfiersStrategy`) tuvimos que reducir la cantidad de pruebas de 1000 a 100, y de 1000 a 500 en las de `MoreSatisfiers` y `MiddleSatisfiers`. Las estrategias que hacen cálculos sobre los *satisfiers* tardan notablemente más que las que sólo hacen una elección sin cálculo, como `Sequential` y `Random`.

Estas pruebas parecen mostrar una ventaja considerable de la estrategia `MoreSatisfiers` frente a las otras, aunque las cantidades de preguntas son muy altas, en promedio más del 25% del total de los síntomas.

Al introducir la negación, aparecieron dos problemas que no existían con las definiciones planteadas sólo con conjunciones: errores y múltiples diagnósticos verdaderos. Estos problemas son interesantes porque también aparecen trabajando con las definiciones reales.

Llamamos error a que la sesión termine con un solo síndrome con su definición evaluando a verdadero, pero que ese síndrome no sea el elegido para la prueba. Diagnósticos múltiples es cuando la sesión termina con varios diagnósticos completos (puede estar o no incluido el elegido).

Cabe aclarar que este tipo de error no necesariamente es un verdadero error, sólo lo es con respecto a la prueba. Al estar armadas las definiciones al azar, podría ser que hayamos encontrado otro síndrome distinto del buscado, pero con la misma definición o bien con una definición implicada por la del buscado. De todos modos, el número indica una diferencia respecto de lo esperado que sería necesario revisar, sobre todo si es muy alto. Este índice es lo que reflejamos en la curva de *Miss ratio* en los gráficos.

Además de los comentarios previos, se ve una gran dispersión y poca estabilidad en los resultados, observándose valores altos de desvío estándar en todas las muestras.

La conclusión preliminar más importante es que ninguna de estas estrategias es viable, ya que aún en los mejores resultados el promedio o la mediana nunca bajaron de preguntar un 25% de los síntomas (más de treinta preguntas).

Otro resultado notable fue la alta tasa de errores. Esto reveló la extrema importancia de que las definiciones sean adecuadamente excluyentes cuando se arma la muestra y se decide qué respuestas dará el AnswerProvider, ya que la aparición de negaciones hace que responder inocentemente por la ausencia de un síntoma si no es del grupo elegido, posibilite que se hagan válidos diagnósticos en que ese síntoma aparece negado en la definición. La aparición de conjunciones, por otro lado, aumenta la cantidad de conjuntos de síntomas que satisfacen una definición y las posibilidades de superposición (que un mismo conjunto de síntomas haga verdadera más de una definición).

Lectura de archivos de ExpertCare

Esta sección describe un grupo de clases auxiliares que no representan conceptos del dominio, sino que atacan un problema de construcción del ambiente de prueba: la importación de la base de conocimiento definida en ExpertCare. El lector interesado en las heurísticas y el modelo de conocimiento puede saltarla sin perjuicio. Hemos incluido una sección para detallar esto por dos razones. Una es la completitud en la descripción del trabajo y las herramientas que construimos y utilizamos. La otra es marcar un punto de inflexión muy importante en la cronología del proyecto: a partir de este punto, no construimos más muestras artificiales de datos para las pruebas sino que nos concentramos en utilizar y explorar la base de conocimientos real, ya definida en el sistema ExpertCare.

Esta base fue exportada en dos archivos de texto con un formato simple: uno para los síntomas y otro para los síndromes y sus definiciones. Para leer y procesar esos archivos, poblando la representación de conocimiento que describimos en la sección "Representación de los conceptos básicos", construimos dos jerarquías de clases bastante sencillas: la de los `FileReaders` y las de los `FileRecords`. Además, tuvimos que implementar un *parser* elemental, para interpretar las expresiones lógicas de las definiciones.

Al principio utilizamos un formato codificado de los síntomas y síndromes, reemplazando los nombres por un número de identificación. Esto nos permitió hacer una primera etapa de trabajo sin vernos influidos por los nombres comunes y las asociaciones que invitan a usar el sentido común. Estos son ejemplos del formato de los dos archivos:

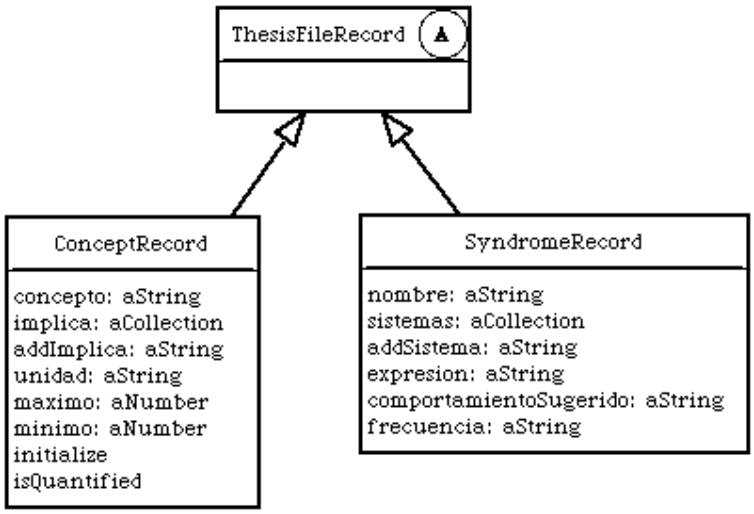
```
#CONCEPTO: 'ExistenceConcept 0112'
#IMPLICA: 'ExistenceConcept 0111'

#CONCEPTO: 'ExistenceConcept 0113'
#IMPLICA: 'ExistenceConcept 0059'

#CONCEPTO: 'ExistenceConcept 0114'
#IMPLICA:


#CONCEPTO: 'ExistenceConcept 0115'
#IMPLICA: 'ExistenceConcept 0059'

#CONCEPTO: 'QuantifiableConcept 0002'
#UNIDAD: semana
#MINIMO: 1 semana
#MAXIMO: 44 semanas

#CONCEPTO: 'ExistenceConcept 0116'
#IMPLICA: 'ExistenceConcept 0113'

#CONCEPTO: 'ExistenceConcept 0117'
#IMPLICA:
```

Muestra del archivo de síntomas

```
        #NOMBRE: 'Syndrome 0022'
        #EXPRESION: 'ExistenceConcept 0138' Y 'ExistenceConcept 0135' Y 'Exis-
tenceConcept 0136'
        #FRECUENCIA: media
        #COMPORTAMIENTOSUGERIDO: '24 horas' | Consultorio | Ninguno
        #SISTEMAS: 'SystemConcept 0032'

        #NOMBRE: 'Syndrome 0024'
        #EXPRESION: ('ExistenceConcept 0155' Y ('ExistenceConcept 0154' O
'ExistenceConcept 0151' O 'ExistenceConcept 0153' O 'ExistenceConcept 0158')) O
'ExistenceConcept 0157'
        #FRECUENCIA: baja
        #COMPORTAMIENTOSUGERIDO: '4 horas' | Hospital | Ninguno
        #SISTEMAS: 'SystemConcept 0003'

        #NOMBRE: 'Syndrome 0026'
        #EXPRESION: 'ExistenceConcept 0161' Y ('ExistenceConcept 0166' O 'Exis-
tenceConcept 0163')
        #FRECUENCIA: media
        #COMPORTAMIENTOSUGERIDO: '24 horas' | Consultorio | Ninguno
        #SISTEMAS: 'SystemConcept 0043'

        #NOMBRE: 'Syndrome 0027'
        #EXPRESION: (('QuantifiableConcept 0004' >= '180 mmhg') O ('Quantifia-
bleConcept 0003' >= '120 mmhg') O 'ExistenceConcept 0170') Y 'ExistenceConcept
0068'
        #FRECUENCIA: baja
        #COMPORTAMIENTOSUGERIDO: '15 minutos' | Hospital | 'Alta comple-
jidad con médico'
        #SISTEMAS: 'SystemConcept 0013'

        #NOMBRE: 'Syndrome 0028'
        #EXPRESION: 'ExistenceConcept 0177' Y ('ExistenceConcept 0178' O 'Exis-
tenceConcept 0006' O true)
        #FRECUENCIA: baja
        #COMPORTAMIENTOSUGERIDO: '30 minutos' | Hospital | 'Ambulancia
con médico'
        #SISTEMAS: 'SystemConcept 0024'
```

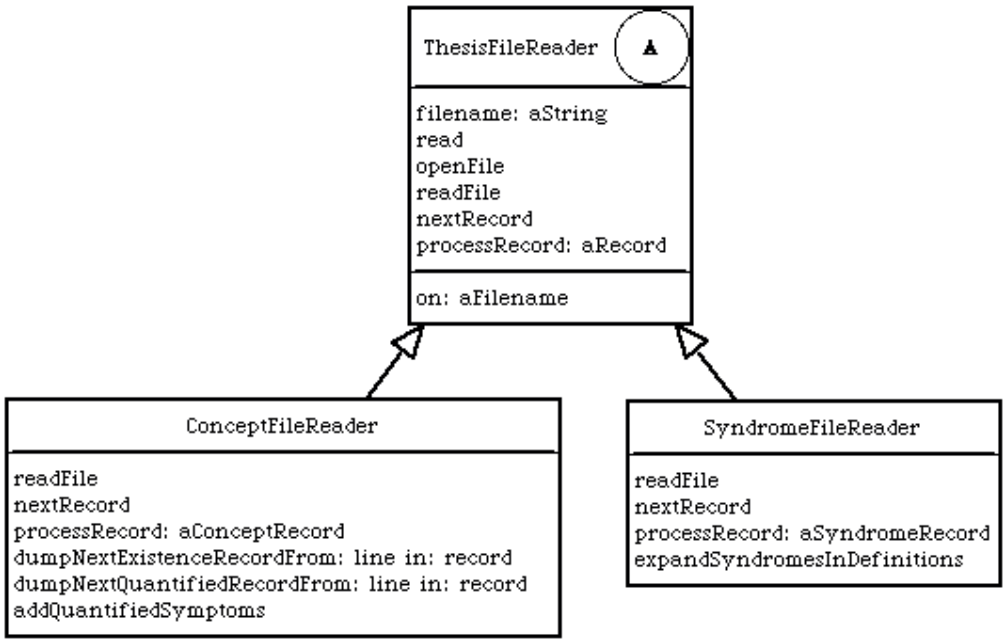
Muestra del archivo de síndromes

Estos formatos de archivo son los que dictaron la estructura de los `Records`, que sirven para mapear directamente lo que se lee del archivo en unidades intermedias, más cercanas a los conceptos del dominio que las líneas de texto.



Como se ve en el diagrama de clases y el protocolo, son poco más que una estructura de datos, un registro (como su nombre lo indica) que contiene de manera más o menos directa la información de los archivos. La clase abstracta `ThesisFileRecord` fue introducida sólo para agrupar las subclases y no tiene ningún comportamiento interesante.

El trabajo de llevar esta información al modelo lo hace la jerarquía de los `Readers`.



La clase abstracta `ThesisFileReader`, además de servir para agrupar las subclases, define el protocolo y la estructura básica para leer un archivo de texto formado por registros: apertura y cierre del archivo, con un ciclo de lectura de registros que continúa hasta el final el archivo.

```
ThesisFileReader>>read
    self openFile.
    [self readFile] ensure: [file close]
```

```
ThesisFileReader>> readFile
    [file atEnd] whileFalse: [self processRecord:
self nextRecord]
```

La implementación de `#openFile` depende de la clase `File` (o similares) y su relación con el manejo de archivos en el sistema operativo, y no reviste mayor interés. La variable de instancia `file` se usa para conservar un *stream* de lectura sobre el archivo de bajo nivel. La implementación de `#nextRecord` y `#processRecord`, a este nivel abstracto, es no hacer nada.

Al implementar la subclase `ConceptFileReader` se hizo visible una división entre los síntomas. El formato del archivo da dos tipos de registro distintos según el síntoma sea cuantificado o simplemente booleano (el paciente lo presenta o no lo presenta, sin graduación). Se puede ver en la muestra del archivo que los síntomas cuantificados no tienen implicaciones pero aportan alguna información adicional: la unidad de medida y un rango, especificando el máximo y mínimo.

Para llevar más fácilmente esta distinción al modelo, creamos una subclase de `Symptom` llamada `QuantifiedSymptom`, que agrega esa misma información en variables de instancia.

Los detalles de esta implementación pueden ser consultados y revisados en el Apéndice.

El `ThesisExpressionParser` es un parser recursivo descendente bastante elemental, que utilizamos para las expresiones lógicas de las definiciones de los síndromes. No vamos a entrar en detalles de su implementación porque escapa a la temática de la tesis y sigue modelos clásicos y bien conocidos. Sólo vamos a consignar someramente la gramática utilizada:

```
expression      -> term exprCont
term            -> $( expr $) | variable | number | <eos>
exprCont        -> op term | <eos>
variableName    -> $' name $'
name            -> <word> (<word>)*
number          -> $- <number> | <number>
op              -> $Y | $O | $NO | $< | $> | $= | $<= | $>= | $<>
```

Lo que se indica entre los signos <> es un tipo de *token*. El *token* `eos` es el *endOfStream*, el que indica el final del *string* que se está interpretando, donde no queda más *input* por leer. Usamos $ para indicar que lo que viene después es un solo carácter o *token*.

Estudio estadístico y agrupamientos

En este punto ya contábamos con la base de conocimiento de la aplicación, incluida en nuestro laboratorio virtual. Y para continuar con la metáfora del laboratorio, estábamos en las mejores condiciones para hacer una disección.

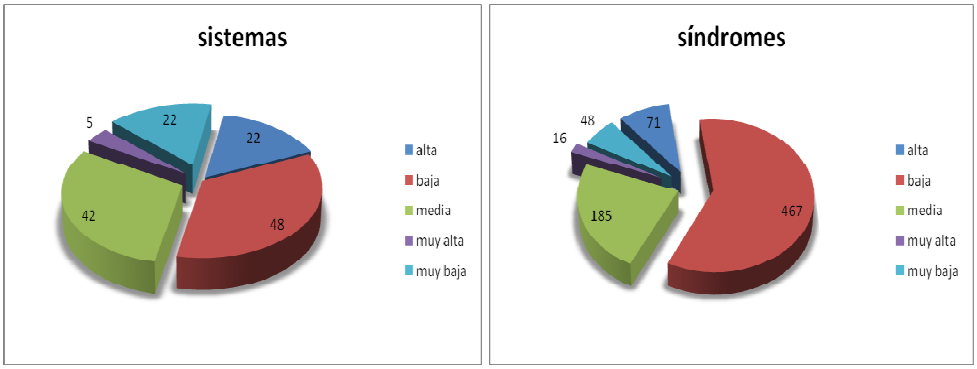
Comenzamos a pensar preguntas básicas sobre cantidades y formas de agrupamiento: ¿Cuántos sistemas hay? ¿Cuántos síntomas y síndromes? ¿Cuántos correspon-

den a urgencias? ¿Se distribuyen parejamente entre los sistemas? ¿En cuántos síndromes aparece un síntoma, en promedio? ¿Y como máximo?

Teníamos montones de preguntas como estas, cuyas respuestas podían guiarnos a imaginar estrategias. Las primeras las respondimos con *scripts* de consulta que navegaban la base recolectando información. Pero pronto fue evidente que las preguntas eran muchas y el número de *scripts* iba a crecer indefinida y desorganizadamente. Por otro lado, la navegación para responder distintas preguntas tenía muchas cosas en común y se repetía bastante código. Así que decidimos poner todas las cuestiones de navegación y agrupación o clasificación en una clase. En esta clase también dejaríamos métodos de consulta sobre la base, para responder a las preguntas más típicas e interesantes. Como su primera y principal finalidad es hacer un estudio estadístico de la base de conocimiento, llamamos a esta clase `StatisticalCollector`.

Cuenta con un protocolo de agrupamiento básico, cuyos métodos sirven para construir estructuras de diccionarios de uno, dos o tres niveles, según algún criterio. También tiene algunos métodos que se usan como *queries* para devolver algún grupo interesante. Como ejemplo de agrupamientos de un nivel tenemos `syndromesBySymptom`, que devuelve un diccionario donde las claves son los síntomas y los valores, las colecciones de síndromes en cuya definición aparece el síntoma; `systemsBySymptom`, donde análogamente las claves son los síntomas y los valores son los sistemas en cuyos síndromes aparece ese síntoma y `syndromesBySystem`, `syndromesBySeverity` y `syndromesByFrequency`, donde las claves son sistemas, severidades o frecuencias y los valores, las colecciones de síndromes con ese atributo. Como ejemplo de agrupamientos en dos niveles, tenemos `syndromesBySystemAndFrequency` y `syndromesBySystemAndSeverity`, donde las claves son los sistemas y los valores son diccionarios donde, a su vez, las claves son las frecuencias o las severidades y los valores, las colecciones de síndromes con ese atributo. Como protocolo de consulta, podemos mencionar `symptomsBySystemOfSeverity:` o `symptomsBySystemOfFrequency:`, que devuelven un diccionario de sistemas y síntomas, filtrando por una severidad o frecuencia.

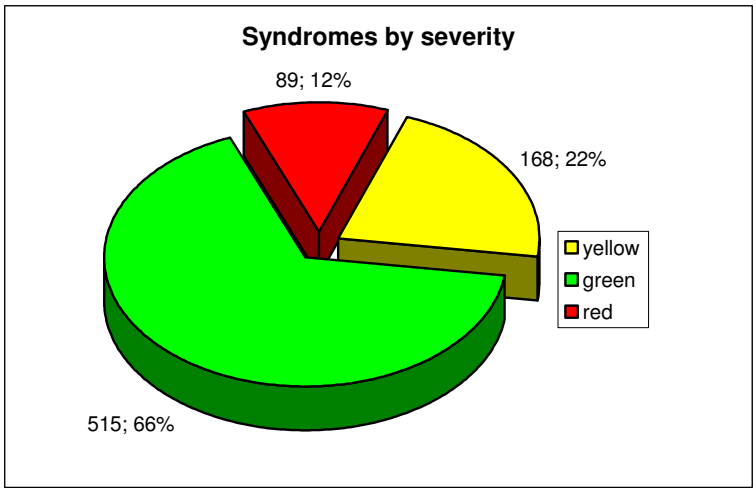
A partir de estos métodos básicos, sacamos varios reportes para estudiar características de la base y la distribución de los síntomas y síndromes según distintos criterios. Podemos ver gráficos con los resultados de algunos.



Esta es la distribución de sistemas y síndromes según la frecuencia. Podemos notar una fuerte concentración de los síndromes en la frecuencia baja, mientras que sólo hay 16 síndromes con frecuencia muy alta, de sólo 5 sistemas diferentes. También hay pocos síndromes y pocos sistemas de frecuencia muy baja.

Para poner en contexto estas cifras, veamos las cantidades brutas: hay 1119 síndromes (de los que sólo 772 son utilizados[9]), 1825 síntomas (de los que sólo 1267 son utilizados[10]), 48 sistemas, 3 severidades y 5 frecuencias distintas.

Casi todos los síndromes están adscriptos a un único sistema, hay solamente 15 que figuran en dos sistemas distintos.



Esta es la distribución de síndromes por severidad. Como era de esperarse, la mayor cantidad de síndromes corresponde a la severidad *green*, es decir, no requieren una atención inmediata. Constituyen los dos tercios del total de síndromes, lo que nos deja un porcentaje apreciable y una cantidad importante de síntomas en las otras criticidades. Los síndromes en situación de emergencia (criticidad *red*) son los menos, pero hay 89 de ellos.

Otra medida interesante es en cuántos sistemas aparece cada síntoma. Esto nos da una idea de cuán característico es cada síntoma de un sistema. Podemos ver que el agrupamiento por sistemas es bastante bueno: el 75% de los síntomas aparecen en síndromes de un solo sistema, el 15% en dos sistemas y sólo un 5% aparecen en 4 o más sistemas.

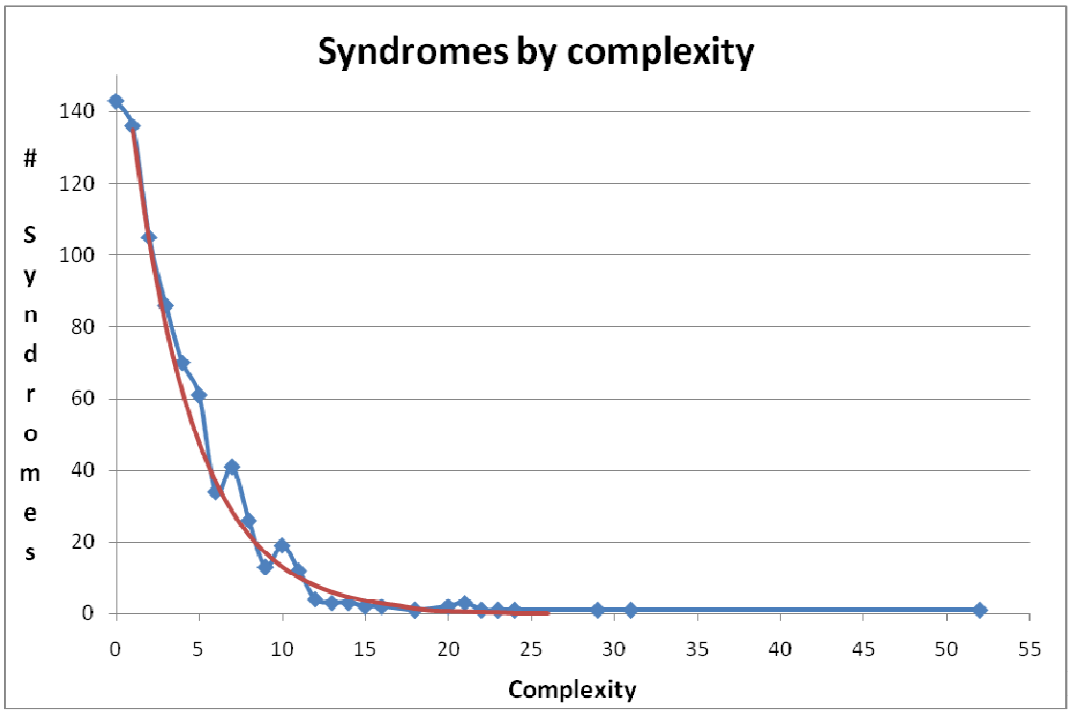
[9] Los demás síndromes están anulados poniendo **false** en sus definiciones, porque carecían de interés para la población a que se enfocan la base de conocimiento y la aplicación

[10] Los demás síntomas están anulados porque aparecen solamente en las definiciones de síndromes anulados, o no aparecen en ninguna definición.

# Sistemas	Síntomas	%
1	956	75.45
2	193	15.23
3	55	4.34
4	25	1.97
5	12	0.95
6	6	0.47
7	5	0.39
8	3	0.24
9	3	0.24
10	3	0.24
11	1	0.08
13	3	0.24
19	1	0.08
31	1	0.08

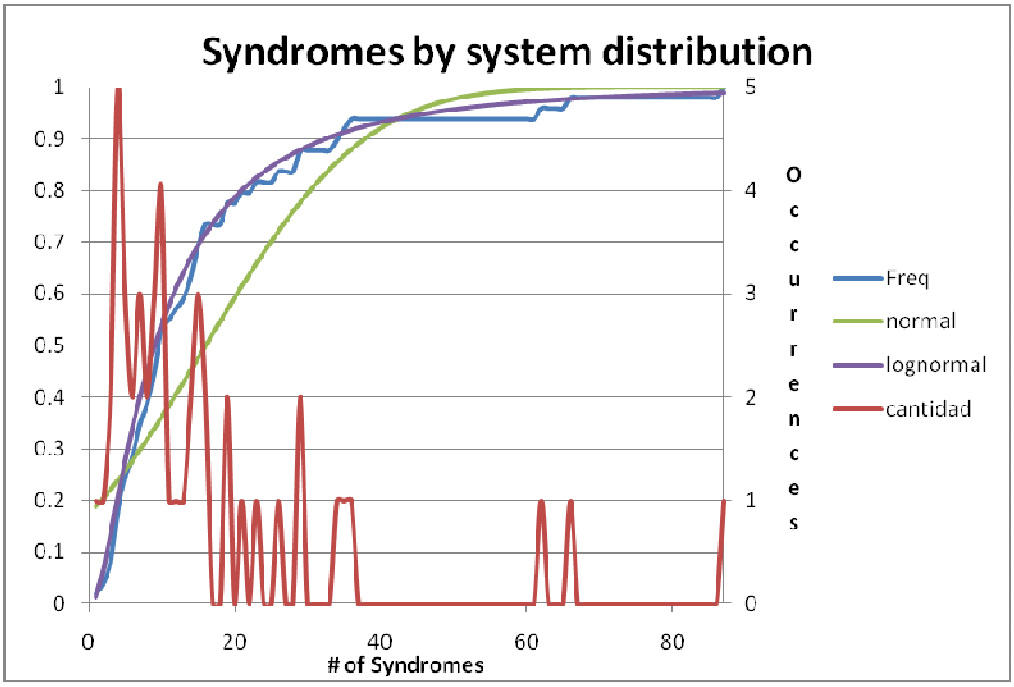
Nos preguntamos también acerca de la complejidad de las definiciones de síndromes. Para estudiarla, definimos una medida de complejidad muy simple: la complejidad de una expresión es la cantidad de operadores (lógicos o de comparación) que incluye. Así, la complejidad de una variable es 0, la negación suma uno a la complejidad de la expresión negada, las operaciones binarias (**and** y **or**) y las comparaciones son uno más la suma de las complejidades de sus sub-expresiones.

Analizando las definiciones de todos los síndromes y agrupándolos según la complejidad de su definición, obtuvimos la siguiente distribución:



Aquí podemos observar varias cosas. La primera y más inmediata es que la distribución se aproxima bastante bien con una curva exponencial, en este gráfico usamos $135 * e^{-0.25 * \text{complejidad}}$. Las mayores cantidades de síndromes corresponden a definiciones de complejidad 0 (un único síntoma/variable) o 2 (una conjunción o disyunción de dos síntomas), y sin embargo apenas representan un 36% del total. Junto con las de complejidad 3 alcanzan a un 50%, pero es necesario sumar hasta complejidad 8 para alcanzar el 90% del total de síndromes. Hay 57 síndromes con complejidad de 10 o más. Esto indica que, aunque la mayoría de las definiciones son bastante simples, hay un número significativo de síndromes con definiciones bastante complejas.

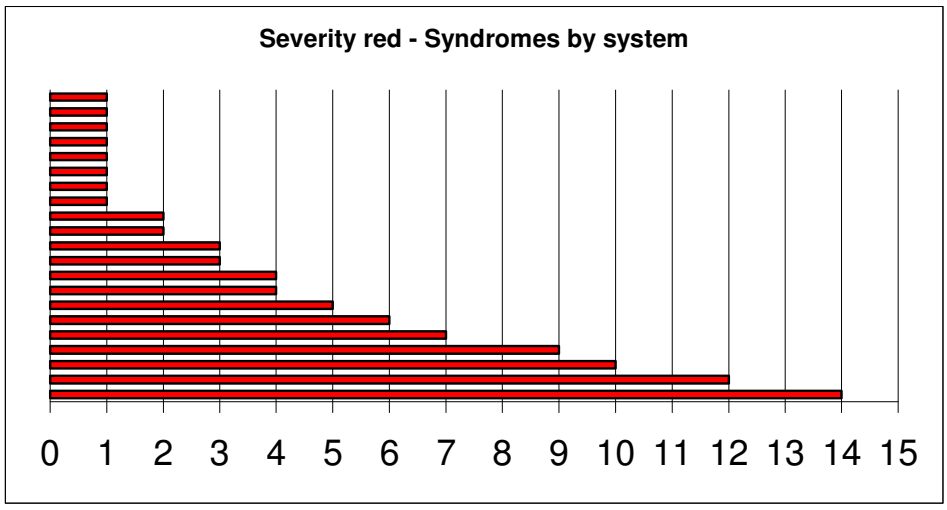
Veamos ahora cómo se distribuyen los síndromes entre los sistemas:

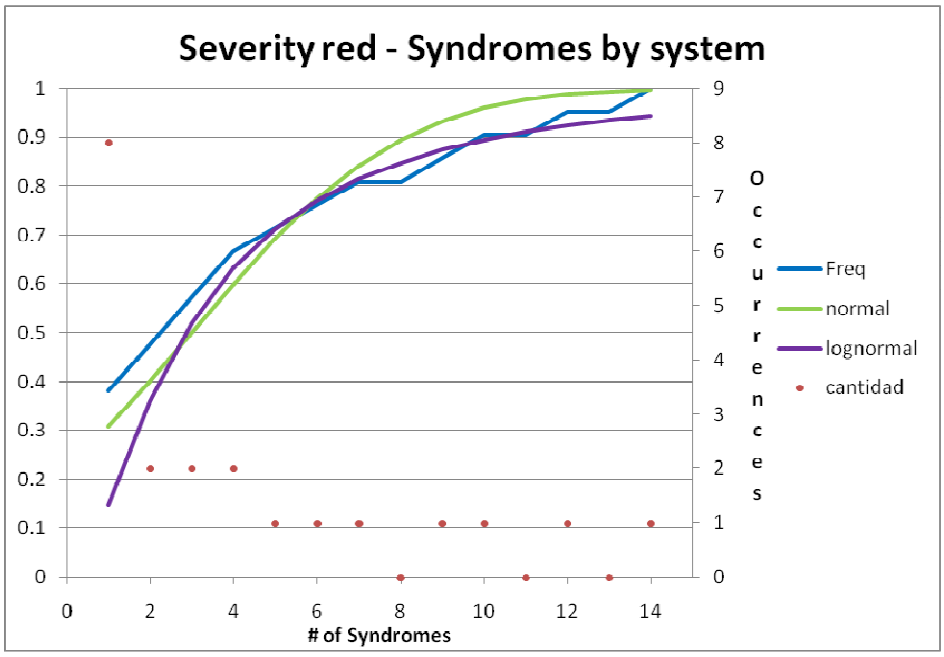


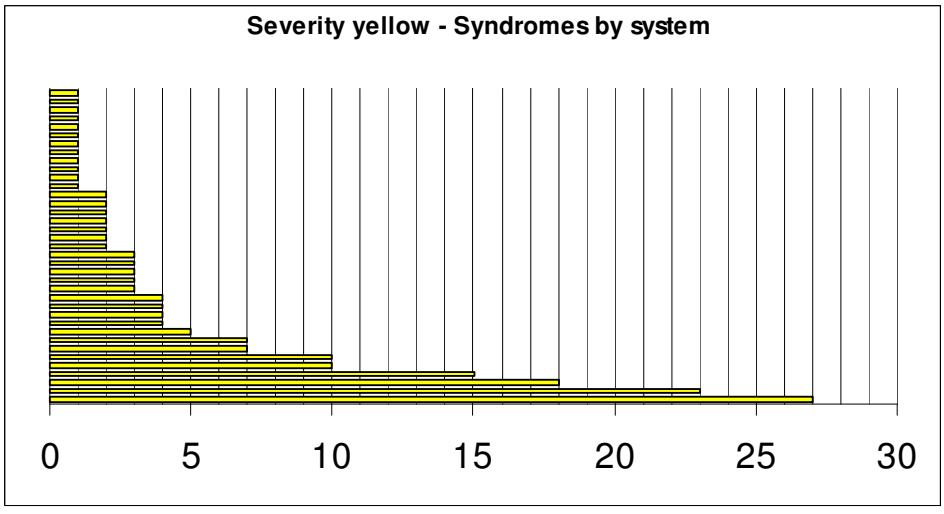
La distribución es bastante pareja y dispersa, no hay una gran concentración en ninguno de los sistemas. Un sólo sistema tiene un sólo síndrome, nueve de ellos tienen menos de cinco síndromes. Por el lado más alto, el sistema con más síndromes reúne apenas un 11% del total, los tres con mayor cantidad superan el 25% y hay que agrupar los nueve mayores (casi el 20%) para reunir el 50% de los síndromes. Estos caen de 62 a 36 entre el tercero y cuarto sistemas, pero luego las diferencias entre cada sistema y el siguiente se suavizan mucho.

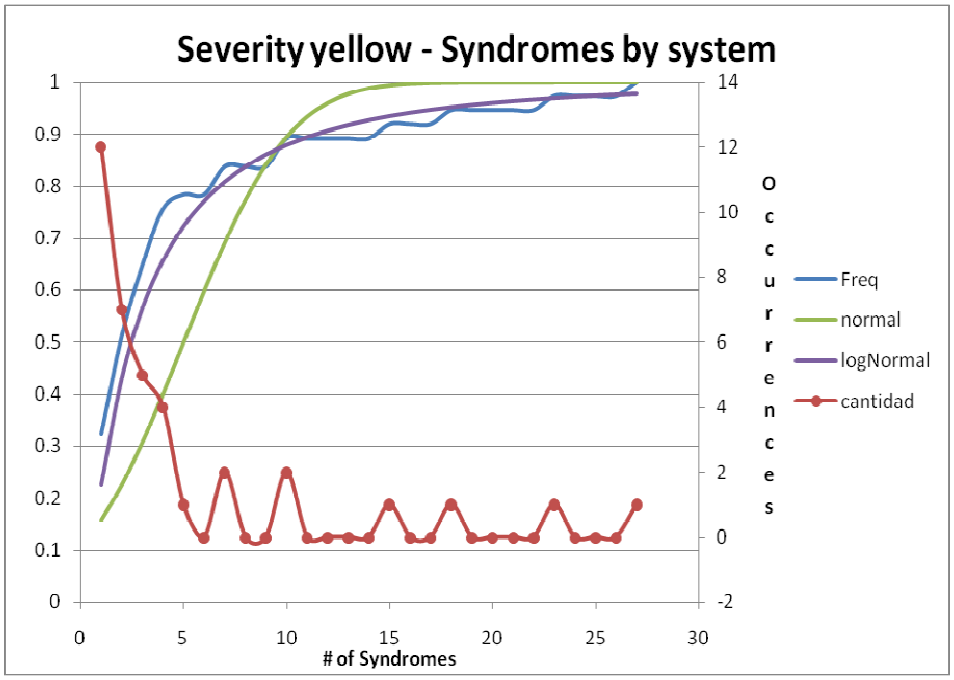
Intentando aproximar la distribución de las frecuencias con alguna distribución estadística conocida, no pudimos hacerlo con una normal pero sí se ajusta bastante bien con una logNormal de media 2 y desvío estándar 1.

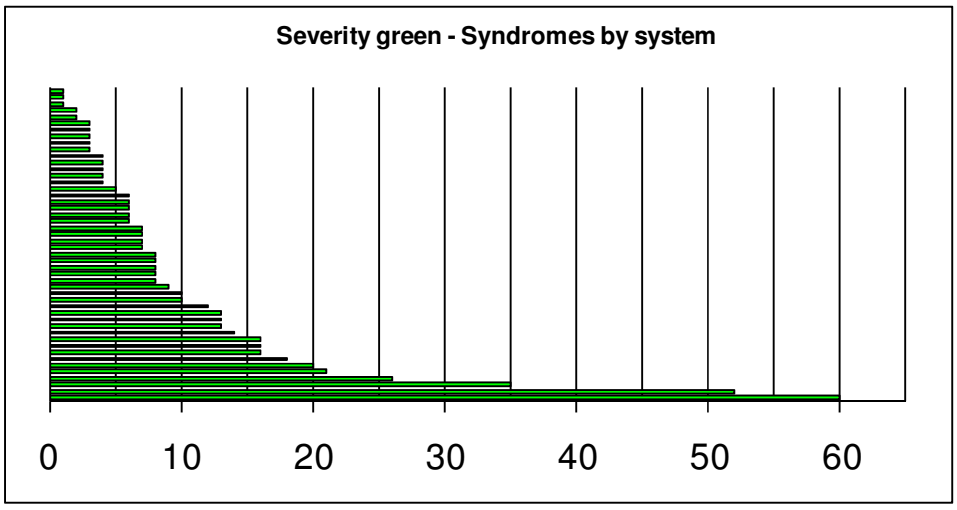
Podemos ver un nivel más de desagregación de esto, analizando la distribución de síndromes por sistema dentro de cada severidad.

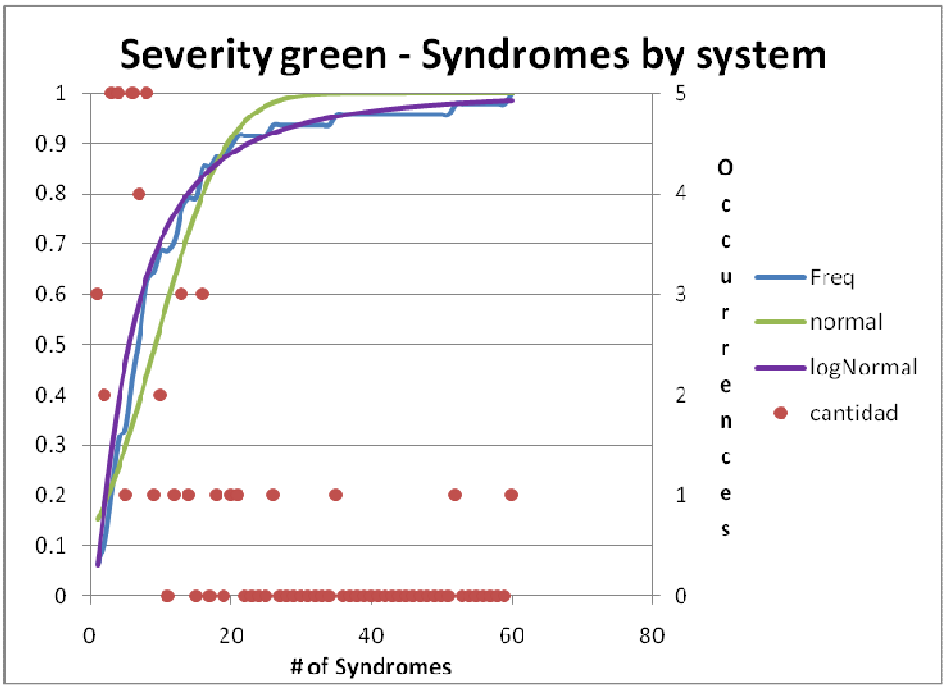

Severity red - Syndromes by system
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


Severity red - Syndromes by system
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
9
8
7
6
5
4
3
2
1
0
0 2 4 6 8 10 12 14
# of Syndromes
Ocurrences
Freq
normal
lognormal
cantidad


Severity yellow - Syndromes by system
0 5 10 15 20 25 30


Severity yellow - Syndromes by system
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
14
12
10
8
6
4
2
0
-2
0
5
10
15
20
25
30
# of Syndromes
Ocurrences
Freq
normal
logNormal
cantidad


Severity green - Syndromes by system
0
10
20
30
40
50
60



En el caso red, la normal tiene una media de 3 y un desvío estándar de 4, pero no se ajusta bien a las frecuencias. La logNormal tiene una media de 1.05 y un desvío estándar de 1 y ajusta mejor, aunque no tan bien como en el total.

En el caso yellow, la normal tiene una media de 5 y un desvío estándar de 4, pero se ajusta bastante mal a las frecuencias. La logNormal tiene una media de 0.9 y un desvío estándar de 1.2 y ajusta mejor, aunque no tan bien como en el total.

En el caso green, la aproximación normal tiene una media de 9.2 y un desvío estándar de 8. La logNormal tiene una media de 1.7 y un desvío estándar de 1.1, ajustándose muy bien a los datos de las frecuencias.

Aquí podemos ver la representación de una idea que luego nos condujo a generar varias estrategias: revisando estas distribuciones, se ve que si pudiéramos saber de antemano el sistema y la severidad del síndrome que tratamos de averiguar, el problema sería de una complejidad mucho menor que el problema general. Salvo para tres sistemas en la severidad *green*, siempre hay menos de 30 síndromes en cada combinación de sistema y criticidad. En el caso de severidad *red*, siempre hay menos de 15 y, salvo tres sistemas, menos de diez. En el caso *yellow*, descartando los tres más numerosos, hay 15 o menos.

A partir de estos datos, resultó una idea sumamente atractiva la de determinar de alguna manera el sistema y la criticidad, para luego trabajar con un conjunto mucho más reducido de síndromes.

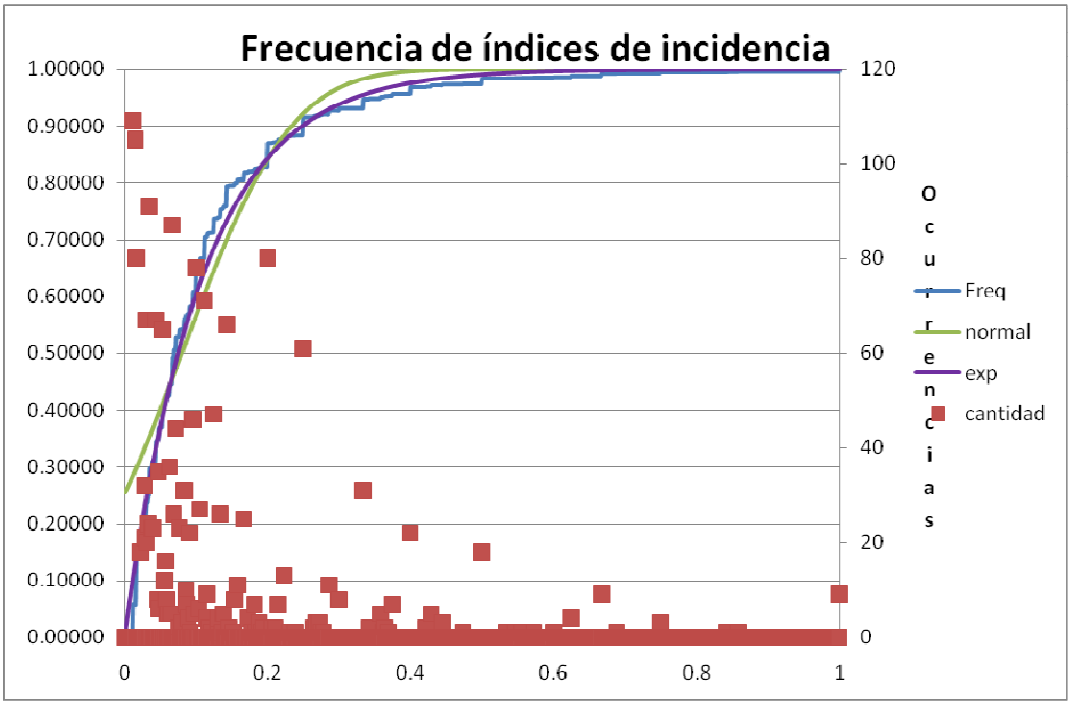
Rápidamente buscamos corroborar esta intuición apoyada por el estudio de la base, y los expertos de dominio confirmaron que, efectivamente, una estrategia bastante común de interrogatorio consiste en intentar determinar rápidamente el sistema y la criticidad para luego buscar el síndrome específico.

Llevados por esta intención, buscamos síntomas que caracterizaran con su presencia (o ausencia) a los grupos, ya que serían buenos candidatos a preguntar para poder determinar una de las combinaciones de sistema y severidad.

Así, definimos dos nociones para cuantificar la relación entre un síntoma y un grupo de síndromes. Podría tratarse de cualquier conjunto arbitrario de síndromes, pero en general la aplicamos a un sistema o a una combinación de sistema y severidad. Llamamos a estas nociones *índice de incidencia* y *probabilidad de no estar*. Ambas surgen intuitivamente de contar en cuántas de las definiciones aparece el síntoma.

El índice de incidencia se calcula dividiendo la cantidad de síndromes en cuya definición aparece el síntoma por la cantidad de síndromes del conjunto. Un índice de incidencia 1 quiere decir que el síntoma aparece en la definición de cada síndrome, un cero indica que no aparece en ninguna definición y un 0,5 quiere decir que aparece en las definiciones de la mitad de los síndromes.

La probabilidad de no estar parece en un principio el complemento del índice de incidencia, pero tiene una pequeña diferencia que se pone de manifiesto cuando hay definiciones alternativas para un síndrome, es decir, cuando la definición contiene una (o más) disyunciones. Se cuenta el total de definiciones alternativas, y se divide la cantidad de definiciones donde NO aparece el síntoma por ese total. Un 1 indica que no aparece en ninguna de las definiciones, un cero indica que aparece en todas, pero un 0,5 indica que aparece en la mitad de las definiciones (incluyendo alternativas), no necesariamente en la mitad de los síndromes.



Un reporte de los índices de incidencia de síntomas en los distintos sistemas donde aparecen (es decir, los que tienen índice de incidencia estrictamente mayor que cero) nos muestra 1913 relaciones. Las correlaciones tienden a los valores bajos, es decir, que los síntomas no se repiten mucho entre síndromes del mismo sistema. La curva normal con media 0.08 y desvío estándar 0.12 se ajusta aproximadamente a la curva de las frecuencias, pero una exponencial con lambda = 9.25 nos da una aproximación mucho mejor.

La probabilidad de no estar arroja resultados en el mismo sentido.

Ejemplo del problema a resolver

Contamos con una métrica fundamental: la cantidad de preguntas que necesita un operador en una sesión para cerrarla. Es decir, llegar al punto en que cuenta con suficiente información/evidencia como para decidir si la situación es o no una urgencia y consolidar un diagnóstico presuntivo.

Podríamos pensar en muchas otras métricas pero ésta apunta al eje central del problema. Las métricas clásicas de eficiencia en tiempo no tienen mucha importancia, en tanto que los tiempos de una interacción telefónica deberían ser de un orden muy superior al tiempo de procesamiento de una estrategia[11]. Dada la escala de la base de conocimiento, tampoco parece un problema importante la eficiencia en la utilización del espacio de memoria.

Definida esta métrica principal, tenemos también un objetivo definido por la práctica: deberíamos lograr que los síndromes de severidad `red` se decidan con menos de 3 o 4 preguntas, los de `yellow` con 4 o 5 y tenemos más margen con los síndromes de severidad `green`, donde el objetivo sería 6 o 7 pero se puede extender hasta 12. Esta variación es porque, lógicamente, las situaciones más urgentes deben resolverse con menos preguntas. En último término, son decisiones potencialmente de vida o muerte donde cada segundo puede hacer la diferencia.

Lo que nos interesa sobre todo es la cantidad de preguntas. Podríamos considerar normalizar esta métrica de alguna otra manera, por ejemplo dividiendo la cantidad de preguntas por la cantidad de síntomas del síndrome buscado. Pero en este dominio, dada una severidad concreta, la cantidad de preguntas es lo único determinante. La práctica indica que no deberían ser necesarias más preguntas que las que indicamos como objetivo.

En este punto, sería bueno revisar varios pasos de un ejemplo concreto extraído de la base de conocimiento y de las corridas de simulación.

Consideremos la *Acidosis Diabética*, un síndrome de severidad `red` cuya definición es "(*diabetes* **O** *antecedentes de diabetes*) **Y** (*pérdida de conciencia* **O** *obnubilación* **O** *aliento cetónico* **O** *disnea*)". A los fines de la simulación, podemos combinar cada una de las dos alternativas del primer miembro de la conjunción con cada una de las del segundo miembro, obteniendo ocho (2x4) conjuntos minimales de dos síntomas, cada uno de los cuales caracteriza este síndrome. A cada uno de ellos lo llamamos *subsíndrome*, y los nombramos con el nombre del síndrome y un número de secuencia arbitrario (basado en un recorrido de la expresión de definición).

Supongamos que elegimos el segundo de estos subsíndromes, con los síntomas *diabetes* y *obnubilación*. Configuramos al `answerProvider` con el mismo, y elige como síntoma inicial reportado por el paciente la diabetes. Nuestro objetivo sería que la estrategia permita determinar en menos de cuatro preguntas (ya que es un síndrome `red`, sería mejor tres) que el paciente presenta el síntoma *obnubilación* también, y que podemos diagnosticar presuntivamente una *Acidosis Diabética*, situación de emergencia que requiere una ambulancia y atención médica dentro de los quince minutos.

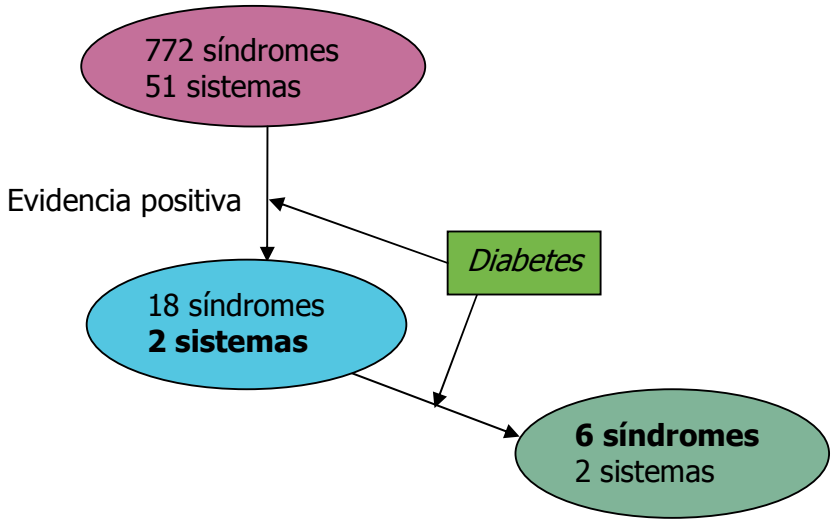
Aquí cabe aclarar que la situación no es real, sino que se trata de un caso con interés para la simulación únicamente, que permite apreciar las dimensiones típicas del problema en casos más reales. No es esperable que en una llamada real (seguramente hecha por otra persona, ya que el paciente padece una extrema confusión) no se comen-

[11] Por supuesto que la eficiencia en tiempo pasaría a ser un factor importante si el cálculo de una estrategia demorara más de lo que tarda el operador en hacer una pregunta por teléfono. Un tiempo de más de diez segundos entre pregunta y pregunta sería inaceptable.

te inmediatamente la *obnubilación* como parte del cuadro. Eso es algo que el `answerProvider` puede decidir automáticamente, a los fines del "juego" de la simulación. Pero para este ejemplo vamos a entrar en ese juego para ver cómo se resolvería la situación.

Con esta consideración en mente, analicemos por qué el `anwerProvider` elige *diabetes* y no *obnubilación* como primer síntoma para mencionar al "inicio de la llamada". *Diabetes* aparece en síndromes de dos sistemas, en tanto que *obnubilación* aparece en nueve sistemas diferentes. En general, un síntoma que aparece en varios sistemas diferentes es un síntoma que aporta poca información y suele ser de menor importancia (como la *fiebre*, por ejemplo). Por esa razón, se decide por *diabetes* como síntoma más importante y que sería mencionado en primer lugar.

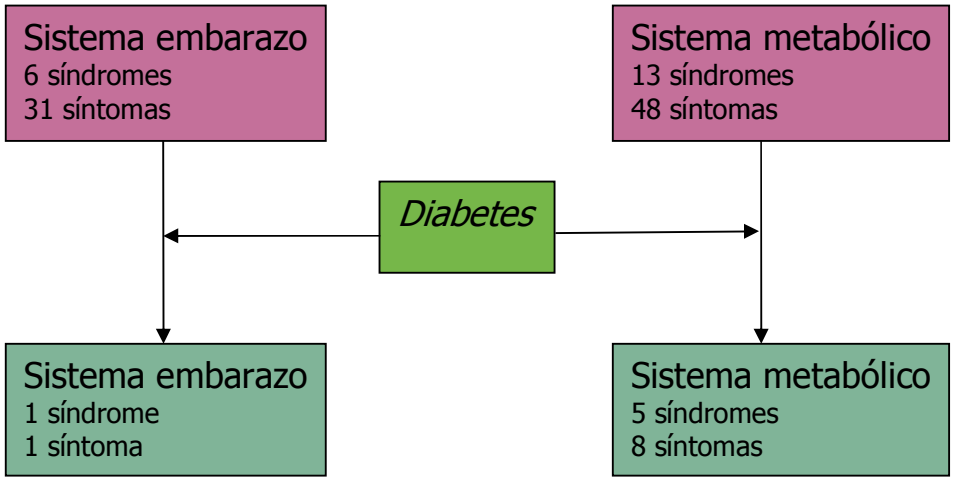
Tenemos la siguiente situación:



Ahora, empezamos con 772 síndromes distribuidos en 51 sistemas distintos. Vamos a enfocarnos en los que tienen *evidencia positiva*, entendiendo por esto que tengan en su definición un síntoma que el paciente haya confirmado tener. El único síntoma que mencionó hasta ahora es *diabetes*, pero eso sólo ya nos permite hacer un gran recorte del problema. En un primer paso, podemos filtrar los sistemas y quedarnos solamente con los que tienen evidencia positiva con el síntoma *diabetes.* Eso nos deja dos sistemas. Luego podemos filtrar con el criterio de evidencia positiva los síndromes dentro de esos sistemas. En esos dos sistemas hay dieciocho síndromes, pero sólo seis de esos síndromes tienen *diabetes* dentro de su definición.

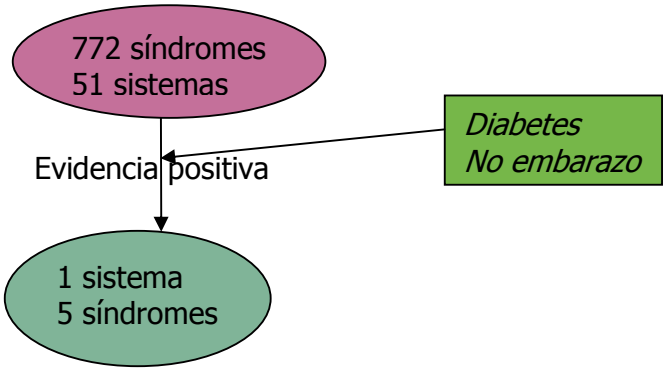
Practicar este recorte es una aplicación de razonamiento abductivo y no deductivo. La conclusión no está implícita en lo que conocemos, y podría incluso estar errada sin entrar en ninguna contradicción ya que, aunque el paciente tenga diabetes, su problema actual podría no estar relacionado con ello. Pero este tipo de razonamiento es muy frecuente en medicina y este procedimiento de reducción no difiere mucho del realizado por un experto.

Ahora, teniendo dos sistemas candidatos, una acción lógica podría ser tratar de decidirse y enfocarse en uno de los dos sistemas; nuestra próxima pregunta podría estar determinada por esa intención.

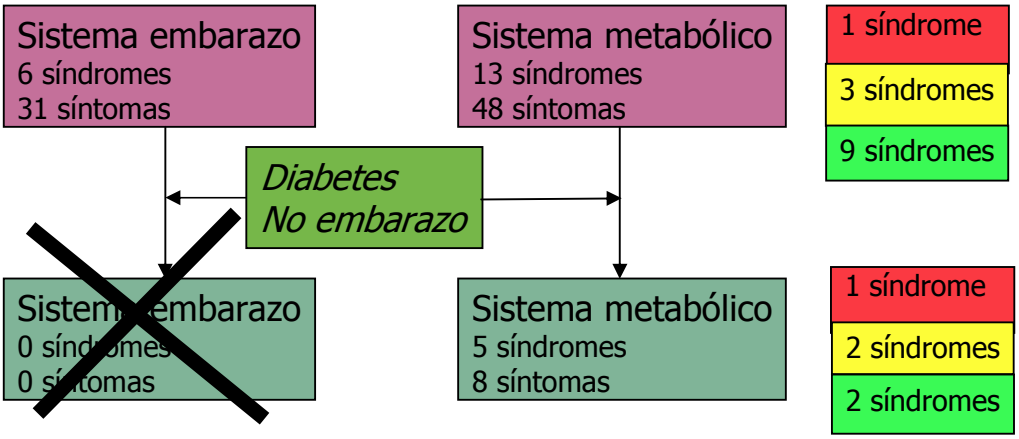


Hay un solo síntoma sin preguntar en los síndromes del *sistema embarazo*, que es el síntoma que indica si la paciente está embarazada. Así, elegimos el síntoma *embarazo* para confirmar o descartar todo lo relacionado con el sistema embarazo. Parece una buena pregunta ya que una respuesta positiva nos dejaría con un solo candidato (situación adecuada para cerrar la sesión) mientras que una negativa, al menos nos deja dentro de un solo sistema.

Notemos que esta es una forma posible de seleccionar la pregunta, con un criterio que apunta a determinar un sistema primero y luego la severidad. Hay muchos otros criterios posibles. Por ejemplo, podríamos intentar descartar urgencias primero, sin importar de qué sistema. Pero sigamos un poco más con el criterio de "sistema primero, severidad después" y veamos a dónde nos lleva.

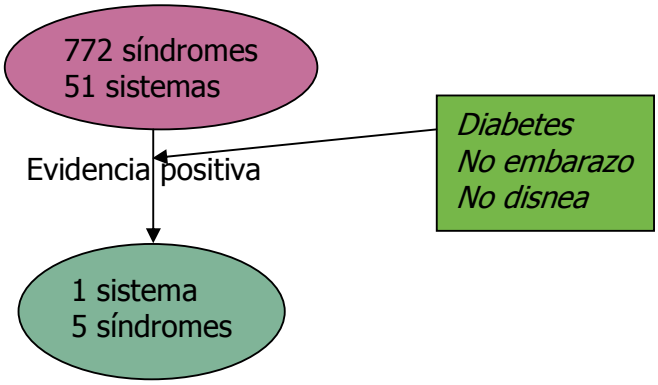


Efectivamente, en este caso el `answerProvider` da una respuesta negativa para *embarazo*, y nos quedan sólo cinco síndromes dentro de un solo sistema con evidencia positiva. Ahora "sabemos" que el paciente tiene un cuadro dentro del sistema metabólico. Graficamos la nueva situación, de acuerdo a la información de la sesión actual:

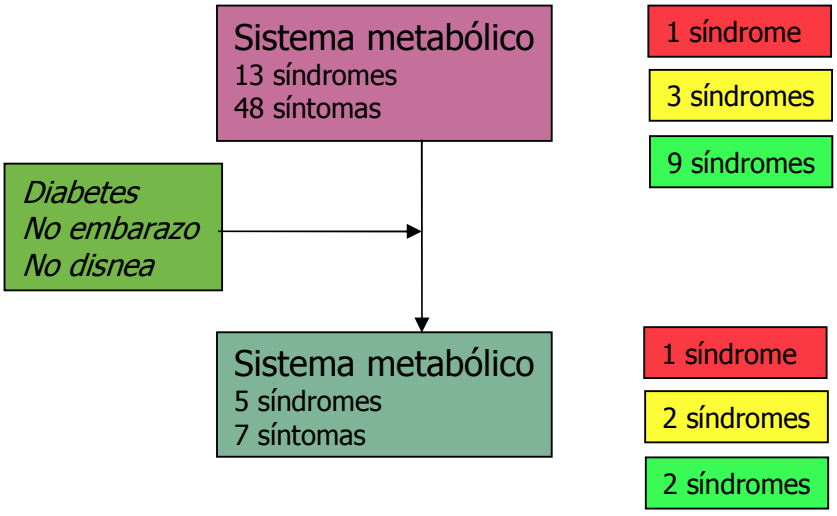


Hemos descartado el sistema de embarazo, y ahora nos quedan 5 síndromes candidatos en el sistema metabólico, distribuidos en las tres severidades. En las definiciones de esos síndromes aparecen ocho síntomas distintos, que son los candidatos para la(s) próxima(s) pregunta(s). Aquí sería imperativo contar con un buen criterio, ya que de otro modo nos arriesgamos a hacer ocho preguntas en el peor caso (además de la que ya hicimos sobre embarazo) y eso es demasiado para zanjar la cuestión de la emergencia.

Supongamos, para seguir un paso más, que elegimos el síntoma *disnea* para considerar una posible situación de urgencia.



Podría haber aportado una buena información con una respuesta positiva, pero la negativa en este caso nos deja prácticamente en la misma situación. Ni siquiera sirvió para descartar un síndrome, sólo permite rechazar algunas ramas de definición de algunos síndromes.



Aún nos quedan siete síntomas candidatos, un número excesivo para las cantidades de preguntas que tenemos como objetivo.

Nueva mecánica de pruebas

Teniendo los archivos de ExpertCare y, consiguientemente, una base de conocimiento real, comenzamos a hacer las pruebas simulando llamadas con síndromes extraídos de la misma. Dejamos de lado las pruebas más artificiales con las definiciones armadas con combinaciones automáticas de conjunciones, disyunciones y negaciones y empezamos a hacer recorridos totales o parciales de los conjuntos de síntomas de los síndromes reales.

Por supuesto, las primeras pruebas fueron con las mismas estrategias básicas y poco informadas que teníamos previamente implementadas. No se tocaron las estrategias, pero sí cambiamos (varias veces) la mecánica de las pruebas. A continuación exponemos la última versión de estas corridas de prueba.

```
testMoreCriticSeparationAllSyndromesRed
| file syndromes |
syndromes := (Syndrome syndromes asArray
  select: [:syn | syn severity == #red])
  sortBy: #name.
file := File newFile:
  '1testMoreCriticSeparationRed.txt'.
[
  self
      basicTestStrategy: MoreCriticSeparationStrategy
      on: syndromes
      reportingTo: file]
  ensure: [file close]
```

Este es un típico test, en este caso de la estrategia `MoreCriticSeparation` y del grupo de síndromes de severidad `red`. Separamos las corridas de prueba por grupos según la severidad, para comparar más fácilmente con nuestros objetivos.

Tenemos una primera etapa donde armamos el grupo de síndromes en que estamos interesados para la prueba. Como dijimos, generalmente se trata simplemente de seleccionar por una severidad en particular, pero en varios casos donde la corrida completa es muy larga, también en esta parte seleccionamos una muestra menor.

Luego creamos un archivo para recoger el reporte de salida con los resultados seleccionados de la corrida de prueba. Elegimos varios indicadores para reflejar el comportamiento en la corrida de simulación de cada caso.

La corrida se desarrolla íntegramente dentro del método `#basicTestStrategy:on:reportingTo:`, que recibe como parámetros una clase de estrategia, un conjunto de síndromes y un archivo para reportar resultados. Este llama a otras versiones del mismo mensaje, configurando algunos parámetros adicionales. El método más concreto y que soporta más variación es:

```
basicTestStrategy: strategyClass
on: syndromes
reportingTo: file
collectorClass: collectorClass
sessionClass: sessionClass
minSyndromes: anInteger
 0 | collector |
 1 self buildSampleFromFiles.
 2 collector := collectorClass new.
 3 self reportSyndromeHeadersOn: file.
 4 SyndromesTraveller with: syndromes do: [:ap |
 5     | t0 strategy |
 6     syndrome := ap syndrome.
 7     subsyndrome := ap subsyndrome.
 8     session := sessionClass new.
 9     session answerProvider: ap.
10     strategy := strategyClass new collector:
   collector.
11     strategy minSyndromes: anInteger.
12     session strategy: strategy.
13     t0 := Time now.
14     session start.
15     self reportSyndromeOn: file previousTime: t0.
16     subsyndrome name show]
```

Nos permite parametrizar la clase de estrategia, el conjunto de síndromes, el archivo de reporte, la subclase de `StatisticalCollector`, la clase de sesión y un umbral de mínimo de síndromes (que sólo usa un grupo de estrategias, siendo ignorado por los demás). Desarrollamos varias subclases de *collector* con distintas estrategias de *cache* para acelerar ciertas operaciones. Hay también algunas subclases de sesión para usar distintos criterios de corte, considerando las implicaciones entre síntomas y/o un algoritmo de *scoring*. Esta es la razón de que se utilice un parámetro para poder configurar la subclase deseada en cada prueba.

La primera acción de este método en la línea (1) es `#buildSampleFromFiles`. Ese método hace un primer chequeo para ver si es necesario leer la base de conocimiento o ésta ya se encuentra cargada en memoria. Luego, si resulta necesario leer la base de los archivos, ejecuta la inicialización de varias clases

y concluye invocando al `ConceptFileReader` y al `SyndromeFileReader` para que lean un grupo de archivos predeterminado. Ya hemos descripto el funcionamiento de estos objetos y el formato de los archivos en una sección anterior, así que ahora no vamos a profundizar más en esta parte.

Luego se instancia el `statisticalCollector` auxiliar a partir de la clase configurada como parámetro (2) y se invoca `#reportSyndromeHeadersOn:` para imprimir en el archivo de salida los encabezados (3). Finalmente se pide al `SyndromesTraveller` que ejecute un bloque (5 a 16) recorriendo la colección de síndromes recibida como parámetro (4).

El `syndromesTraveller` es un nuevo objeto auxiliar con la responsabilidad de hacer un recorrido exhaustivo de las combinaciones de síntomas que satisfacen las definiciones de una colección de síndromes, que recibe como parámetro. Veamos cómo se hace esto:

```
SyndromesTraveller class>>with: syndromes do: aBlock
    | instance |
    instance := self new.
    instance syndromes: syndromes.
    instance do: aBlock.
    ^instance
```

Este es el servicio de la clase para la creación de una instancia y ejecución del recorrido. Es el que se invoca desde los tests, y su funcionamiento es bastante sencillo y claro: crea una instancia, la configura con la colección de síndromes recibida y le envía el `#do:` con el bloque que debe ejecutar en cada paso del recorrido.

```
SyndromesTraveller>>do: aBlock
    syndromes do: [:syn |
      syndrome := syn.
      self subsyndromesDo: aBlock]
```

Este método también es sencillo. Itera sobre la colección de síndromes recibida y por cada uno, se envía el `#subsyndromesDo:` con el mismo bloque a ejecutar.

```
SyndromesTraveller>>subsyndromesDo: aBlock
    syndrome subsyndromes do: [:sub |
      subsyndrome := sub.
      symptoms := sub symptoms.
      self subsyndromeDo: aBlock]
```

Este es apenas un poco más complejo que el anterior: requiere los *subsíndromes* del síndrome sobre el que está iterando, y los recorre a su vez con un `#do:`. Por cada *subsíndrome*, pide el conjunto de síntomas que satisface la definición (con el mensaje `#symptoms`) y llama al `#subsyndromeDo:` con el bloque a ejecutar. Recordemos que los *subsíndromes* de un síndrome son los que se obtienen de cada rama de alternativas de la definición del síndrome, es decir que la definición de un *subsíndrome* ya no tiene posibilidades alternativas, siendo simplemente una conjunción de síntomas.

Aquí finalmente se llega a un método de cierta complejidad, `#subsyndromeDo:`, aunque esta complejidad se debe principalmente a que se re-

suelven varios casos juntos dentro del mismo método en lugar de refactorizarlo en varios métodos. Los detalles de esta implementación y los casos involucrados pueden verse en el Apéndice.

Luego de esta larga explicación sobre el funcionamiento del `syndromesTraveller`, volvamos al test y veamos qué hace el bloque que tiene como argumento el `answerProvider` (líneas 5 a 16 en `#basictestStrategy:on:reportingTo:…`).

Lo primero que hace es registrar en variables propias del test el síndrome y el *subsíndrome* de la iteración, sacándolos del `answerProvider`. Instancia una sesión con la clase recibida como argumento y le asigna el `answerProvider` suministrado por el *traveller*. Instancia una estrategia a partir de la clase indicada por el test y la configura con el collector previamente construido y el umbral de síntomas mínimos (que, como ya dijimos, afecta sólo a un grupo de estrategias). Luego arranca la sesión y cuando termina, invoca el método `#reportSyndromeOn: file previousTime: t0` para registrar la información correspondiente a esta iteración en el archivo de salida. El último paso es sólo para mostrar el *subsíndrome* iterado en pantalla, para poder saber por dónde va la prueba.

A continuación mostramos el encabezado del archivo de salida, para explicar qué tipo de información nos interesó registrar.

Syndrome	Definition	Symptoms	Severity	#questions	
#for System	#for Severity	#for Syndrome	Time (min)	#Full diags	
Is Full?	Is Multiple?	Is Diag Error?	Is Sev Error?	Succeeded?	Clue1
Clue2	Question 1	Question 2	Question 3	Question 4	Question 5

Syndrome, *Definition* y *Severity* se refieren al síndrome de cada iteración, en tanto que *Symptoms* indica los síntomas que caracterizan al *subsíndrome*. *#questions* es la cantidad de preguntas realizadas en la sesión (nuestra métrica principal) y *#for System*, *#for Severity* y *#for Syndrome* se aplican en el último grupo de estrategias (del que hablaremos más adelante) que tienen etapas distintas para buscar esas tres cosas. *Time* es el tiempo en minutos que se tardó la ejecución y el cierre de la sesión, no tiene mayor interés para el trabajo pero tenía cierto interés práctico durante las corridas de pruebas.

Ahora aparece una serie de indicadores que están relacionados con nuestra mecánica de pruebas, el criterio de cierre de sesión y la adecuación de la base de conocimiento.

#Full diags es la cantidad de diagnósticos que estaban completos al final de la sesión, una medida interesante porque lo ideal es que haya uno solo y que sea el de la prueba. Esto, en muchos casos no sucede; es común en nuestra base que haya varios diagnósticos completos al final de la sesión.

Is Full? se refiere justamente a si el diagnóstico del síndrome elegido para la iteración está completo; si no lo está, la sesión se cerró con un diagnóstico "errado". Esto no siempre es un error, es posible (y se da en casos reales y concretos) que el diagnóstico elegido esté implicado por el buscado, y se encuentra y se cierra antes.

Is Multiple? se refiere a si hay más de un diagnóstico cerrado.

Is Diag Error? indica si hay un error de diagnóstico, es decir si el diagnóstico del síndrome buscado no está completo. *Is Sev Error?* en cambio, indica si está equivocada la severidad del diagnóstico de cierre de la sesión. Se considera equivocada de dos

maneras distintas: si el síndrome es de severidad `green`, se considera error tener algún síndrome completo de severidad `red` o `yellow` ya que indicaría una falsa alarma con una severidad superior a la "real"; si el síndrome es `red` o `yellow`, se considera error sólo si ninguno de los diagnósticos completos es de la severidad requerida. Nuevamente, estos son errores en comparación con lo que deseamos, pero hay casos en que las definiciones hacen que suceda esto sin que sea un error real. Utilizamos este nombre de "error" para abreviar, y nos indica que sería necesario revisar el caso.

Succeeded? indica si la sesión terminó con un solo diagnóstico completo y es el del síndrome de la prueba.

Clue1 y *Clue2* son las pistas indicadas por el *answerProvider*.

Question1 en adelante son los primeros cinco síntomas para preguntar determinadas por la estrategia.

Resultados de las primeras pruebas

Este es un cuadro que resume los resultados de las pruebas con las cinco estrategias previamente descriptas, para los casos de severidad `red`.

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
Sequential	739,04	745	19,00	5,58
Random	746,58	732,5	18,05	5,46
LessSatisfiers	726,28	932	18,84	0,00
MiddleSatisfiers	579,28	794	18,05	2,44
MoreSatisfiers	85,29	52	14,23	2,51
MoreCriticSeparation	174,44	35	18,29	7,13

Se puede ver fácilmente que ninguna de estas estrategias está siquiera cerca de los objetivos de cantidad de preguntas que mencionamos antes. La más prometedora, que sería `MoreCriticSeparation`, aún está muy por encima de la cantidad buscada.

Cabe mencionar además que en el caso de `LessSatisfiers` se tomó una muestra y no se usó la totalidad de los casos, ya que resultaba demasiado lento el proceso, por la gran cantidad de preguntas y el cálculo requerido en cada caso.

Es interesante ver que en ningún caso (salvo el error de severidad de `LessSatisfiers`) el error llega a cero. Recordemos que el nombre de "error" es utilizado para abreviar una "falta de coincidencia" entre lo encontrado y el objetivo buscado.

Muchas de estas discordancias se deben a limitaciones de la mecánica de prueba y a características particulares de la base de conocimiento.

Fue necesario incluir en la base algunos síndromes muy genéricos, con definiciones de un solo síntoma, para poder diagnosticar casos en que sólo ese síntoma estuviera presente. Esto hace que haya altas probabilidades de que esos diagnósticos se completen muy rápidamente, antes que las definiciones de otros síndromes más complejos lleguen a satisfacerse. Cuando alguno de esos síndromes tiene severidad red o yellow, la sesión se termina prematuramente al completarlo, ya que un síndrome completo de una de esas severidades es condición suficiente para mandar una ambulancia. Este es

un sesgo que se le da intencionalmente al sistema para contemplar el hecho de que un falso negativo es potencialmente mucho más peligroso que un falso positivo en la severidad. Pero esta es también la razón de que haya tasas relativamente altas de "error". En cada uno de estos casos, se hizo una verificación y comprobación (automática) de que la definición del síndrome buscado era implicada por la definición del síndrome que causó el cierre de la sesión.

Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
Sequential	961,12	975	9,61	5,19
Random	1001,61	1067	16,76	9,72
LessSatisfiers	662,00	730,5	0,00	0,00
MiddleSatisfiers	460,41	568	3,13	0,00
MoreSatisfiers	92,82	43	7,75	4,89
MoreCriticSeparation	215,57	122	11,72	5,04

Pocas diferencias se observan en los resultados de las pruebas con severidad `yellow`. Las cantidades de preguntas son un poco más altas y las tasas de error son un poco más bajas, pero no hay una diferencia significativa con respecto a los objetivos. Es interesante ver que en este grupo, *MoreSatisfiers* funciona mejor que *MoreCriticSeparation*, a diferencia de lo que vimos antes.

Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
Sequential	1049,18	1245	17,72	9,85
Random	1099,11	1359	17,75	9,91
LessSatisfiers	561,03	526	11,76	0,00
MiddleSatisfiers	764,34	951	25,93	1,39
MoreSatisfiers	259,46	220	14,62	4,71
MoreCriticSeparation	301,97	266,5	13,21	4,60

Otro tanto sucede con los resultados de severidad green. Aquí las cantidades de preguntas suben notablemente, pero eso era lo esperado, ya que hay muchos más síndromes en la severidad más baja y puede resultar más difícil decidir con poca información. Las tasas de error son un poco más altas también, reflejando una mayor probabilidad de encontrar un síndrome genérico de gravedad más alta antes de completar las respuestas necesarias para el síndrome de prueba.

Estos primeros resultados no hicieron más que confirmar lo obvio: este primer grupo de estrategias que utilizan muy poca información e inteligencia, no tienen ninguna chance de resolver el problema satisfactoriamente. No obstante, es también un buen indicador de que valía la pena encarar el trabajo, porque significa que el problema no es trivial y no podía ser resuelto sin un estudio y un esfuerzo adecuados.

Intermedio: Estrategias y adivinación

Nuestros siguientes intentos estuvieron motivados por la información estadística recolectada en la base de datos, que sugería que intentar determinar por algún medio la combinación de sistema y severidad reduce mucho el tamaño del problema. Esto coin-

cide también con un criterio intuitivo de aprovechamiento de la información contenida en cada síndrome. Además, consultas con especialistas médicos confirmaron que es una estrategia aplicada –a grandes rasgos— en los interrogatorios reales.

Así que, una vez elegido el camino, sólo nos faltaba responder la pregunta clave de este enfoque: ¿cómo adivinar el sistema y la severidad?

Intentamos tres bases para la adivinación, usando alternativamente los dos indicadores ya descriptos en la sección de estudios estadísticos: el *índice de incidencia* y la *probabilidad de no estar*. Con esto desarrollamos tres estrategias principales: `GuessSystemByFrequency`, `GuessSystemBySeverity` y `GuessSystemByPairs`. También definimos tres estrategias auxiliares: `MoreCorrelation`, `LessNegation` y `LessNegationPair`. Podríamos también haber implementado `MoreCorrelationPair`, pero los resultados no justificaban el intento. Las diferencias entre usar auxiliares basados en el *índice de incidencia* (`MoreCorrelation`) o en la *probabilidad de no estar* (`LessNegation`) resultaron casi insignificantes en la enorme mayoría de los casos.

En cuanto a las tres estrategias principales, en *ByFrequency* intentamos usar la frecuencia de los síndromes como un indicador para tratar de guiar la búsqueda del sistema, eligiendo entre los candidatos el que tuviera síndromes de frecuencia más alta. Esta era una mala idea desde un principio, por dos razones. La frecuencia de los síndromes fue ingresada en la base de conocimiento de forma subjetiva por diferentes especialistas, sin tener un apoyo en estadísticas de casos de una población concreta, por lo que es una medida que puede servir para dar una orientación al operador humano, pero dudosa para usar un criterio automático. Además, aunque fuera completamente confiable, la mecánica de nuestras pruebas recorre imparcial y exhaustivamente todos los síndromes, la misma cantidad de veces, independientemente de su supuesta frecuencia. Hicimos intentos de modificar la mecánica de pruebas para simular una población que reflejara una distribución de síndromes relacionados con esa frecuencia, pero los tiempos de la simulación crecían y el valor agregado de esa información era dudoso.

En *BySeverity*, como su nombre lo indica, usamos como criterio para elegir el sistema el que tuviera los síndromes de severidad más alta entre los candidatos.

La estrategia *ByPairs* tuvo un origen distinto, donde no cambiamos el criterio (es el mismo que en *BySeverity*) sino que intentamos mejorar el tiempo de cálculo de los índices de incidencia buscando, en lugar de un solo síntoma con el índice de incidencia más alto, el par de síntomas que tuviera el mejor índice. Detrás de esto había una intuición de que quizás un solo síntoma no estuviera suficientemente correlacionado con un sistema, pero un par sí.

Veamos un poco de la implementación de `GuessSystemBySeverity`, aunque los detalles concretos de la misma están en el Apéndice. La estrategia tiene un *collector* para navegar la base de conocimiento y una estrategia auxiliar en la variable de instancia `secondary`. Esta estrategia auxiliar es la que intenta resolver, una vez (presuntamente) adivinado el sistema, cuál es el mejor síntoma para preguntar. Puede ser *MoreCorrelation* o *LessNegation*; como dijimos, encontramos muy poca diferencia entre ambas. Además, tiene un diccionario donde se registran castigos para los sistemas que han resultado malos candidatos.

Esta estrategia busca un sistema candidato recorriendo las severidades según un orden preestablecido que el *collector* conoce. Es el orden de gravedad decreciente, primero *red*, luego *yellow* y finalmente *green*. Mientras encuentre un sistema candidato en la severidad más grave, la iteración no llega a la siguiente. Es una forma de expresar que

primero se revisan todos los sistemas candidatos de la severidad más alta, y luego los de las siguientes severidades en orden.

La implementación de la estrategia `GuessSystemByFrequency` es casi igual, excepto que usa las frecuencias ordenadas (de más frecuente a menos frecuente) en lugar de las severidades, y busca los síndromes por sistema y frecuencia.

En ambos casos se busca el sistema candidato, dada una severidad o una frecuencia. Se empieza pidiendo al *collector* los síndromes agrupados por sistema, filtrando sólo los que tengan la severidad/frecuencia correspondiente. Se arman pares sistema ➔ soporte, donde el soporte intenta ser una medida de la evidencia que respalda la hipótesis de que el síndrome buscado sea de ese sistema y esa severidad. De estos candidatos se remueven los que tengan un castigo alto (mayor que diez, arbitrariamente) o un soporte negativo muy bajo. Luego se ordenan los pares por soporte y se devuelve el que tiene soporte más alto.

El soporte para un sistema en la severidad elegida se calcula de la siguiente manera: se piden al *collector* los síntomas que aparecen en la base de conocimiento con ese sistema y severidad. Si no hay, se devuelve -1 como convención, indicando que no hay soporte en absoluto para ese sistema. En el caso normal, se recorren los síntomas y se verifican las respuestas de la sesión, sumando un punto por cada síntoma que ya haya sido preguntado y respondido afirmativamente (evidencia positiva) y restando medio punto por cada síntoma preguntado y respondido por la negativa (evidencia negativa). Es una forma ingenua de computar alguna medida de la credibilidad/verosimilitud de un sistema, viendo qué síntomas de ese sistema ya han sido confirmados.

Veamos cómo la estrategia secundaria intenta elegir el síntoma candidato. Pide a su propio *collector* los síntomas del sistema con que fue configurada previamente (el candidato elegido por la estrategia principal), y se queda con la intersección entre esos y los que no han sido preguntados. Genera pares síntoma ➔ índice de incidencia, obteniendo los índices de cada síntoma del *collector*. Ordena por este índice y devuelve el síntoma candidato que tenga la correlación más alta.

La implementación de `LessNegationStrategy` es en un todo similar, sólo que pide al *collector* la *probabilidad de no estar* en lugar del *índice de incidencia*.

Así, ya hemos visto cómo funcionan las estrategias primarias `GuessSystemBySeverity` y su variante `GuesssystemByFrequency`, en combinación con las estrategias secundarias alternativas y casi equivalentes `MoreCorrelationStrategy` y `LessNegationStrategy`. Nos queda por analizar el funcionamiento de la última variante, la que intenta una búsqueda por pares de síntomas en lugar de síntomas aislados.

Se complica un poco por el manejo de los pares, pero en líneas generales es parecida a `GuessSystemBySeverity`. La búsqueda del sistema candidato procede exactamente como en `GuessSystemBySeverityStrategy`, mientras que la estrategia auxiliar de `LessNegationPairStrategy` funciona igual que `LessNegationStrategy` excepto por el detalle de que busca y devuelve un par de elementos en lugar de uno sólo.

La única idea para usar esta estrategia en lugar de `GuessSystemySeverity` era aprovechar un poco mejor el tiempo (que en ese momento previo a la implementación de buenos caches era muy alto) de calcular las *probabilidades de no estar* de los síntomas en un sistema y severidad determinados. Por otra parte, esto introduce una cierta inercia, ya que lo observado en un momento dado determina dos preguntas, en

lugar del comportamiento habitual de las estrategias de reevaluar la situación dentro del nuevo contexto en cada paso.

Resultados de las pruebas intermedias

Habiendo explicado el funcionamiento de estas tres estrategias que podemos llamar intermedias (por su ubicación en el desarrollo del proyecto) o de adivinación (por los métodos que utilizan) pasamos a mostrar los resultados para los casos de severidad `red`.

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
GuessSystemByFrequency	362,37	219	18,18	1,46
GuessSystemBySeverity	92,12	134	75,77	66,15
GuessSystemUsingPairs	9,40	4	46,56	34,20

La estrategia de los pares tiene resultados prometedores, hasta el momento en que miramos sus tasas de error, muy altas en comparación con las del primer grupo. También son exageradamente altos los índices de error para `GuessSystemBySeverity`. El error en `GuessSystemByFrequency` es similar al de las estrategias del primer grupo y su cantidad de preguntas también, siendo demasiado alta esta medida como para resultar interesante.

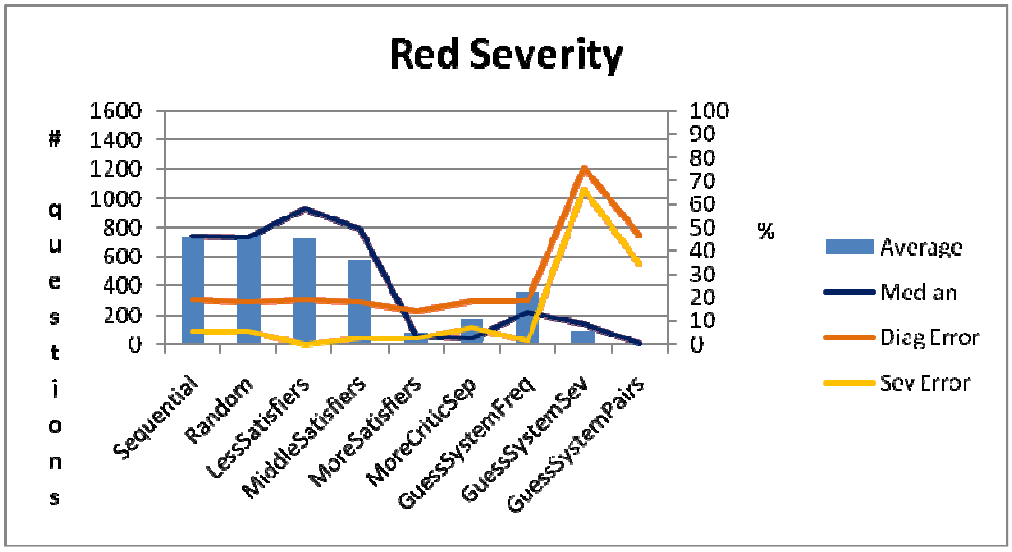
Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
GuessSystemByFrequency	295,86	73	13,39	10,12
GuessSystemBySeverity	110,33	134	75,89	72,75
GuessSystemUsingPairs	18,36	20	88,43	85,35

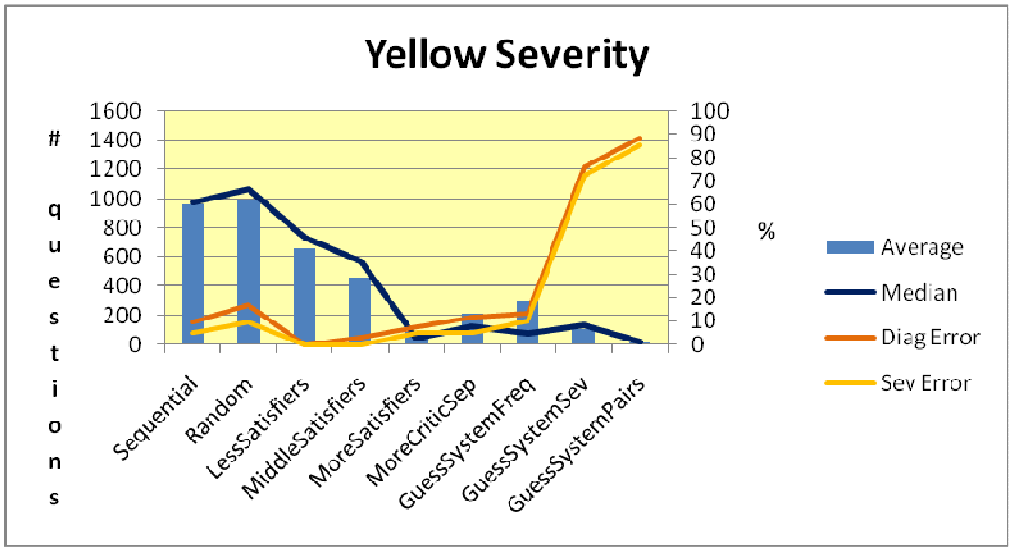
El análisis de los resultados en los síndromes de gravedad `yellow` no da mayores esperanzas. Nuevamente, las tasas de error y de preguntas de *ByFrequency* son comparables a las del primer grupo de estrategias (las preguntas están en un punto medio entre las mejores y las peores aunque el error es más alto), mientras que los errores de las otras dos son exageradamente altos como para tomarlas en cuenta. Se observa que las cantidades de preguntas suben con respecto a los síndromes de severidad `red`.

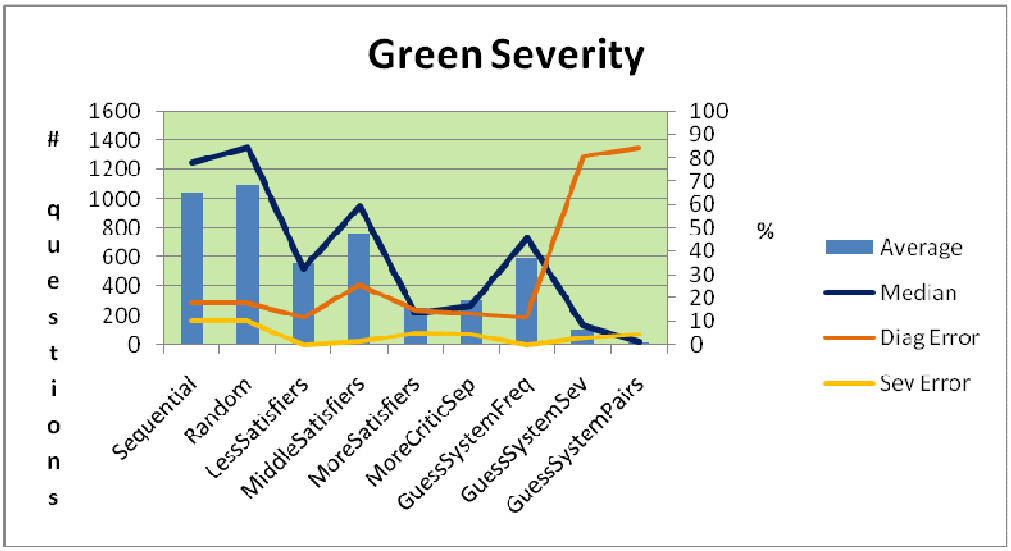
Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
GuessSystemByFrequency	596,32	739	11,76	0,00
GuessSystemBySeverity	110,20	132	80,90	2,83
GuessSystemUsingPairs	17,13	20	84,50	4,42

En los resultados para síndromes de severidad `green`, hay una mejoría. En *ByFrequency*, el error es bastante bajo aunque las cantidades de preguntas son bastante altas en comparación con el primer grupo. En las otras dos las cantidades de preguntas son más bajas que las del primer grupo, y el error de severidad también, aunque siguen teniendo una tasa extremadamente alta de errores en el diagnóstico.

Analizando el conjunto, aunque la idea de adivinar el sistema y la severidad para reducir la complejidad del problema es interesante, la implementación de estas estrategias que intentan utilizarla no dio buenos resultados.







Estrategias basadas en soporte

Introducción

En las estrategias intermedias usamos un enfoque que introduce un elemento heterogéneo y agrega complejidad innecesaria: utilizamos dos estrategias con criterios completamente diferentes en dos etapas de la resolución del problema. Esto surgió de pensarlos como problemas separados y enfocar una estrategia distinta en cada uno, pero una reflexión posterior nos mostró que no había ninguna base para pensar que elegir sistema y severidad o un síndrome dentro del sistema y la severidad fueran problemas distintos. Por el contrario, usar el mismo criterio y la misma herramienta/estrategia a lo largo de la sesión sería mucho más simple.

El problema es que partimos de los indicadores de incidencia o probabilidad de no estar, y ambos tienen sentido para relacionar un síndrome (o grupo de síndromes) dentro de un grupo determinado (como un sistema). No era fácil pensar en aplicarlos para elegir un sistema, ya que habría que pensar y definir la correlación del sistema con algo que no estaba claro qué podría ser.

Pero casi inadvertidamente había surgido otro indicador, mucho más interesante para aplicarlo a cualquier síndrome o grupo de síndromes y más fácil de usar como medida y objetivo a mejorar para una heurística. Este indicador es el *soporte* que usamos para tratar de adivinar el sistema en las estrategias intermedias.

La intención del soporte era dar una medida de credibilidad, posibilidad o verosimilitud de un síndrome de acuerdo a la evidencia dada en un contexto de sesión de interrogatorio. Esta idea es fácil de generalizar a cualquier grupo de síndromes, arbitrario o no, simplemente sumando los soportes de cada síndrome y normalizando la suma contra algún máximo. Así, es fácil pensar en el soporte de un sistema, de una severidad o de una combinación de sistema y severidad.

Para calcular el soporte usamos una aproximación intencionalmente ingenua y simplista. Consideramos que el soporte es cuánto hay de conocido o confirmado en la definición de un síndrome, en relación al total de lo que podríamos conocer o confirmar. Por ejemplo, si la definición de un síndrome es la conjunción de tres síntomas, y hemos preguntado dos que tienen respuesta positiva, en este momento ese síndrome tendría un soporte de 2/3. Es más alto que otro que tuviera cinco síntomas en la definición y también dos con presencia confirmada por el paciente, que tendría un soporte de 2/5. Cual-

quier síndrome que no tenga ningún síntoma confirmado en la sesión, tiene un soporte de cero.

Tendríamos que dar soporte negativo a los que tengan síntomas ya preguntados y negados en la sesión. Volviendo al primer ejemplo, un síndrome con tres síntomas en la definición, dos de ellos preguntados y presentes. Si preguntamos el tercero y el paciente no lo presenta, el soporte total debería ser negativo, porque en este caso ya sabemos que la definición se evalúa como falsa en el contexto. En cambio, tomemos el síndrome con cinco síntomas, uno de ellos preguntado y positivo. Supongamos que hay alguna disyunción y negar uno cancela una rama, pero no es concluyente. Si preguntamos el segundo y tiene una respuesta negativa, ¿se cancela el soporte, haciendo +1/5 – 1/5 y volviendo a cero? Un poco de experimentación y análisis de la base de conocimiento sumado a la experiencia de interrogatorios reales nos hizo pensar que eso no era bueno y nos llevó a darle menos peso a las negativas.

El paso siguiente fue plantear una heurística que aprovechara y tratara de mejorar el indicador. La primera y más simple que nos surgió era elegir el par de sistemas con mayor soporte como candidato e intentar decidir entre ellos. De manera similar a como se hace con el *scoring* en ExpertCare, buscamos generar suficiente diferencia de soporte entre los dos primeros sistemas como para que sea claro que hay mucha más evidencia del primero, al punto de que no valga la pena analizar el segundo (ni ninguno con soporte más bajo). Si después de la primera pregunta tenemos, por ejemplo, dos sistemas con soporte de 0.6 y 0.54 y el resto con soporte cero, no hay información como para descartar el segundo y aún tendríamos que considerar los dos sistemas como posibilidades interesantes. Entonces, buscamos preguntar un síntoma que genere una brecha más grande entre ellos, analizando cómo variaría el soporte de cada uno con la respuesta de ese síntoma. Este análisis lo podemos hacer calculando de nuevo el soporte de ambos sistemas en un contexto con las mismas respuestas actuales, más la respuesta del nuevo síntoma. Analizamos esto para todos los síntomas candidatos, y nos quedamos con el que maximice la diferencia entre los dos sistemas candidatos.

En algún punto, un valor umbral, la diferencia de soporte entre ambos sistemas basta para "convencernos" de concentrar nuestros esfuerzos en el primero. Luego buscamos de manera similar una severidad que tenga un soporte mucho mayor que las otras dentro de ese sistema.

Una vez acotados un sistema y una severidad, nos queda un grupo de síndromes de tamaño manejable. Dentro de ese grupo, repetimos el procedimiento y buscamos los dos (o más) síndromes con mayor soporte, apuntando a generar mayor diferencia y nuevamente "convencernos" de que hay un candidato con clara diferencia, que sería la respuesta dada por la estrategia, posiblemente cerrando la sesión con ese diagnóstico presuntivo. Para ello analizamos cada síntoma de los síndromes, evaluando cómo cambiaría el soporte en el grupo por respuestas positivas o negativas y elegimos el síntoma que provoque una mayor diferencia de soporte entre los primeros al preguntarlo.

Con variaciones y refinamientos de esta idea sencilla, surgió toda una familia de estrategias que probaron ser muy superiores a las anteriores.

Implementación de *SupportSeparation*

Veamos algunos detalles del código y estructura de la principal, la primera y la más simple de las estrategias basadas en soporte: `SupportSeparationStrategy`.

Lo primero es indicar que, a diferencia de las primeras estrategias que usan muy poca información, se agregan unas cuantas variables de instancia para registrar el estado

y la historia de la operación de la estrategia en la sesión. Además de la variable `symptoms` que hereda de `Strategy` y sigue teniendo la lista de síntomas que no se han preguntado, incorpora las siguientes:

- `collector` es un *statisticalCollector*, auxiliar que sirve para navegar y obtener información de la base de conocimiento.
- `session` conserva la sesión de interrogatorio en curso, para poder obtener información de contexto de la misma en cualquier momento y en cualquier método.
- `system` es el sistema candidato del momento actual, el que se ha elegido hasta ahora y se está analizando.
- `severity` es la severidad candidata del momento actual. Dentro del sistema candidato, es la severidad que se ha elegido hasta ahora como más prometedora y se está analizando.
- `history` es una colección que guarda los pares de sistema y severidad que se han analizado hasta ahora. Esto sirve en principio para no repetir el análisis de candidatos ya descartados.
- `candidates1/candidates2` son dos colecciones de síntomas candidatos que se están comparando en el momento actual. Se usan en varios cálculos auxiliares.
- `syndromes1/syndromes2` son dos colecciones de síndromes candidatos que se están comparando en el momento actual. Se usan en varios cálculos auxiliares.
- `group1/group2` son dos grupos específicos síndromes candidatos que se están comparando en el momento actual, por ejemplo dos sistemas o dos severidades. Se usan en varios cálculos auxiliares.

Ahora analicemos el código como en las estrategias anteriores, desde el punto de entrada principal:

```
SuppportSeparationStrategy>>nextSymptomFor: aSession
    session := aSession.
    self updateExcludedFrom: aSession.
    system isNil ifTrue: [^self
nextSymptomLookingForSystem].
    severity isNil ifTrue: [^self
nextSymptomLookingForSeverity].
    ^self nextSymptomLookingForSyndrome
```

Primero se registra la sesión en curso en la variable de instancia dedicada a tal fin y se actualiza la lista de excluidos del *collector* auxiliar. Luego, tenemos tres casos alternativos: que actualmente no haya un sistema candidato analizado y tendríamos que escoger uno en `#nextSymptomLookingForSystem`, o que haya un sistema pero todavía no se haya elegido una severidad candidata dentro del mismo y se elegiría una en `#nextSymptomLookingForSeverity`. El último caso es donde ya hay un sistema y una severidad elegidos como candidatos, y se está buscando un síndrome candidato.

Veamos el primer caso, ya que es también el que el primero se da cuando empieza a funcionar la estrategia, sin ningún candidato. Debemos buscar un sistema.

```
nextSymptomLookingForSystem
 ^self nextSymptomLookingForSystemComingFrom: nil.
```

Esto es sólo un pase, para llamar a otro método más general usando un argumento que indica que no tenemos un origen interesante que considerar.

```
nextSymptomLookingForSystemComingFrom: aSystemOrNil
 | support candidates difSupport |
 support := self supportForPositiveSystemsComingFrom:
   aSystemOrNil.
 support isNil ifTrue: [^nil].
 candidates := self systemCandidatesFrom: support.
 candidates isCollection ifFalse: [^candidates].
 difSupport := self differencesOfSupportCausedBy:
   candidates.
 ^self answer: difSupport last first
```

Aquí ya interviene fuertemente la noción de soporte: lo primero que se hace es calcular el soporte para los sistemas con evidencia positiva. En general, nos devuelve un arreglo de pares (sistema, soporte), a partir de los cuales se filtra y se obtiene un conjunto de síntomas candidatos. Finalmente, se calculan las diferencias de soporte que genera cada candidato, en un arreglo de pares (síntoma, diferencia) ordenados de menor a mayor diferencia, por lo que devolvemos el último síntoma (que es el que origina una mayor diferencia en el soporte). Recordemos que estas diferencias son entre los dos primeros sistemas candidatos, aquellos que tienen soporte más alto y entre los que estamos tratando de decidir.

```
supportForPositiveSystemsComingFrom: aSystem
    | positive support |
    positive := self systemsWithPositiveEvidence.
    positive remove: aSystem ifAbsent: [].
    history
      select: [:pair | pair last isNil]
      thenDo: [:pair | positive remove: pair first
ifAbsent: []].
    support := positive
      collect: [:sys | Array with: sys with: (self
supportOfSystem: sys)].
    support := (support sortBy: #second) reversed.
    ^support
```

Este método primero selecciona en la variable `positive` los sistemas con evidencia positiva que los apoye. Esto refleja que, en principio, analizamos sólo síndromes que incluyan alguno de los síntomas ya mencionados por el paciente. Esto podría parecer apresurado pero se basa en que es muy raro que el paciente presente un síndrome pero mencione un conjunto de síntomas totalmente disjunto con los de ese síndrome. Representa una aplicación del razonamiento abductivo que hemos mencionado antes. Con la noción de soporte, además, cualquier síndrome que no tenga evidencia positiva tendrá un soporte de cero o negativo. Por otro lado, por la reevaluación constante de

ExpertCare, si esto resultara en un callejón sin salida, se revisaría la suposición contando con más información y obteniendo una nueva lista de candidatos.

Luego se calcula una nueva lista de pares con el sistema y su soporte a partir de los sistemas candidatos. Esta lista se ordena de mayor a menor por soporte y se devuelve.

```
systemsWithPositiveEvidence
    | answer dic |
    answer := OrderedCollection new.
    dic := collector syndromesBySystem.
    dic keysAndValuesDo: [:sys :syndromes |
      (syndromes anySatisfy: [:syn |
        syn symptoms anySatisfy: [:obs |
          (session answerFor: obs symptom) value =
obs value
            and: [obs symptom isQuantified or: [obs
value]]]])
        ifTrue: [answer add: sys]].
    ^answer
```

Se analiza un diccionario donde las claves son los sistemas y los valores, las colecciones de síndromes de cada sistema. Este diccionario es provisto por el `collector` auxiliar y se recorre, incluyendo en la colección de respuesta los sistemas que cumplan con la condición encerrada en un doble `anySatisfy:` que indica que se verifique si algún síntoma de algún síndrome del sistema que se está recorriendo la cumple. La condición es que haya una respuesta en la sesión sobre el mismo síntoma, y esa respuesta sea igual a lo que indica la definición y además, sea verdadero o un valor cuantificado.

```
supportOfSystem: aSystem
    | syndromes |
    syndromes := collector syndromesOfSystem:
aSystem.
    ^self supportOfSyndromes: syndromes
```

El soporte de un sistema se calcula pidiendo al `collector` auxiliar los síndromes de ese sistema y calculando luego el soporte de ese conjunto de síndromes con un método más general.

Lo primero es chequear si el conjunto está vacío y en ese caso consideramos que tiene un soporte negativo, devolviendo -1 por convención. Podríamos haber devuelto cero, que parece más adecuado como definición de soporte de un conjunto vacío, pero optamos por definirlo como negativo para que, comparado con cualquier conjunto no vacío, el vacío tenga menor soporte. Esto refleja la idea más general de que no es verosímil (no debe tener soporte) que no haya un síndrome finalmente elegido en la sesión, por lo que el conjunto vacío no es un candidato viable.

El método utiliza un `solver` auxiliar, que es un *visitor* sobre la jerarquía de expresiones lógicas que permite hacer la evaluación parcial de las definiciones en un contexto, devolviendo `true`, `false` o la expresión que queda sin resolver. Recorre

todos los síndromes del conjunto, evaluando con este `solver` la definición de cada uno.

Aquí empieza un análisis de casos. Los casos especiales son aquellos en que la definición ya se puede resolver en `true` o `false`. En estos casos se suma al total y a la cuenta de positivo o negativo, según corresponda, un aporte de 5 veces el puntaje de la severidad del síndrome. Esto es para dar mayor peso a los síndromes de mayor severidad. Los puntajes de cada severidad son provistos por el `collector` y son de 1, 2 y 3 para las severidades `green`, `yellow` y `red`. O sea que el aporte que se suma para un síndrome ya resuelto, sea positiva o negativamente, es de 5, 10 o 15.

En el caso restante sólo sabemos que todavía no se puede resolver el valor de verdad de la definición con la información existente. En este caso se recorren todos los síntomas mencionados en la definición y se suma 1 al total por cada síntoma. El síntoma influye en el soporte si fue preguntado, así que se analiza la respuesta para ese síntoma en la sesión. Si la respuesta existe y coincide con la definición (es la misma, o en el caso de un síndrome cuantificado, su valor está dentro del rango de la definición) suma un aporte positivo de 1, mientras que en otro caso el aporte negativo es de ½. Esto da mayor peso a la evidencia positiva que a la negativa.

Finalmente se restan los aportes negativos de los positivos y se divide por el total, asegurando que el número final queda entre -1 y 1.

Esta medida es ingenua y es empírica. Está orientada por nuestras ideas sobre el problema y la adoptamos en tanto resulta efectiva. Es simplemente una función heurística posible, adecuada para comparar valores en un conjunto de síndromes y fácil de calcular. Se podría intentar hacer un ajuste fino de cada uno de los números involucrados, pero dado el contexto en que la usamos, no creemos que eso ocasione mucha variación en los resultados.

Con esto completamos el análisis de cómo se calcula el soporte de los sistemas con evidencia positiva. Vamos al siguiente paso para obtener un sistema candidato:

A partir de la colección de pares, en este caso de sistema y soporte, se busca construir en `candidates` una lista de los síntomas que es interesante preguntar. Inicia un ciclo mientras no haya candidatos, del que puede salir por varios casos especiales.

El primer caso especial es definitivo, es que la lista venga vacía. Esto querría decir que no hay nada que valga la pena analizar y se devuelve una respuesta nula, indicando en la historia con un par nulo que se llegó a esta situación.

El siguiente caso es cuando en la lista de soporte hay un único elemento, o cuando hay una diferencia muy grande entre el primer elemento y el siguiente. Esto se decide usando valores umbral que también tienen una base empírica. Consideramos que la diferencia es muy grande si el soporte del primer elemento tiene más diferencia que 0.35 (recordemos que son valores con módulo entre 0 y 1) o si es más de 250 veces más grande. En estos casos consideramos que el sistema está decidido y no vale la pena considerar más que el primero (que registramos en la variable de instancia `system`), y vamos a la siguiente etapa que es tratar de decidir una severidad en `nextSymptomLookingForSeverity`.

En los otros casos vale la pena analizar los dos primeros sistemas, apuntando a encontrar un síntoma que nos permita decidir entre ellos. Podría pasar que no quede ningún síntoma, por haber sido todos preguntados. En ese caso pasaríamos a trabajar con sistemas de soporte más bajo pero que todavía tengan candidatos.

El último paso para determinar el síntoma a preguntar cuando no se sabe el sistema candidato, es calcular la diferencia de soporte que puede provocar cada uno de los candidatos.

Se recorre la lista de síntomas candidatos y por cada uno se guarda un arreglo de cuatro posiciones: en la primera va el síntoma, en la segunda y la tercera la variación de soporte que produce en los conjuntos de síndromes 1 y 2 respectivamente (recordemos que en este punto, en estas colecciones estarían los síndromes del sistema 1 y 2), y en la última, la suma de ambas variaciones. Para devolver una respuesta, se ordenan estos arreglos por la última posición, que tiene la suma de ambas variaciones, con la idea de que se pueda escoger el síntoma que más produce mayor variación considerando las dos respuestas igualmente posibles. Se desempata por nombre en el orden si producen la misma variación, simplemente para asegurar algún orden repetible.

Para calcular las diferencias de soporte generadas por los candidatos se usan dos `solvers` auxiliares en lugar de uno. Uno de ellos "simula" una respuesta positiva para el síntoma indicada con `positive:` y el otro, `solver_`[12] usa `negative:` para considerar una respuesta negativa al mismo síntoma. Con esas consideraciones, se calculan dos juegos con las mismas variables `positive`, `negative` y `total` usadas en el cálculo del soporte, duplicadas en las que llevan un *underscore* al final, para el caso negativo. El cálculo se limita únicamente a los síndromes en los que el síntoma analizado tenga algo que ver, y únicamente computa el valor relacionado con este síntoma, ya que lo que estamos buscando es la variación que provoca y no el soporte total. Finalmente se consolidan las diferencias, restando el total del caso negativo del positivo y devolviendo el valor absoluto de la diferencia.

De esta manera hemos cubierto todas las alternativas para el caso en que no tenemos predilección por un sistema candidato, que es justamente el caso inicial.

El segundo caso general que esta estrategia trata por separado es el de estar analizando un sistema candidato determinado, y tratar de decidir una severidad. Es muy similar al de elegir un sistema candidato, sólo que en este caso nos limitamos al espacio de los síndromes del sistema que nos interesa y las únicas alternativas son las tres severidades. El resto del procedimiento es análogo a la búsqueda de síntoma cuando no se conoce el sistema.

[12] Por una limitación del parser, no pudimos utilizar solver- (solver negativo) o –solver, que hubieran sido nombres más adecuados.

Resultados de SupportSeparation

Veamos los resultados de esta estrategia, comparados con el primer grupo:

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
Sequential	739,04	745	19,00	5,58
Random	746,58	732,5	18,05	5,46
LessSatisfiers	726,28	932	18,84	0,00
MiddleSatisfiers	579,28	794	18,05	2,44
MoreSatisfiers	85,29	52	14,23	2,51
MoreCriticSeparation	174,44	35	18,29	7,13
SupportSeparation	6,09	3	15,32	5,58

Para los síndromes de máxima gravedad, se puede ver claramente la enorme diferencia en cantidad de preguntas. Por otra parte, los errores no suben de manera significativa. Es bueno resaltar que, por primera vez, alcanzamos una mediana de preguntas que cumple con nuestro objetivo, aunque el promedio es todavía un poco alto.

Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
Sequential	961,12	975	9,61	5,19
Random	1001,61	1067	16,76	9,72
LessSatisfiers	662,00	730,5	0,00	0,00
MiddleSatisfiers	460,41	568	3,13	0,00
MoreSatisfiers	92,82	43	7,75	4,89
MoreCriticSeparation	215,57	122	11,72	5,04
SupportSeparation	11,75	6	9,61	3,91

En los síndromes de severidad *yellow* se comprueba también una gran diferencia, aunque en este caso la cantidad de preguntas todavía está por encima de nuestro objetivo. Nuevamente las tasas de error son comparables a las del primer grupo.

Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
Sequential	1049,18	1245	17,72	9,85
Random	1099,11	1359	17,75	9,91
LessSatisfiers	561,03	526	11,76	0,00
MiddleSatisfiers	764,34	951	25,93	1,39
MoreSatisfiers	259,46	220	14,62	4,71
MoreCriticSeparation	301,97	266,5	13,21	4,60
SupportSeparation	25,43	8	18,50	9,97

En los síndromes de severidad *green* se ve algo similar. La mediana de la cantidad de preguntas está dentro del rango aceptable pero el promedio está bastante por encima. Las tasas de error son comparables a las del primer grupo.

Cabe aclarar que en este caso, por ser un número manejable de errores y para poder estar seguros de la confiabilidad de los resultados, verificamos automáticamente que en la totalidad de los errores de diagnóstico, el diagnóstico elegido por la estrategia era implicado por el síndrome elegido para el *answerProvider*. Repetimos la comprobación manualmente en todos los casos de severidad *red* también. De este modo, estamos en condiciones de afirmar que estas tasas de error no constituyen un auténtico error de la estrategia, sino que son una consecuencia de las definiciones en la base de conocimiento y de la mecánica de automatización de las pruebas.

Estos resultados fueron sumamente alentadores, confirmando la creencia de que estábamos en el camino correcto con las ideas que llevaron a la implementación de esta estrategia. Pero están lejos de ser suficientes como para dar el trabajo por terminado.

Para las siguientes estrategias, dejaremos de lado las comparaciones con el primer grupo y tomaremos como base de comparación los resultados de *SupportSeparation*.

Vamos a exponer las otras estrategias, que derivaron de la noción de separación de soporte y son sucesivos intentos de mejorar la básica de *SupportSeparation*, en un orden aproximadamente cronológico que refleja la manera en que se nos fueron ocurriendo ideas para mejorar y cómo fue creciendo la jerarquía de clases de estas estrategias.

Estrategias de *SupportImplication*, *Tracking* y *Closing*

La siguiente idea que surgió fue la de intentar aprovechar la información de implicación entre los síntomas. Hay una relación de implicación cuya información está incluida en la base de conocimientos y que hasta ahora no habíamos utilizado para nada. Así, pues, la siguiente estrategia se llama `SupportWithImplicationStrategy` y es una subclase directa de `SupportSeparationStrategy`, de la que hereda la mayor parte del comportamiento.

La forma más sencilla que se nos ocurrió para utilizar la información de implicación fue hacer que cada vez que teníamos una respuesta positiva para un síntoma, se anotaran respuestas positivas también para todos los síntomas implicados por él, ya que esto es una consecuencia lógica de la definición de implicación, una aplicación directa de la regla de *modus ponens*. Del mismo modo, si tenemos una respuesta negativa, anotamos una respuesta negativa para todos los síntomas que implican a éste ya que ninguno puede estar presente si implica un síntoma que no está. Estas reglas no se aplican a los síntomas cuantificados.

Para lograr esto, la manera más sencilla fue crear una subclase de `CallSession` que hiciera exactamente eso e hicimos el cambio correspondiente en la estrategia. Cuando se calculan las diferencias de soporte evaluando posibles respuestas positivas y negativas para el síntoma actual, se utiliza la clausura transitiva de la relación de implicación correspondiente. Es equivalente a lo que pasaría si en la sesión se incluyera cada respuesta de la clausura.

Estos son los únicos cambios necesarios para utilizar consistentemente la información de implicación entre los síntomas.

Otra idea que se nos ocurrió fue la de tratar de no ampliar el espacio de búsqueda. Intentamos mantenernos en la lista de candidatos que inicialmente tenían evidencia positiva, en lugar de reevaluar y obtener una nueva lista de candidatos cada vez. Así nació la estrategia `SupportImplicationTracking`. Le pusimos *tracking* por la

idea de ir llevando registro (*keeping track*) de los candidatos y de ir siguiendo la misma pista (*track*) inicial.

En la implementación básica de la superclase, al principio se obtiene la lista de candidatos evaluando cada vez `#systemsWithPositiveEvidence`, mientras que aquí se utiliza sólo la primera vez. Las siguientes veces no vuelve a calcular la lista, cosa que daría un conjunto posiblemente mayor de sistemas, sino que sigue refinando sobre la lista inicial.

Otra idea que surgió de las primeras observaciones de los resultados fue la de forzar el cierre de la sesión si sólo queda un síndrome candidato. Esto puede resultar exageradamente optimista en ocasiones, pero quisimos probarlo ya que en muchos casos, por la condición de cierre de la sesión, el sistema se quedaba haciendo varias preguntas adicionales sólo para confirmar un síndrome candidato que ya tenía "decidido". Para evitar esto, implementamos la `SupportImplicationtrackingClosingStrategy`. Aquí se evita la falsa competencia entre definiciones alternativas del mismo síndrome, que al considerarse como *subsíndromes* separados, pueden quedar como si fueran distintos candidatos competidores.

Esta nueva estrategia hereda de la anterior y por eso su nombre se construyó por agregación, sumando el *Closing* que indica que cierra cuando se queda con un solo candidato.

Cambia la condición de finalización de la estrategia, para indicar que termina si encontró este candidato único. En el momento en que se busca un candidato para decidir entre síndromes, si resulta que todos los candidatos corresponden al mismo síndrome, se considera que tenemos al *candidato único*, y lo colocamos en la variable `chosen`, indicando finalización porque no tiene sentido seguir haciendo más preguntas.

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	6,09	3	15,32	5,58
SupportSeparationWithImplication	4,62	2	18,29	7,13
SupportSeparationImplicationTracking	3,98	2	15,32	5,58
SupportSeparationImplicationTrackingClosing	2,95	1	16,86	7,13

Vemos en *red* que las tres nuevas estrategias representan mejoras sobre la cantidad de preguntas con respecto a la básica de *SupportSeparation*, aunque las de *Implication* y *Closing* aumentan un poco la tasa de error.

Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	11,75	6	9,61	3,91
SupportSeparationWithImplication	8,53	6	16,76	9,72
SupportSeparationImplicationTracking	7,31	5	10,33	4,16
SupportSeparationImplicationTrackingClosing	5,94	4	16,25	10,39

Para *yellow* vemos algo similar, aunque las modificaciones del error se amplifican.

Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	25,43	8	18,50	9,97
SupportSeparationWithImplication	17,67	8	23,55	10,66
SupportSeparationImplicationTracking	16,54	6	18,11	9,97
SupportSeparationImplicationTrackingClosing	5,79	1	19,01	16,43

Para *green* tenemos un efecto similar aunque, sorprendentemente, en el caso de la estrategia *Closing* la cantidad de preguntas se reduce mucho más, pero aumenta el error de manera considerable. Esto refleja que quizá en el caso de síndromes de esta severidad, es peligroso el optimismo de cerrar cuando queda un sólo candidato ya que posiblemente esto lleve a diagnosticar una severidad más alta. Sin duda, esto está ligado a características de la base de conocimiento y en particular a las definiciones de los síndromes.

Recordemos que el sistema en general ya tiene (en el criterio de corte de la sesión) un sesgo que favorece diagnosticar anticipadamente severidades altas. Sumarle a esto otro efecto en el mismo sentido dentro de la estrategia, no aprece una buena idea. Recordemos también que, si bien un falso negativo de urgencia es terrible porque casi con seguridad implica la muerte del paciente, un falso positivo no deja de tener consecuencias. Al ser las ambulancias un recurso finito (y más bien escaso), dedicarlas a atender casos que no son urgencias puede hacer que no estén disponibles para una urgencia real.

Nuevamente, tenemos resultados que en conjunto son alentadores pero todavía no son suficientes para satisfacer nuestros objetivos. La estrategia con implicación mejora un poco la cantidad de preguntas, pero trae aparejado un aumento de los errores. La estrategia *TrackingClosing* parece excesivamente optimista en el caso *green*, aumentando mucho el error, aunque también es la que reduce más la cantidad de preguntas. La de *Tracking* es consistentemente mejor en cantidad de preguntas, y sólo aumenta un poco el error en los casos de severidad *yellow*.

Seguimos explorando otras ideas y elaborando nuevas estrategias.

Estrategias MainSyndrome y MainScoringSyndrome

La idea de evitar la falsa competencia entre distintos *subsíndromes* que en realidad representan el mismo síndrome nos pareció interesante, y por ello decidimos intentar otra aproximación en la estrategia *MainSyndrome*. Lo que intentamos aquí es calcular la variación de soporte unificando el aporte de los distintos *subsíndromes*. Concretamente, si un síntoma aparece repetido en varias definiciones alternativas, por el efecto multiplicador de una conjunción, sólo lo contabilizamos una vez.

Otra idea que surgió durante estas pruebas fue la de utilizar un esquema de *scoring* más similar al que implementa ExpertCare. Aunque no tiene una relación definida con la idea anterior, implementamos esta idea en una subclase de la estrategia *MainSyndrome*, que se llama *MainScoringSyndrome* y combina las dos cosas.

El uso del *scoring* también va relacionado con un cambio en la condición de cierre de una sesión. Para hacerlo más similar al manejo de ExpertCare, la sesión se corta si ninguno de los síndromes todavía no decididos que pueden completarse a partir de los completos, puede superar en puntaje a los ya cerrados. Dicho de otra manera, si hay

oportunidad de que un síndrome de mayor puntaje se complete, la sesión continúa para ver si esto sucede.

Así, hicimos una nueva subclase de `CallSession` llamada `CallSessionScoring` que implementa esta variación.

El puntaje de un síndrome se define simplemente como la suma de los puntajes de los síntomas que aparecen en su definición.

En la estrategia, el único cambio es usar el puntaje calculado por la sesión en las partes donde se utilizaba el puntaje indicado por la severidad.

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	6,09	3	15,32	5,58
SupportSeparationWithImplication	4,62	2	18,29	7,13
SupportSeparationImplicationTracking	3,98	2	15,32	5,58
SupportSeparationImplicationTrackingClosing	2,95	1	16,86	7,13
SupportMainSyndrome	3,96	2	15,32	5,58
SupportMainSyndromeScoring	5,41	3	15,20	5,70

Si tomamos como base la *SupportSeparation* podemos ver, en los síndromes de severidad *red*, que la idea del síndrome principal acarrea una pequeña mejora en las preguntas, sin variación de error; mientras que el uso del puntaje sólo produce efectos marginales en ambas cosas. No hay mejoras interesantes con respecto a las estrategias que usan la implicación.

Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	11,75	6	9,61	3,91
SupportSeparationWithImplication	8,53	6	16,76	9,72
SupportSeparationImplicationTracking	7,31	5	10,33	4,16
SupportSeparationImplicationTrackingClosing	5,94	4	16,25	10,39
SupportMainSyndrome	7,18	5	10,95	4,47
SupportMainSyndromeScoring	7,29	6	10,80	4,88

En *yellow*, *MainSyndrome* sigue mejorando un poco la cantidad de preguntas respecto de *SupportSeparation*, pero ahora con un error algo mayor. La estrategia que usa el puntaje también sube el error y varía poco el promedio de preguntas, sin llegar a cambiar la mediana. Tampoco mejora los resultados de las estrategias con uso de implicación, más bien se ve una desmejora.

Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	25,43	8	18,50	9,97
SupportSeparationWithImplication	17,67	8	23,55	10,66
SupportSeparationImplicationTracking	16,54	6	18,11	9,97
SupportSeparationImplicationTrackingClosing	5,79	1	19,01	16,43
SupportMainSyndrome	16,46	6	18,20	9,97
SupportMainSyndromeScoring	11,92	6	19,65	10,00

En *green* se observa un resultado similar. Con respecto a *SupportSeparation*, mejora consistente en las preguntas para *MainSyndrome* y mejora en preguntas pero aumento del error en *Scoring*. Con respecto a las estrategias que usan implicación, igual o un poco peor.

En síntesis, la idea del síndrome principal mejora *SupportSeparation* y parece tener un efecto comparable a las de implicación. Por otro lado, el uso de *Scoring* no mejora los resultados de manera significativa.

Estrategias *OnlyPositive* y *OnlyPositiveClosing*

Pensando sobre la idea que dio origen a la estrategia *ImplicationTracking*, la de tomar sólo los candidatos positivos y tratar de no salir de ese conjunto original sino refinarlo, quisimos profundizarla en las estrategias *OnlyPositive*. La primera consideración fue que habíamos aplicado eso sólo para obtener los sistemas candidatos, pero después considerábamos todos los síndromes dentro de los sistemas y severidades candidatos, tuvieran o no evidencia positiva. La estrategia *OnlyPositive* es una subclase de `SupportMainSyndromeStrategy` y toma, para calcular diferencias de soporte entre candidatos, únicamente los síndromes con evidencia positiva.

Luego quisimos combinarlo con la idea de *Closing* de evitar la falsa competencia entre *subsíndromes* del mismo síndrome. Del mismo modo que ya la habíamos combinado con *Tracking*, hicimos una subclase de `SupportOnlyPositiveStrategy` llamada `SupportOnlyPositiveClosingStrategy` que modifica la condición de finalización de la estrategia como vimos antes.

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	6.09	3	15.32	5.58
SupportSeparationImplicationTracking	3.98	2	15.32	5.58
SupportSeparationImplicationTrackingClosing	2.95	1	16.86	7.13
SupportMainSyndrome	3.96	2	15.32	5.58
SupportOnlyPositive	3.34	2	15.20	5.11
SupportOnlyPositiveClosing	2.40	1	16.86	6.77

Observando los resultados de los síndromes de severidad *red*, la estrategia *OnlyPositive* representa una mejora consistente, tanto en cantidad de preguntas como en errores. Sólo *TrackingClosing* tiene menos preguntas, pero mayor error. En situación similar se encuentra la nueva *OnlyPositiveClosing*.

En estos cuadros se suprimieron algunas de las estrategias anteriores, para dejar sólo las más interesantes.

Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	11.75	6	9.61	3.91
SupportSeparationImplicationTracking	7.31	5	10.33	4.16
SupportSeparationImplicationTrackingClosing	5.94	4	16.25	10.39
SupportMainSyndrome	7.18	5	10.95	4.47
SupportOnlyPositive	5.11	4	10.69	4.27
SupportOnlyPositiveClosing	4.24	3	12.08	5.91

Los resultados para los síndromes de severidad *yellow* apuntan en la misma dirección, aunque la tasa de error de *OnlyPositive* es un poco más alta que la de *SupportSeparation*.

Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	25.43	8	18.50	9.97
SupportSeparationImplicationTracking	16.54	6	18.11	9.97
SupportSeparationImplicationTrackingClosing	5.79	1	19.01	16.43
SupportMainSyndrome	16.46	6	18.20	9.97
SupportOnlyPositive	13.44	5	18.08	10.00
SupportOnlyPositiveClosing	5.18	1	18.34	10.48

En los resultados de síndromes de severidad *green*, todas las diferencias se acentúan. La mejora de *OnlyPositive* respecto de *SupportSeparation* es mayor que en los otros grupos, pero el cierre anticipado de las estrategias *Closing* da más frutos y en el caso de *OnlyPositiveClosing*, la diferencia en el error es baja.

En resumen, ambas estrategias están dentro de los parámetros aceptables de nuestros objetivos y representan importantes mejoras respecto de *SupportSeparation*, con un balance entre cantidad de preguntas y tasa de error mejores que las estrategias anteriores. Pero esto nos dio nuevas ideas para probar y seguir intentando mejoras más profundas.

Estrategias derivadas de *OnlyPositiveClosing*

Una idea que consideramos desde las primeras estrategias de soporte era variar la regla de composición que usamos para ordenar los síndromes de acuerdo a las diferencias que producen en los dos grupos de síndromes candidatos. Hasta el momento, sólo buscábamos maximizar la suma de las diferencias entre los dos grupos, pero hay otras posibilidades: podíamos utilizar el candidato que provocara mayor variación en alguno de los dos, o el que produjera la mayor diferencia en las diferencias (valga la redundancia) entre ambos grupos.

Llegados a este punto en que contamos con estrategias de *performance* aceptable y bien conocida, decidimos hacer un pequeño ensayo, utilizando la diferencia entre las diferencias como forma de ordenar los candidatos. Lo limitamos a los casos en que había que decidir entre dos severidades dentro del mismo sistema, por algunos indicios de que en varios casos estábamos "desperdiciando" preguntas en esa etapa. Así surgió la estrategia `SupportOnlyPositiveClosingDifSevStrategy`, una subclase de

`SupportOnlyPositiveClosingStrategy` que no agrega ninguna variable de instancia e implementa sólo dos métodos para lograr el efecto que describimos arriba.

Por la forma en que se realizó esta implementación, sólo surte algún efecto cuando hay que decidir el síntoma candidato entre dos grupos de síndromes de distinta severidad dentro del mismo sistema, con miras a desempatar entre las dos severidades y elegir una como candidato, para luego continuar la búsqueda del síndrome.

Otra idea que quisimos explorar no era nueva, sino una forma de seguir avanzando en la misma dirección que la estrategia *OnlyPositive*. Allí la idea fue quedarse únicamente con los síndromes con evidencia positiva para analizar los grupos, sin considerar más aquellos síndromes que no la tenían para no ampliar los grupos iniciales y no dispersar el esfuerzo analizando casos sin soporte cuando hay otros mucho más interesantes. Pero por otro lado, una vez decididos por un grupo, la lista de síntomas candidatos se armaba con las definiciones de todos los miembros del grupo (tuvieran o no evidencia positiva). Un paso lógico es filtrar estas listas de síntomas candidatos únicamente a los síntomas de las definiciones de síndromes con evidencia positiva, concentrando aún más nuestros esfuerzos.

Cada uno de estos pasos fue dado con cierta vacilación, ya que de alguna manera se reduce el espacio y "dejamos de mirar" posibles candidatos. Siempre está el "temor" de cerrar prematuramente caminos, focalizar demasiado y terminar en un callejón sin salida. El argumento decisivo es, como en ExpertCare, proveer mecanismos que aseguren una re-evaluación completa si se llega a esa situación. Así, nunca hay verdaderos callejones sin salida.

Para explorar esta idea agregamos la estrategia `SupportOnlyPositiveStrictClosingStrategy`, otra subclase de `SupportOnlyPositiveClosingStrategy` que agrega la partícula *Strict* al nombre, para indicar un filtrado más estricto: no solamente de los síndromes, sino también de los síntomas. Nuevamente la implementación resulta muy sencilla y no requiere de ninguna variable de instancia adicional.

La última idea que exploramos en esta etapa va en la misma línea de reducir el espacio de candidatos a analizar y tratar de no ampliarlo posteriormente con la nueva información proveniente de las respuestas del paciente. Es parecida a la que acabamos de ver en *StrictClosing*, de hecho algunos métodos usan la misma implementación. Es otra subclase de `SupportOnlyPositiveClosingStrategy` a la que llamamos `SupportOnlyPositiveCandidatesClosingStrategy`, agregando la palabra *candidates* para indicar que pondremos foco exclusivamente en los candidatos con evidencia positiva.

El grupo de síndromes candidatos se obtiene conservando sólo aquellos que vienen de sistemas candidatos a su vez y filtrando nuevamente entre estos síndromes los que tienen evidencia positiva. El método que obtiene los síndromes con evidencia positiva se modifica para contemplar únicamente los miembros dentro de este grupo original. Así, el grupo inicial siempre va reduciéndose, y nunca se agrega un candidato fuera de ese grupo (a menos que quede vacío del todo y se fuerce una re-evaluación).

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	6.09	3	15.32	5.58
SupportSeparationImplicationTracking	3.98	2	15.32	5.58
SupportSeparationImplicationTrackingClosing	2.95	1	16.86	7.13
SupportMainSyndrome	3.96	2	15.32	5.58
SupportOnlyPositive	3.34	2	15.20	5.11
SupportOnlyPositiveClosing	2.40	1	16.86	6.77
SupportOnlyPositiveCandidatesClosing	1.85	1	16.50	6.41
SupportOnlyPositiveClosingDifSev	2.12	1	20.55	6.29
SupportOnlyPositiveStrictClosing	2.13	1	16.86	6.77

En los resultados para los síndromes de severidad *red* podemos ver que estas tres estrategias representan una pequeña mejora en cantidad de preguntas con respecto a las demás, aunque sube un poco el error. La de *StrictClosing* presenta un comportamiento muy similar a *OnlyPositiveClosing*.

Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	11.75	6	9.61	3.91
SupportSeparationImplicationTracking	7.31	5	10.33	4.16
SupportSeparationImplicationTrackingClosing	5.94	4	16.25	10.39
SupportMainSyndrome	7.18	5	10.95	4.47
SupportOnlyPositive	5.11	4	10.69	4.27
SupportOnlyPositiveClosing	4.24	3	12.08	5.91
SupportOnlyPositiveCandidatesClosing	3.15	2	11.21	5.19
SupportOnlyPositiveClosingDifSev	4.65	3	12.90	6.17
SupportOnlyPositiveStrictClosing	3.61	2	11.52	5.40

El panorama es similar para los síndromes de severidad *yellow*. Se ve mayor diferencia en la cantidad de preguntas y menor en los errores. El parecido entre *StrictClosing* y *OnlyPositiveClosing* se atenúa.

Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	25.43	8	18.50	9.97
SupportSeparationImplicationTracking	16.54	6	18.11	9.97
SupportSeparationImplicationTrackingClosing	5.79	1	19.01	16.43
SupportMainSyndrome	16.46	6	18.20	9.97
SupportOnlyPositive	13.44	5	18.08	10.00
SupportOnlyPositiveClosing	5.18	1	18.34	10.48
SupportOnlyPositiveCandidatesClosing	2.82	1	18.41	14.48
SupportOnlyPositiveClosingDifSev	5.23	1	18.11	9.70
SupportOnlyPositiveStrictClosing	3.39	1	18.14	13.82

En los casos de severidad *green* se destaca la estrategia *ClosingDifSev*, registrando la menor tasa de error de severidad hasta el momento y un error de diagnóstico relativamente bajo. Esto no sucedía en los otros dos grupos, en *yellow* tiene el error más alto de las tres. Su cantidad de preguntas no es tan buena como las de las otras dos, pero

de las estrategias anteriores sólo la supera la de *OnlyPositiveClosing*. Las otras dos registran las menores cantidades de preguntas en síndromes de severidad *green*.

En general las estrategias de este grupo representan una mejora sobre todas las anteriores, con resultados muy interesantes y dentro de los objetivos que nos habíamos propuesto. Pero nos quedaba la impresión todavía de que se podía mejorar, y teníamos más ideas para explorar.

Estrategia *LessMissingScore*

Una de las últimas ideas que ensayamos fue otro intento de utilizar el score. En este caso, buscamos reducir el grupo de candidatos considerando que los más prometedores serían aquellos síndromes a los que "les falta poco" para completar su definición. Esa noción informal la cuantificamos como el grupo de síndromes con menos puntaje faltante (de ahí el nombre de *less missing score*).

Para ello hicimos una subclase de `SupportOnlyPositiveCandidatesClosingStrategy` llamada `SupportLessMissingScoreStrategy`. Esta subclase no requiere ninguna nueva variable de instancia y sólo agrega seis métodos.

Se reimplementa la inicialización de los síndromes candidatos, agregando un filtro adicional. Antes eran todos los síndromes con evidencia positiva, ahora se registra sólo el grupo de mayor coincidencia entre ésos.

El grupo de mayor coincidencia se busca, como dijimos, computando el score faltante de los síndromes con evidencia positiva (que vienen como argumento en este método) y devolviendo el grupo que tenga el valor mínimo.

El cálculo del puntaje de cada síntoma se deriva a la sesión (que debe ser de la subclase `CallSessionScoring`). Ya hemos visto en otra estrategia cómo se realiza allí el cálculo del puntaje.

En cada nueva actualización de información por una nueva pregunta, también se filtra nuevamente el grupo de síndromes candidatos, utilizando el mismo criterio.

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	6.09	3	15.32	5.58
SupportOnlyPositiveClosing	2.40	1	16.86	6.77
SupportOnlyPositiveCandidatesClosing	1.85	1	16.50	6.41
SupportOnlyPositiveClosingDifSev	2.12	1	20.55	6.29
SupportOnlyPositiveStrictClosing	2.13	1	16.86	6.77
SupportLessMissingScore	0.22	0	57.84	39.55

Los resultados para los síndromes de severidad *red* muestran que el criterio utilizado es demasiado agresivo. La cantidad de preguntas que la estrategia considera necesarias para decidir es muy baja, pero casi siempre este cierre resulta ser prematuro y equivocado, subiendo la tasa de error a un margen inaceptable.

Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	11.75	6	9.61	3.91
SupportOnlyPositiveClosing	4.24	3	12.08	5.91
SupportOnlyPositiveCandidatesClosing	3.15	2	11.21	5.19
SupportOnlyPositiveClosingDifSev	4.65	3	12.90	6.17
SupportOnlyPositiveStrictClosing	3.61	2	11.52	5.40
SupportLessMissingScore	0.48	0	55.89	35.63

Los resultados para el grupo de severidad *yellow* indican exactamente lo mismo.

Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	25.43	8	18.50	9.97
SupportOnlyPositiveClosing	5.18	1	18.34	10.48
SupportOnlyPositiveCandidatesClosing	2.82	1	18.41	14.48
SupportOnlyPositiveClosingDifSev	5.23	1	18.11	9.70
SupportOnlyPositiveStrictClosing	3.39	1	18.14	13.82
SupportLessMissingScore	0.43	0	38.97	13.91

Entre los síndromes de severidad *green* los resultados tampoco mejoran.

En conjunto, debemos concluir que esta estrategia no es una buena alternativa, ya que llega muy rápidamente a conclusiones que da por buenas, pero con alto índice de error.

Estrategia de *MoreCoincidences*

Para producir la siguiente mejora, analizamos algunos casos en los que las estrategias más exitosas hacían gran cantidad de preguntas. Encontramos que, con cierta frecuencia, la estrategia se "despistaba" analizando dos candidatos que consideraba parejos (porque los cálculos de diferencia de soporte daban igual), pero de los cuales uno tenía más coincidencias en los síntomas de la definición con los declarados por el paciente.

Resultó muy sencillo elaborar una nueva estrategia, a partir de las experiencias anteriores, que considerara como grupo de síndromes candidatos a aquellos que tuvieran el mayor número de coincidencias (de ahí el nombre *more coincidences*) con los síntomas confirmados en la sesión. Hicimos una nueva subclase de `SupportOnlyPositiveCandidatesClosingStrategy` llamada `SupportMoreCoincidenceStrategy`, que no requiere ninguna variable de instancia adicional y sólo implementa cuatro métodos.

De manera similar a la de *LessMissingScore*, se agrega un filtro adicional en la inicialización de la lista de síndromes candidatos, para quedarse con los que registran mayor número de coincidencias entre aquellos síndromes que tienen evidencia positiva. Pero en esta estrategia definimos de manera distinta el cálculo (y la semántica) de "síndromes con más coincidencias". En *LessMissingScore* llamábamos así a los que les faltara menor puntaje para completarse; aquí procedemos de una forma que podríamos considerar opuesta: tomamos los que más síntomas presentes tengan en su definición.

Finalmente, en cada actualización de información desde la sesión, se vuelve a calcular la lista de síndromes candidatos eligiendo los que registran más coincidencias con las respuestas del paciente.

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	6,09	3	15,32	5,58
SupportOnlyPositiveClosing	2,40	1	16,86	6,77
SupportOnlyPositiveCandidatesClosing	1,85	1	16,50	6,41
SupportOnlyPositiveClosingDifSev	2,12	1	20,55	6,29
SupportOnlyPositiveStrictClosing	2,13	1	16,86	6,77
SupportMoreCoincidences	1,65	1	16,50	6,41

Podemos ver en el grupo de los síndromes de severidad *red* que la nueva estrategia resulta efectiva en cuanto a reducir el promedio de preguntas. La mediana no cambia, indicando que actúa principalmente sobre los casos que antes daban una cantidad de preguntas más alta que el promedio. Las tasas de error son un poco más altas que las de *SupportSeparation*, pero están en el grupo de los errores más bajos entre las mejoras posteriores.

Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	11,75	6	9,61	3,91
SupportOnlyPositiveClosing	4,24	3	12,08	5,91
SupportOnlyPositiveCandidatesClosing	3,15	2	11,21	5,19
SupportOnlyPositiveClosingDifSev	4,65	3	12,90	6,17
SupportOnlyPositiveStrictClosing	3,61	2	11,52	5,40
SupportMoreCoincidences	2,63	2	11,47	5,45

Exactamente la misma tendencia se ve en el grupo de síndromes de severidad *yellow*.

Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	25,43	8	18,50	9,97
SupportOnlyPositiveClosing	5,18	1	18,34	10,48
SupportOnlyPositiveCandidatesClosing	2,82	1	18,41	14,48
SupportOnlyPositiveClosingDifSev	5,23	1	18,11	9,70
SupportOnlyPositiveStrictClosing	3,39	1	18,14	13,82
SupportMoreCoincidences	2,12	1	18,29	14,12

Hay una leve diferencia en el grupo mayoritario de síndromes con la menor severidad. Aquí la tasa de error en la severidad sube de manera apreciable, aunque no pasa lo mismo con la cantidad de errores en diagnósticos. Esto indicaría un mayor número de falsas alarmas; un mayor porcentaje de los errores son atribuidos a síndromes de severidad mayor de la que corresponde.

En conjunto, podemos decir que esta estrategia representa una mejora efectiva en la dirección en que fue pensada, reduciendo la cantidad de preguntas promedio al

cortar antes algunos casos que antes oscilaban entre dos candidatos, usando un criterio que parece acertado.

Estrategias derivadas de *MoreCoincidences*

Varias de las estrategias de la última serie se basan en aplicar filtros adicionales a la lista de síndromes candidatos. El efecto de esto es que después de unas pocas preguntas, en general quedan pocos síndromes que hayan pasado los filtros. En estas condiciones, ya no tiene tanto sentido pasar primero por las etapas de elegir un sistema y una severidad candidatos. La idea de esa decisión era limitar el conjunto de síndromes y síntomas a analizar, pero esto ya se logra con los otros filtros.

De hecho, estas reflexiones surgieron de observar en varias ejecuciones paso por paso que las estrategias usaban varias preguntas para decidir entre dos severidades, cuando en realidad sólo había dos síndromes candidatos, uno de cada severidad.

Entonces, pensamos una nueva estrategia definida en estos términos: aplicar *MoreCoincidences* como siempre, con las tres etapas "clásicas" de buscar candidatos para sistema, severidad y síndrome, pero si en algún caso intermedio detectamos que la cantidad de síndromes candidatos es menor que un umbral (por ejemplo, tres o cuatro), decidimos "saltar etapas" y pasar a través de todas, para decidir únicamente entre síndromes. Esto implica que la decisión entre síndromes puede tener ahora competidores de distinta severidad o sistema, algo que hasta ahora no sucedía. Pero no sucedía únicamente por la forma en que nosotros planteamos las estrategias, nada en la implementación lo impide.

Hicimos una subclase de `SupportMoreCoincidenceStrategy` llamada `SupportMoreCoincidencePassThruStrategy` a la que agregamos una variable de instancia para usar como parámetro, el valor umbral de la decisión que indica a partir de cuántos síndromes consideramos que es mejor pasar a través (*pass thru*) de las etapas pre-establecidas y hacer la decisión directamente dentro del conjunto de síndromes candidatos.

Cabe aclarar que, de acuerdo a la forma en que definimos la estrategia, un valor umbral de 1 la haría totalmente equivalente a *MoreCoincidences*, ya que sólo saltearía etapas si la cantidad de síndromes candidatos fuera 1, y ya sabemos que en ese caso la estrategia termina por acción de otra regla.

Strategy	Red			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	6,09	3	15,32	5,58
SupportOnlyPositive	3,34	2	15,20	5,11
SupportOnlyPositiveClosing	2,40	1	16,86	6,77
SupportOnlyPositiveCandidatesClosing	1,85	1	16,50	6,41
SupportOnlyPositiveClosingDifSev	2,12	1	20,55	6,29
SupportOnlyPositiveStrictClosing	2,13	1	16,86	6,77
SupportMoreCoincidences	1,65	1	16,50	6,41
SupportMoreCoincidencesPassThru_1	1,65	1	16,50	6,41
SupportMoreCoincidencesPassThru_2	1,65	1	16,50	6,41
SupportMoreCoincidencesPassThru_3	1,96	1	15,32	5,23
SupportMoreCoincidencesPassThru_4	2,12	1	15,20	5,10
SupportMoreCoincidencesPassThru_5	2,13	1	15,20	5,11
SupportMoreCoincidencesPassThru_6	2,19	1	15,08	5,11
SupportMoreCoincidencesPassThru_7	2,14	1	15,08	5,11
SupportMoreCoincidencesPassThru_8	2,11	1	15,08	5,11
SupportMoreCoincidencesPassThru_9	2,09	1	15,08	5,11
SupportMoreCoincidencesPassThru_10	2,05	1	15,08	5,11

Aquí vemos el desempeño de esta nueva estrategia con distintos valores del parámetro umbral de cantidad de síndromes, indicados en el número final.

Podemos comprobar que, efectivamente, usar el valor de 1 da los mismos resultados que en la estrategia *MoreCoincidences*. Entre los síndromes de severidad *red*, de hecho, usar el parámetro 2 tampoco provoca variación. En los siguientes parece delinearse una tendencia a subir el promedio de preguntas y bajar el error, pero los valores fluctúan bastante. Se podría escoger el mejor entre el tres y cuatro en este caso.

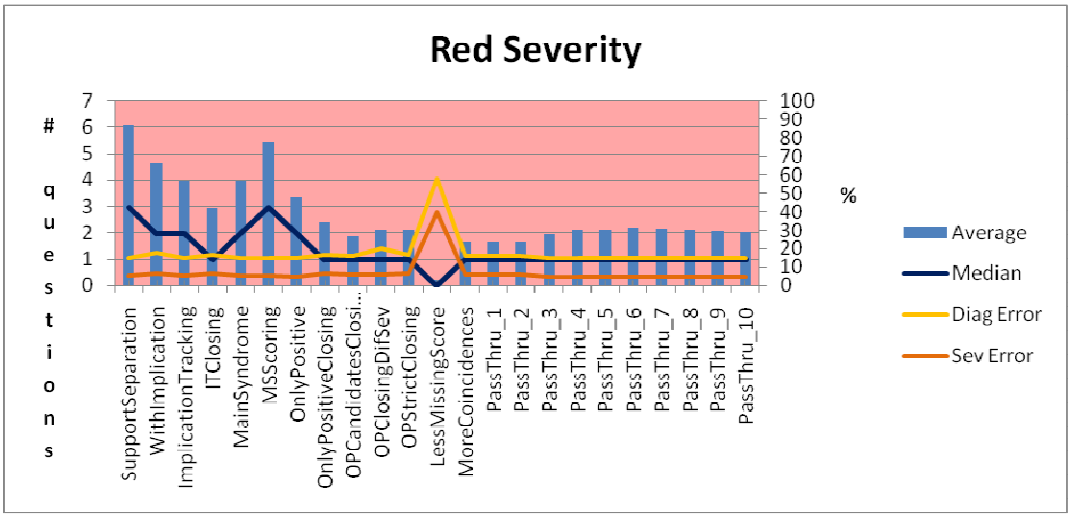
Strategy	Yellow			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	11,75	6	9,61	3,91
SupportOnlyPositive	5,11	4	10,69	4,27
SupportOnlyPositiveClosing	4,24	3	12,08	5,91
SupportOnlyPositiveCandidatesClosing	3,15	2	11,21	5,19
SupportOnlyPositiveClosingDifSev	4,65	3	12,90	6,17
SupportOnlyPositiveStrictClosing	3,61	2	11,52	5,40
SupportMoreCoincidences	2,63	2	11,47	5,45
SupportMoreCoincidencesPassThru_1	2,63	2	11,47	5,45
SupportMoreCoincidencesPassThru_2	2,62	2	11,47	5,45
SupportMoreCoincidencesPassThru_3	3,04	2	11,10	5,09
SupportMoreCoincidencesPassThru_4	3,07	2	10,00	4,99
SupportMoreCoincidencesPassThru_5	2,98	2	10,33	4,63
SupportMoreCoincidencesPassThru_6	2,98	2	9,77	4,06
SupportMoreCoincidencesPassThru_7	2,80	2	10,33	4,06
SupportMoreCoincidencesPassThru_8	2,77	2	10,28	4,06
SupportMoreCoincidencesPassThru_9	2,74	2	10,28	4,06
SupportMoreCoincidencesPassThru_10	2,73	2	10,18	4,01

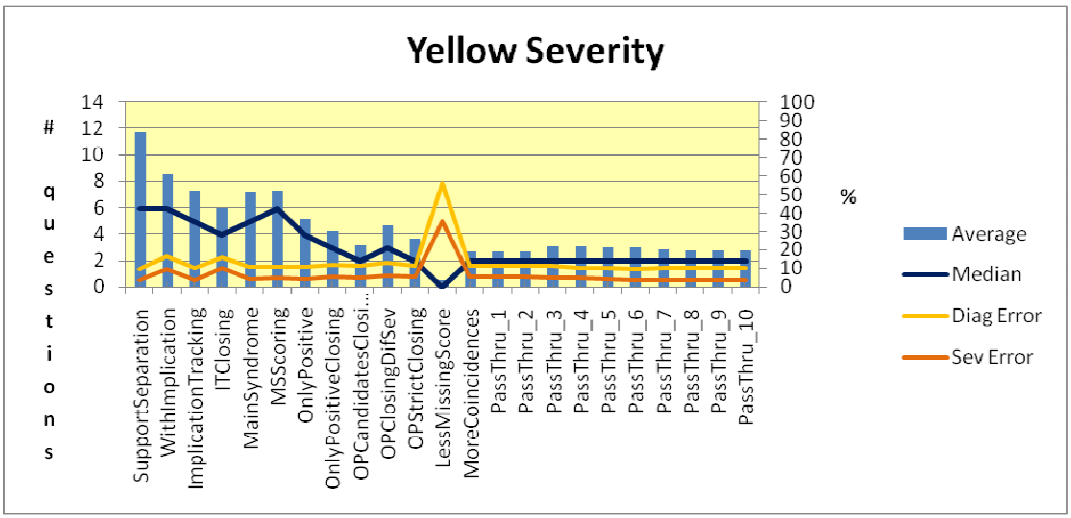
Se observa algo similar para los síndromes de severidad *yellow*. Los valores de 1 y 2 presentan poca variación, y en los dos siguientes aumenta un poco la cantidad de preguntas y se reduce nuevamente el error, aunque en todo este grupo aún sigue por encima el valor de *SupportSeparation*. Los valores más altos vuelven a decrecer en cantidad de preguntas y error y en este caso las opciones para el mejor valor se plantearían, por ejemplo, entre el 2 y el 10.

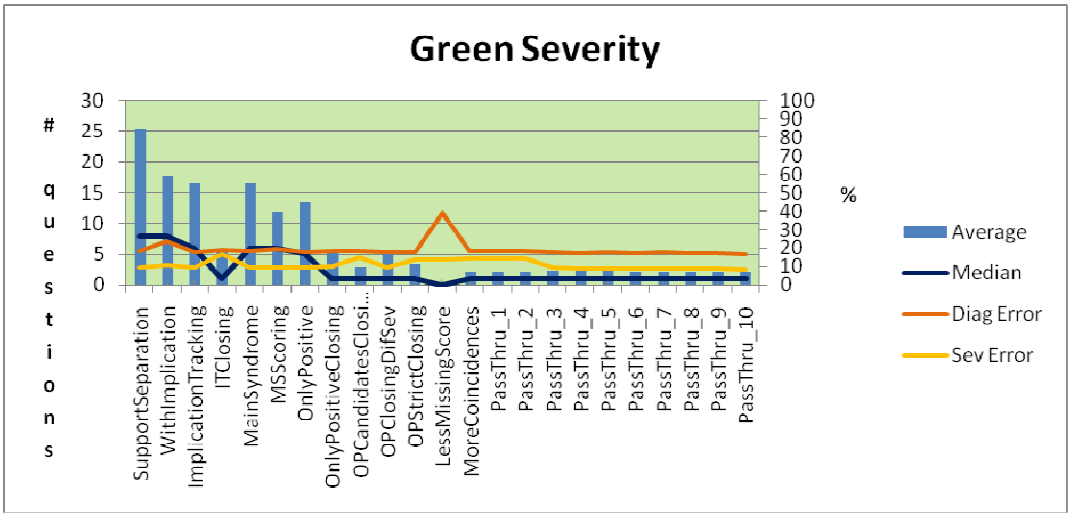
Strategy	Green			
	Average	Median	Diag Error %	Sev Error %
SupportSeparation	25,43	8	18,50	9,97
SupportOnlyPositive	13,44	5	18,08	10,00
SupportOnlyPositiveClosing	5,18	1	18,34	10,48
SupportOnlyPositiveCandidatesClosing	2,82	1	18,41	14,48
SupportOnlyPositiveClosingDifSev	5,23	1	18,11	9,70
SupportOnlyPositiveStrictClosing	3,39	1	18,14	13,82
SupportMoreCoincidences	2,12	1	18,29	14,12
SupportMoreCoincidencesPassThru_1	2,12	1	18,29	14,12
SupportMoreCoincidencesPassThru_2	2,11	1	18,29	14,06
SupportMoreCoincidencesPassThru_3	2,24	1	17,66	10,00
SupportMoreCoincidencesPassThru_4	2,27	1	17,39	9,19
SupportMoreCoincidencesPassThru_5	2,27	1	17,63	9,22
SupportMoreCoincidencesPassThru_6	2,18	1	17,57	9,22
SupportMoreCoincidencesPassThru_7	2,14	1	17,84	9,25
SupportMoreCoincidencesPassThru_8	2,15	1	17,51	9,07
SupportMoreCoincidencesPassThru_9	2,17	1	17,24	9,07
SupportMoreCoincidencesPassThru_10	2,20	1	16,73	8,62

El grupo de síndromes de severidad *green* confirma las observaciones anteriores. También presenta escasa diferencia entre los umbrales 1 y 2, y los siguientes hacen algunas preguntas más, disminuyendo a cambio la cantidad de errores. Nuevamente, las mejores opciones serían los valores 2 o 10.

En conjunto, los resultados son mejores que los de *MoreCoincidences* y que los del resto de las estrategias en general, permitiendo además especular con el ajuste fino del valor del parámetro para establecer un balance entre cantidad de preguntas y error, que podría ser una opción de configuración interesante.


Red Severity
questions
%
0 1 2 3 4 5 6 7
0 10 20 30 40 50 60 70 80 90 100
SupportSeparation
WithImplication
ImplicationTracking
ITClosing
MainSyndrome
MSScoring
OnlyPositive
OnlyPositiveClosing
OPCandidatesClosi...
OPClosingDifSev
OPStrictClosing
LessMissingScore
MoreCoincidences
PassThru_1
PassThru_2
PassThru_3
PassThru_4
PassThru_5
PassThru_6
PassThru_7
PassThru_8
PassThru_9
PassThru_10
Average
Median
Diag Error
Sev Error


Yellow Severity
questions
%
0 2 4 6 8 10 12 14
0 10 20 30 40 50 60 70 80 90 100
SupportSeparation
WithImplication
ImplicationTracking
ITClosing
MainSyndrome
MSScoring
OnlyPositive
OnlyPositiveClosing
OPCandidatesClosi...
OPClosingDifSev
OPStrictClosing
LessMissingScore
MoreCoincidences
PassThru_1
PassThru_2
PassThru_3
PassThru_4
PassThru_5
PassThru_6
PassThru_7
PassThru_8
PassThru_9
PassThru_10
Average
Median
Diag Error
Sev Error


Green Severity
questions
%
0 5 10 15 20 25 30
0 10 20 30 40 50 60 70 80 90 100
SupportSeparation
WithImplication
ImplicationTracking
ITClosing
MainSyndrome
MSScoring
OnlyPositive
OnlyPositiveClosing
OPCandidatesClosi...
OPClosingDifSev
OPStrictClosing
LessMissingScore
MoreCoincidences
PassThru_1
PassThru_2
PassThru_3
PassThru_4
PassThru_5
PassThru_6
PassThru_7
PassThru_8
PassThru_9
PassThru_10
Average
Median
Diag Error
Sev Error

Discusión y metodología

En la sección anterior hemos expuesto con bastante detalle la resolución del problema, las sucesivas estrategias que se fueron implementando y cómo variaron los resultados. Pero hemos dejado de lado uno de los aspectos fundamentales de este trabajo, que nosotros consideramos clave: la metodología.

En cada paso de este trabajo intentamos aplicar un enfoque científico, utilizando la computadora y el ambiente de programación como herramientas para estudiar problemas y pensar soluciones. Estas soluciones son a su vez instrumentos computables que se implementaron en ese mismo ambiente, y contamos con la ventaja de poder probarlos de manera completa en un laboratorio virtual, corriendo simulaciones en ese mismo ambiente. Así, la mayor parte del trabajo consistió en analizar resultados de "experimentos" (las corridas de simulación, utilizamos las comillas porque son experimentos virtuales), elaborar pequeñas teorías que los explicaran y permitieran mejorar una estrategia, y realizar nuevos experimentos para verificar o refutar esas teorías (implementando una nueva estrategia y haciendo nuevas corridas de simulación).

Nuestra hipótesis de trabajo fue que la base de conocimiento de ExpertCare y la ontología subyacente en la misma contenían suficiente información como para que una herramienta automática pudiera conducir el interrogatorio con tan buenos resultados (en cuanto a la cantidad de preguntas necesarias) como el conjunto de reglas definidas por los expertos. Como hipótesis alternativa, consideramos que si no alcanzaba con esa información, bastaría con agregar uno o dos atributos más, es decir, mejorar un poco la base de conocimiento.

Todo el trabajo arrancó con una gran pregunta: ¿Sería suficiente con la información capturada en la base de conocimiento para conducir el interrogatorio? Esta pregunta nace del deseo de prescindir, en la medida de lo posible, de las reglas de interrogatorio. Repasemos brevemente por qué querríamos prescindir de las reglas de interrogatorio, siendo que el ExpertCare funcionando con ellas da buen resultado.

En primer lugar, el conjunto de reglas tiene, en cantidad de objetos, un tamaño casi igual al del resto de la base de conocimiento. Sin embargo, la experiencia probó que incurre en el 80% de la complejidad del software y causa el 90% de los costos de construcción y mantenimiento.

Hay varias razones por las que construir y mantener un conjunto de reglas de interrogatorio es tan costoso en comparación con el resto de la base de conocimiento. Los síntomas, los síndromes, a qué sistema/s corresponden, cuál es su criticidad y qué conjuntos de síntomas los caracterizan son cosas que se pueden encontrar en los libros, en bibliografía médica conocida, accesible y, lo que es más importante, son un conocimiento ya establecido, consensuado y verificado. A tal punto está ese conocimiento instituido que fue posible obtener un certificado de la Facultad de Medicina de la Universidad de Buenos Aires que confirma que la base de conocimientos del ExpertCare representa adecuadamente el estado del arte actual de la práctica médica sobre esas materias. Es decir, es un conocimiento relativamente fácil de adquirir, sistemático (lo que implica que es fácil de llevar a una forma computacional) y relativamente fácil de verificar (lo que implica que es igualmente fácil de mantener).

Por contraste, en el panorama médico de Argentina, prácticamente no hay interrogatorios ordenados, consensuados ni rigurosos. La fuente de estas reglas son los médicos (de diferentes especialidades) y su experiencia personal. Esto trae inmediatamente el problema de que no suelen ser eficientes, formales ni completos, pero sí proclives a error dada su alta improvisación y manualidad. Está en un nivel de conocimien-

to propio de una práctica artesanal. En otros países este conocimiento está un poco más organizado e inclusive formalizado en diversos esquemas, como hemos comprobado en varios artículos (por ejemplo [DONG/07], [WHE/06], [HOL/07], [KAW/05]). Pero este conocimiento médico también tiene otro problema y es que está fuertemente relacionado con las características de una población. Es decir, que sería muy difícil que pudiéramos tomar las reglas de interrogatorio que se utilizan en un sistema de salud en Londres, traducirlas y utilizarlas sin más en el Gran Buenos Aires. Lo más probable es que un intento así (aunque sería mejor que el soporte nulo con que habitualmente cuentan hoy en día los servicios de urgencias) resultara en un cierto choque cultural con las prácticas locales y que las mejoras obtenidas fueran muy inferiores a las que tuvo en el lugar donde fue desarrollado.

Por otra parte, los médicos y su tiempo son un recurso costoso y difícil de manejar. Se trata de un dominio en el que acceder a los conocimientos de los expertos es complicado, y además las consecuencias de un error pueden impactar directamente en la salud (o la vida) de un paciente. Esto por no mencionar que en general, son expertos celosos de su conocimiento y reacios a admitir que el mismo pueda ser "capturado" en una computadora.

Todos estos problemas aumentan en el caso de una base general que abarca muchas especialidades médicas, ya que obtener consenso sobre estrategias de interrogatorio entre médicos de distintas especialidades, que ven cada síntoma con distinta óptica y marco de pensamiento, es notablemente más difícil.

Por las características de la aplicación, además, es difícil revisar el funcionamiento del conjunto de reglas, ya que el "resultado" es un conjunto muy variado de interrogatorios que surgen de la interacción de esas reglas con las respuestas posibles de los pacientes.

Las reglas no sólo son costosas de confeccionar, revisar y poner a punto: también son una traba para mejoras y modificaciones. Por ejemplo, realizar una adaptación para un servicio pediátrico, que parecería una pequeña modificación de la base de conocimiento, sacando los síndromes que no apliquen a la población infantil y tal vez agregando algunos más, requeriría la revisión completa de las reglas y posiblemente una nueva construcción de otro conjunto de reglas para los interrogatorios, muy diferente del original.

El trabajo se hizo muy largo por la dedicación discontinua que aplicamos al mismo, ya que fue realizado únicamente en tiempo libre y con varias interrupciones más o menos prolongadas. Pero contar con un ambiente de objetos y realizar el trabajo de manera evolutiva, nos permitió ante cada interrupción retomar y continuar sin gran esfuerzo, sin necesidad de prolongadas revisiones y adaptaciones del código existente.

Hubo muchas etapas en el trabajo, y vamos a repasar brevemente la historia del mismo.

Definición del problema y enfoque del trabajo

Entre fines de diciembre de 2005 y principios de enero de 2006 acordamos una definición del problema y tomamos algunas decisiones clave sobre el enfoque del proyecto: utilizar Smalltalk como ambiente de estudio y programación, y realizar un desarrollo que fuera independiente del sistema ExpertCare en sí. La única comunicación con el mismo sería a través de importación de archivos. Esto conllevaba el riesgo de duplicar cierto esfuerzo de programación ya que ExpertCare contaba con una imple-

mentación completa, probada y funcionando de todos los conceptos que necesitaríamos para poder implementar una estrategia: síndromes, definiciones, síntomas, etc. Pero consideramos que no sería un gran esfuerzo programar lo necesario para tener un soporte elemental de dichos conceptos en un nuevo ambiente, y que resultaba mucho más valioso asegurar la independencia de este proyecto de la evolución del ExpertCare. Además, al construir "desde cero" el nuevo ambiente, aseguramos una cierta limpieza inicial y teníamos mucho menos riesgo de incluir cosas que no necesitáramos y que luego fuera más difícil sacar.

La definición del problema fue rápidamente capturada en un documento, con el que pudimos comprobar que efectivamente estábamos hablando de lo mismo y estábamos de acuerdo en las definiciones, límites, problemas a resolver, dificultades a enfrentar y alcance del trabajo.

Aún no estaba del todo claro el tipo de técnicas a utilizar, así que enero y febrero del 2006 fueron dedicados a revisar bibliografía general de Inteligencia Artificial, Razonamiento Aproximado, Heurísticas, *Data Mining* y *Machine Learning*. Todos estos campos tienen cierta superposición o similitud con el problema que intentábamos resolver.

De esta revisión y posteriores discusiones surgió la decisión final sobre el enfoque del trabajo: explicitar e implementar con clases y objetos la ontología subyacente en la base de conocimiento de ExpertCare, elaborar estrategias basadas en la información contenida en las relaciones de esa base de conocimiento, y probarlas y depurarlas en un laboratorio virtual. Dejamos abierta la posibilidad de enriquecer la ontología y la base de conocimiento con nuevas relaciones si no alcanzaba con las existentes, ya que algunos proyectos de mejoras al ExpertCare también involucran el agregado de nueva información relevante en su base de conocimiento, por ejemplo agregar los atributos de fisiopatología y ubicación.

A esta altura también definimos que la única métrica que nos interesaba era la cantidad de preguntas necesarias para alcanzar una condición de corte, y establecimos los objetivos de cantidades de preguntas para cada severidad. Esta condición de corte es en primer lugar tener suficiente información para decidir si la llamada corresponde a una situación de urgencia o no, y en segundo lugar contar con un diagnóstico presuntivo suficientemente diferenciado (por su valor medido por una función de scoring) como para adoptarlo.

Dado que el tamaño de la base de conocimiento es manejable y la cantidad de síndromes de la misma lo permite, decidimos realizar mediciones para comparar el sistema actual y las distintas aproximaciones automáticas con una cobertura total de la base de conocimiento. Nos planteamos recorrer cada síndrome, y de cada síndrome tomar cada uno de los conjuntos de síntomas que hacen verdadera su definición. Por cada uno de esos conjuntos, simularíamos una llamada con un paciente que presente esos síntomas, y veríamos cuántas preguntas se necesitan para satisfacer el criterio de corte.

Nuestro trabajo comenzaría entonces por la construcción de un ambiente o laboratorio virtual donde podamos hacer estas pruebas. Sería un ambiente de objetos Smalltalk, una tecnología que conocemos y manejamos bien y que facilita la construcción rápida, la modificación y, en este caso, el intercambio de información y la inclusión final en el sistema existente.

En este ambiente tendríamos que incluir la base de conocimientos y la ontología de ExpertCare, con el agregado de los atributos y relaciones que necesitemos. También se incluirían los componentes necesarios para hacer las simulaciones de prueba recorriendo la base de conocimiento de manera automática y generando archivos de resulta-

dos que pudieran ser analizados luego. Y, por supuesto, las estrategias y heurísticas que se quisieran medir y comparar.

Establecimos que la mecánica de la simulación debe ser en principio genérica, y por completo aislada de la operatoria de la estrategia. Este punto es fundamental por dos razones, una metodológica y otra "económica". La razón metodológica es que es necesario evitar un acoplamiento no deseado que podría conducirnos a que la estrategia y la simulación "cooperen" para producir y medir, respectivamente, los resultados que nosotros *queremos escuchar*, es decir, que estén mutuamente adaptadas de manera de sesgar la medición. La razón económica, que también podríamos llamar "ingenieril del software" si tal expresión existiera, es la posibilidad de reusar la simulación por separado en otros proyectos similares. Intentamos que toda la construcción del laboratorio virtual además genere un *framework* y una aplicación concreta del mismo que tenga posibilidades de ser modificada y adaptada para dominios similares. Bastaría con que el problema se pueda enfocar como una sesión de preguntas y respuestas, donde el interrogado conoce una serie de respuestas predeterminadas y el interrogador busca identificar un concepto entre muchas alternativas, y eso es sumamente genérico.

Algo similar se aplica al caso de las estrategias heurísticas. Si bien pensamos buscar estrategias que actuaran de la mejor manera posible en esta base de conocimiento, las estrategias en sí son artefactos genéricos que podrán operar sobre otras ontologías (aunque, nuevamente, en un contexto de sesión de preguntas y respuestas). Es probable que, cuanto más se parezcan el dominio y la ontología a las que utilizamos, las mismas estrategias operen de manera más similar. Pero eso no impide que una estrategia que opera bien aquí pueda ser llevada a cualquier otro dominio con éxito. Otro tanto puede decirse de estrategias que en este contexto no sean las mejores, pero que tengan mejor *performance* en otra base de conocimiento, con distinta información. Obviamente, estrategias que hagan uso explícito de algún concepto determinado, pueden requerir un *mapping* para adaptarse a otro dominio, pero ése es un problema de relaciones entre ontologías y no de la estrategia en sí.

En esa etapa definimos la ontología con la que íbamos a trabajar. Tiene las clases básicas de Síndrome (ejemplos: gripe, varicela, asma severo) y Síntoma (ejemplos: dolor abdominal agudo, ictericia, mareos), y otras auxiliares para los atributos (que también son instancias de clases): Sistema (ejemplos: respiratorio, nervioso, digestivo), Criticidad (ejemplos: red, yellow y green). La relación principal es que un síndrome está definido por algunos conjuntos alternativos de síntomas (que indicamos operativamente como una expresión lógica tomando la presencia de síntomas como variable, por ejemplo: (fiebre alta Y diarrea Y NO vómitos) O convulsiones), pero también hay relaciones de implicación entre los síntomas (ejemplo: dolor abdominal agudo implica dolor abdominal). Todas las instancias de las clases que indicamos son nombradas y se identifican por su nombre. Los síndromes tienen los atributos de sistema y criticidad, los síntomas no tienen atributos.

Esta ontología puede parecer extremadamente simplificada, aunque la relación de definición es muy rica. Sin embargo, decidimos elegirla por varias razones que nos hicieron pensar que es adecuada para la tarea:

- El trabajo de relevamiento hecho con los médicos al construir las reglas indica que estas categorizaciones y atributos están siempre presentes en la mente del médico que realiza el interrogatorio (el experto de dominio), y ocupan un papel preponderante en su razonamiento
- Esas mismas categorizaciones y atributos están abundantemente descriptos en la bibliografía médica, son fácilmente accesibles y son fáciles de validar.

- Los atributos y relaciones mencionados ya se utilizan exitosamente en los diagnósticos de ExpertCare, siendo la base del algoritmo de scoring y del criterio de corte. Es razonable suponer que si nuestro objetivo es alcanzar la condición de corte lo antes posible, nos basemos la misma información.

Además del conocimiento capturado en la ontología, creemos que es posible aprovechar en las estrategias la información de contexto de la sesión en formas que no son fáciles de capturar en reglas. Podemos dar un ejemplo con un razonamiento abductivo: Si el paciente declara tener vómitos, éste es un síntoma bastante inespecífico que puede corresponder a varios sistemas y muchas patologías, pero si además aparece dolor abdominal, el médico razona guiado por la asociación de síntomas y resignifica los vómitos como un síntoma de trastorno digestivo. Estas asociaciones de síntomas serían difíciles de indicar en reglas (salvo que se haga caso por caso), pero un esquema de clustering que separe en una categoría los síndromes que tengan ambos síntomas, probablemente capturaría que todos los miembros de esa categoría son síndromes del sistema digestivo.

Enfoques alternativos

En esta etapa inicial también hemos considerado otros enfoques posibles, a la luz de lo que leímos y sabemos sobre aplicaciones similares. Haremos un breve repaso de las razones principales por las que preferimos encuadrar el trabajo como se explicó en la sección anterior.

El uso de modelos para razonamiento incierto (*lógica evidencial* o *posibilística*, *probabilidades subjetivas*, modelos de *Dempster-Shaffer*, ver [GCA/97]) se hizo popular para encarar problemas similares, a partir del éxito de famosos sistemas expertos como MYCIN y PROSPECTOR. A pesar de las aparentes similitudes, descartamos este tipo de modelo ya que requeriría una reconstrucción completa de la base de reglas y conocimientos, para proporcionar los valores cuantitativos necesarios para construir un modelo así. Tal construcción sería larga, compleja, requeriría la participación de un equipo de expertos médicos y no hay una buena base para suponer que el resultado sería un modelo mejor que el que tenemos. No disponemos de fuentes confiables para obtener las cuantificaciones que alimentarían estos sistemas. Por otra parte, se haría más compleja su adaptación a cambios en la población, lo que va en contra de otro de nuestros objetivos: facilitar la adaptación del sistema a otros grupos.

Como las reglas que guían el interrogatorio constituyen un gran árbol de decisión, es tentador pensar en utilizar técnicas de aprendizaje automático para construir un árbol similar ([MAT/98]). En un primer momento lo descartamos porque no vimos una adaptación clara de ninguno de ellos que pareciera conducirnos a algo mejor. Por ejemplo, un razonamiento basado en casos puede darnos grupos para estructurar los síndromes, pero es dudoso que esos grupos sean mejores o más fáciles de entender que los que establecen los atributos que elegimos.

Descartamos las redes neuronales de plano, porque nos interesa sobre todo que se puedan explicitar y trazar fácilmente las razones de las decisiones que tome la estrategia automática. Eso sería muy difícil de lograr con una red neuronal.

Por último, aún dentro del enfoque propuesto, podríamos haber intentado inducir la ontología a partir de las reglas de interrogatorio ya definidas manualmente. En principio, no tomamos este enfoque por varios motivos:

- Las reglas todavía cambian con cierta frecuencia, reflejando nuevo conocimiento capturado a partir de casos puntuales, o nuevas estrategias de interrogatorio que

se ensayan. No son un cuerpo estable y sólido de conocimiento, como sí lo son las categorías elegidas en la ontología.

- Una revisión inicial de las reglas y consultas con los expertos indicó que las reglas están basadas principalmente en estas categorías, aunque orientadas a casos particulares. Aún con esto en mente, la carga fue artesanal, por especialistas diversos, y no siempre con un criterio homogéneo y común.
- Partir de las reglas para definir la ontología o las estrategias puede conducir a una situación donde es difícil mejorar o superar lo existente. Es más probable que vicios ocultos en la construcción de las reglas se propagaran a las estrategias. Sería un enfoque más recomendable si estuviéramos convencidos de que el conjunto de reglas es óptimo, pero no es el caso.
- Si utilizamos las reglas existentes, no estaríamos explorando nuevas direcciones, sino repitiendo lo que ya se hizo.

Implementación inicial y primeras pruebas

A mediados de febrero comenzamos la implementación de las clases de la ontología, las que representan los conceptos básicos de operación (sesión, pregunta, observación, etc.) y los componentes que nos permiten automatizar la interacción entre la sesión y la estrategia. También implementamos las primeras estrategias, las más triviales.

Podría decirse que en un par de semanas (de dedicación parcial) estuvo lista la primera versión del laboratorio virtual, gracias a las bondades de la tecnología de objetos, la maravilla del ambiente Smalltalk y nuestro amplio dominio del mismo. La realidad es que, aunque todo eso no deja de ser cierto, esta primera versión era muy limitada y estaba muy lejos de lo que construimos después. Sólo servía para establecer y probar los mecanismos básicos del *framework*, y empezar a estudiar realmente el problema.

Esto da pie para comentar nuevamente el enfoque evolutivo del trabajo. Casi todo lo que se construyó en ese momento es casi todo lo que hay ahora (salvo las estrategias), pero casi nada quedó igual. En las sucesivas etapas del trabajo fuimos comprendiendo mejor el dominio, descubriendo casos, ampliando definiciones y modificando el código sin miramientos. Al ser un proyecto totalmente independiente del producto, no fue necesario asumir ninguna traba de *back-compatibility* y tuvimos total libertad para cambiar implementación, diseño y representación cada vez que fue necesario. La única condición que impusimos y aceptamos sobre el código es que fuera simple y entendible. Tomando en cuenta que probablemente hubiera largas interrupciones y largos períodos en que el conocimiento adquirido sólo estaría capturado en el código de la implementación, era necesario garantizar que ese conocimiento fuera recuperable luego, con el mínimo esfuerzo posible.

Entre marzo y abril de 2006 se realizaron las pruebas descriptas en las secciones de Primeras estrategias y Primeros ensayos. Estas pruebas fueron muy simples y básicas, su propósito principal fue depurar el ambiente y la mecánica de pruebas y conocer más del dominio del problema, encontrando las dificultades que deberíamos superar más adelante. Este es el tipo de conocimiento que es más fácil adquirir programando, el que no se encuentra en ningún libro y el que conduce a la auténtica comprensión de los problemas (y soluciones).

Después de estas pruebas iniciales vino un largo *impasse*, impuesto por el trabajo cotidiano y las actividades de docencia.

Pruebas con la base real y estrategias intermedias

A fines de septiembre de 2006 salimos del letargo. En una semana concretamos la lectura e importación de los archivos de exportación de la base de conocimiento de ExpertCare.

A lo largo de octubre de 2006 realizamos algunas pruebas y se hizo evidente la necesidad de un *cache* para bajar el tiempo de las corridas, ya que en la base real las primeras estrategias requerían demasiadas preguntas. El cálculo de los *satisfiers* de las definiciones era algo muy utilizado por ese grupo de estrategias, y era fácil pensar en guardar ese tipo de información en un *cache*. También descubrimos rápidamente que no podíamos aspirar a una tasa de error de cero, por las características de las definiciones de los síndromes en la base de conocimiento. Esto nos llevó a poner a punto mecanismos de verificación automática para garantizar que, aún en los casos de error, éste se debiera a implicación entre las definiciones y no a un error de programación o de lógica en la estrategia.

Por otro lado, empezamos a hacer estudios cuantitativos sobre la base de conocimiento para responder cuestiones básicas. Esto es lo que se expuso en la sección de Estudio estadístico y agrupamientos. Ambas cosas confluyeron en el `StatisticalCollector`, del que hicimos distintas subclases implementando caches.

Cabe aclarar que estos estudios, paralelamente, llevaron a correcciones y mejoras en la base de conocimiento del ExpertCare.

Entre noviembre de 2006 y febrero de 2007 construimos y probamos las estrategias intermedias, las que se describen en la sección Intermedio: Estrategias y adivinación. También se realizaron importantes trabajos en la mecánica de pruebas y reportes, el recorrido de los síndromes, la simulación de sesiones y los *caches*.

Toda esta etapa estuvo fuertemente influida por los datos obtenidos en el estudio estadístico de la base de conocimiento. El análisis de los resultados de las primeras estrategias no aportó nada más que el convencimiento de que no tenían ninguna posibilidad de resolver el problema. Luego, los números nos guiaron a pensar que el problema se reducía mucho y podía ser manejado con más facilidad en el marco de un solo sistema y una sola severidad. Así empezamos a buscar criterios para adivinar el sistema y la severidad, y surgieron como ayuda los indicadores de incidencia y probabilidad de no estar. En eso basamos las estrategias intermedias, pero rápidamente necesitamos más elementos. Esas estrategias resultaron innecesariamente complejas (cada una involucraba la colaboración de dos estrategias diferentes) y probaron ser inefectivas para su propósito, además de que resultaban lentas. Los resultados obtenidos nos desalentaron de tratar de mejorar los *caches* para acelerar su operación.

Pero en esta etapa se terminó de consolidar la mecánica de trabajo. Fue en ese momento cuando establecimos una corrida de pruebas muy similar a la final y definimos el tipo de información que queríamos/necesitábamos en lo reportes. También empezamos a aplicar el método de implementar la estrategia, hacer una corrida de pruebas y extraer algunos casos problemáticos. Estos casos luego eran analizados paso por paso, tratando de encontrar las causas por las que la estrategia utilizaba muchas preguntas. Eso llevaba a modificaciones en la estrategia o a ideas para una nueva estrategia. Este enfoque resultó extremadamente interesante y productivo.

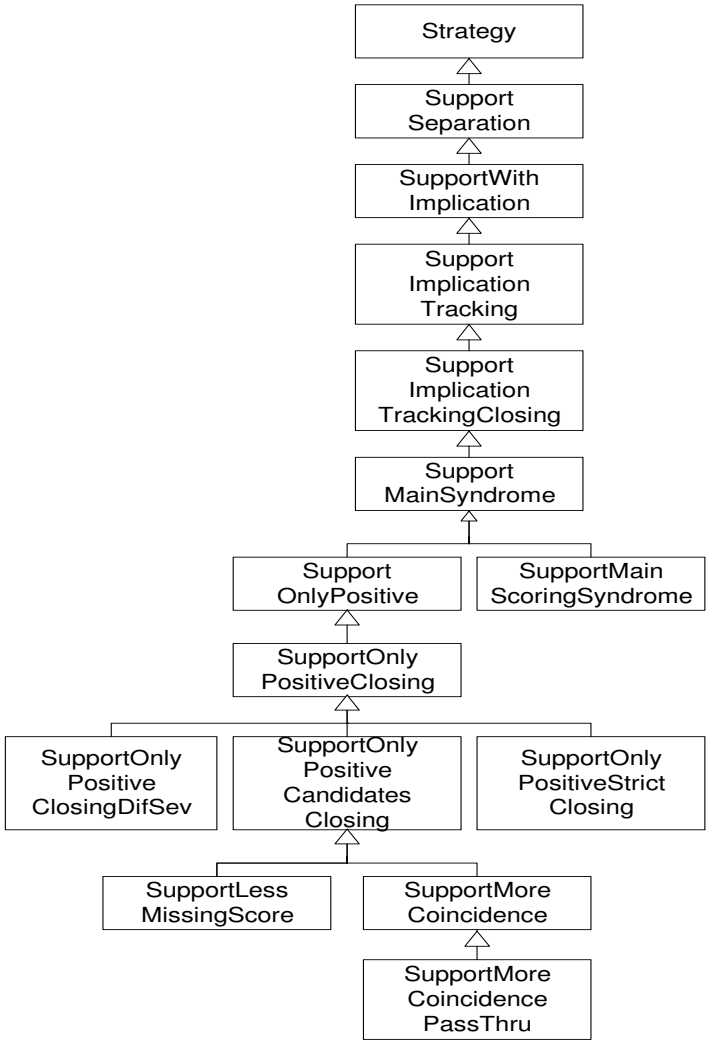
Estrategias de soporte

A fines de febrero de 2007 decidimos utilizar el concepto de soporte como base para otras estrategias, las que describimos en la sección Estrategias basadas en soporte.

La implementación de la primera estrategia basada en este concepto (`SupportSeparationStrategy`) resultó más compleja que las anteriores, sobre todo porque fue necesario hacer adaptaciones y mejoras en los *caches* de `StatisticalCollector` para que la información de soporte se pudiera calcular más rápidamente y las pruebas se pudieran realizar en un tiempo razonable. A diferencia de las estrategias intermedias, en este caso los resultados preliminares eran muy prometedores y justificaban realizar ese esfuerzo, que luego sería capitalizado al analizar toda una familia de estrategias derivadas.

Recién a mediados de abril de 2007 se terminaron las pruebas de `SupportSeparationStrategy` y pudimos completar el ciclo para analizar los casos problemáticos y pensar nuevas líneas para las siguientes estrategias.

Pero la utilización del soporte marcó un cambio cualitativo en el trabajo. Los resultados de la primera estrategia basada en soporte nos decidieron a tomar definitivamente ese camino, y todas las siguientes estrategias evolucionaron a partir de allí. Este cambio se hizo patente inclusive a nivel del diseño e implementación: de la primera etapa y de la etapa intermedia obtuvimos 12 estrategias, todas subclases directas de la clase abstracta `Strategy`. Es decir, que estaban todas al mismo nivel y eran relativamente independientes entre sí. Comparemos esto con el siguiente diagrama de la jerarquía de `SupportSeparationStrategy`:



Aquí se ve claramente el cambio de rumbo al que nos referíamos: Prácticamente, cada nueva estrategia es un refinamiento de la anterior. En pocos casos establecimos líneas alternativas con estrategias "hermanas", casi siempre el análisis de los resultados nos condujo a una pequeña modificación encaminada a reducir la cantidad de preguntas en algún caso problemático que detectamos en la etapa anterior. También este uso de la herencia y trabajar con modificaciones pequeñas hizo que la cantidad de código necesario en cada nueva estrategia fuera muy poco.

Esta modalidad de trabajo y la mecánica de pruebas ya aceitada nos llevaron a desarrollar y probar todas estas estrategias con bastante rapidez. Entre abril y julio de 2007 se logró completar esta etapa del trabajo con éxito.

Sólo quedaba por delante escribir el informe que documentara todo el trabajo, una labor larga que carece de los incentivos del descubrimiento y la resolución de problemas. En el medio, el nacimiento de mi hija hizo aún más prolongada la tarea. Pero eso ya son consideraciones ajenas al trabajo y a la metodología.

Conclusiones

Empezamos el trabajo con una gran pregunta, que se transformó en nuestra hipótesis de trabajo: ¿Sería suficiente la información capturada en la base de conocimiento para conducir el interrogatorio? Por cierto, esta pregunta, llevada al terreno práctico, se transformó en: ¿Podríamos nosotros construir una herramienta automática que condujera el interrogatorio, usando solamente la información de la base de conocimientos y la sesión en curso? Porque no olvidemos que no aspiramos a una demostración de tipo matemático de que tal cosa es posible (que podría ser un deafío intelectual interesante, pero no constructivo), sino que necesitamos una implementación que funcione, al menos a nivel de prueba de concepto.

Esta pregunta fue respondida afirmativamente, y con creces: obtuvimos no sólo una, sino una familia de estrategias automáticas que cumplían con los objetivos de cantidad de preguntas que fijamos de antemano, guiándonos por las opiniones de los expertos y los resultados previos del ExpertCare con el conjunto de reglas de interrogatorio.

Esto acarrea una serie de beneficios inmediatos y relevantes para el sistema ExpertCare, ya que automatizar el procedimiento de interrogación nos permite

1. optimizar los interrogatorios a los fines de realizar menos preguntas y poder determinar la gravedad de la llamada y el diagnóstico presuntivo en menos tiempo
2. tener mayor cobertura y mejor utilización de la base de conocimientos
3. contar con un proceso de decisión racional y verificable
4. facilitar el mantenimiento de la base de conocimientos y el sistema mismo, así como la generación de nuevas bases de conocimiento para adaptarlo a otras condiciones de uso
5. explicar el razonamiento para obtener un diagnóstico presuntivo.

Mejorar el ExpertCare era uno de los objetivos iniciales del trabajo y en todo momento nos pareció importante porque se trata de una aplicación que podría tener un gran impacto en la salud de la población. Pero además, si este sistema se transformara en un caso de éxito, podríamos ver (o hacer) surgir otras aplicaciones de características similares que se utilicen más ampliamente, con perspectivas de mejorar aún más la atención de la salud de la gente, esfera donde pudimos notar una enorme carencia de soporte informático. En algunos casos también, esta carencia está motivada por una negativa a aceptarlo que podría ser rebatida mostrando resultados exitosos.

Pero no solamente eso: utilizar inteligencia artificial y estrategias automáticas permitiría adaptar la misma aplicación a distintos dominios. De un modo similar a la separación de base de conocimiento y motor de inferencia de los sistemas expertos clásicos, que permitió reutilizar los mismos motores de inferencia en muchos y muy diversos dominios, esta separación de base de conocimiento y estrategias para conducir eficientemente interrogatorios e ir adquiriendo información que permita identificar un concepto permitiría reutilizar estrategias y técnicas para construirlas en muchos dominios y aplicaciones. Algunos de los que hemos pensado son aplicaciones para el dominio legal (donde legislación y antecedentes conforman un conjunto con importantes semejanzas con la medicina), análisis de segmentos de riesgo para finanzas o seguros, servicios de ayuda telefónica para atención de problemas diversos (por ejemplo, reclamos a compañías de electricidad sobre el servicio que podrían apuntar a descubrir la

causa de la falla) o sistemas de ayuda on-line con asistentes automáticos (los simpáticos *wizards* y *how-to's*, pero con más inteligencia detrás).

Esto no es sólo para ExpertCare o sus derivados, confiamos en que la difusión de casos de éxito también podría ayudar a introducir aplicaciones de inteligencia artificial basadas en estos principios y técnicas, en otros dominios.

También creemos que es importante que esto se haya realizado con tecnología de objetos, ampliamente usada y aceptada por la comunidad de programadores y "el mercado", a diferencia de las herramientas específicas de inteligencia artificial que son vistas siempre como un nicho separado y coto de unos pocos especialistas. Por supuesto, esa imagen es prejuiciosa y falsa, y una buena forma de mostrarlo es haciendo aplicaciones de inteligencia artificial con herramientas de producción comunes y ampliamente conocidas.

Este trabajo no sólo produjo las estrategias, sino también un *framework* de objetos que permite conceptualizarlas. Esto es importante porque ese *framework* es una representación de conocimiento, captura un modelo abstracto que posibilita que otras personas o grupos estudien en un ambiente de objetos las distintas estrategias, la sesión, la base de conocimiento y las interrelaciones entre estos componentes. Pero un *framework* de objetos es una representación viva y activa del conocimiento, que no sólo permite estudiar y entender, sino que facilita nuevos desarrollos. Esto permitiría estudiar otros dominios y elaborar distintas estrategias de interrogatorio con mucho menor esfuerzo que arrancar desde cero.

La tecnología utilizada permitiría generar a bajo costo aplicaciones similares para otros dominios y otras ontologías. Cada uno de sus componentes principales, por otra parte, podría utilizarse separadamente en otro tipo de proyectos. La mecánica de simulación de respuestas en una sesión de interrogatorio, considerada en un contexto de medición de performance de estrategias competitivas, es algo que podría usarse para muchas cosas, sin que tengan ninguna relación con la medicina. Disponer de una herramienta así programada con objetos y fácil de adaptar es un "producto" de indudable interés. Algo similar pasa con estrategias que puedan utilizarse para mejorar interrogatorios, usando información del interrogatorio actual y de una ontología o base de conocimientos; esto no tiene relación específica con medicina sino que son herramientas de explotación de conocimiento basadas en ontologías, que pueden usarse en muchos contextos.

Es decir, que no solamente el *framework* de objetos es valioso, sino que además sus distintas partes podrían ser reutilizadas por separado.

Y mirando un poco más allá de la tecnología específica, las técnicas de análisis y aprovechamiento de la base de conocimiento que utilizamos aquí, sin duda podrían ser aprovechadas por muchos más, y para mucho más.

Otro concepto que querríamos resaltar es la aplicación sostenida, continua y consistente de un enfoque científico para este trabajo. Queremos mencionarlo específicamente porque aún hoy hay gente que sostiene que la investigación "va por un carril distinto" a la industria o que el enfoque científico está reñido con las necesidades de producción de la industria de software. Nosotros creemos, muy por el contrario, que la única manera de obtener un software que resuelva problemas en el mundo real, que se pueda poner en producción, que resista las continuas modificaciones y operaciones de mantenimiento y adaptación que se exigen del software hoy en día, es con un método científico. Es necesario estudiar y modelar el dominio, y construir modelos sólidos exige razonar de manera científica.

También resultó interesante para el grupo la posibilidad de hacer un trabajo interdisciplinario, concepto que está siempre de moda pero siempre con menos ejemplos

concretos de los que querríamos. Este trabajo tiene elementos de medicina, de inteligencia artificial y de tecnología de objetos.

Llevando estos puntos a un nivel particular, quisiéramos comentar algunos beneficios que encontramos utilizando tecnología de objetos en general y Smalltalk en particular.

En primer lugar, la representación de la ontología y la base de conocimientos fue casi trivial. Esto podría deberse a que ya estaba modelada con objetos en ExpertCare o a que la ontología es muy sencilla, pero nuestra experiencia con otras aplicaciones desarrolladas en Smalltalk es que generalmente resulta muy sencillo expresar allí el conocimiento de un dominio.

La construcción del laboratorio virtual y la maquinaria para realizar las pruebas también resultó muy sencilla y rápida. Nuevamente, podríamos atribuir esto a que ya contamos con experiencia en este ambiente, pero nunca habíamos realizado una aplicación similar y sin embargo, pudimos tener algo en marcha y comenzar los experimentos en un tiempo muy corto. Esto no quiere decir que la tarea haya sido trivial, ni que haya estado exenta de errores. Pero el ambiente Smalltalk nos permite hacer ciclos completos de experiencia muy rápidamente, y nos proporciona herramientas adecuadas de inspección y *debugging*.

Por las mismas razones, también resultó relativamente rápida y sencilla la construcción de herramientas para el estudio y la navegación en la base de conocimiento.

Todo esto se tradujo en que no encontramos barreras ni obstáculos para probar ideas implementando nuevas estrategias. Esto facilitó mucho la mecánica de trabajo de estudiar resultados, elaborar una pequeña teoría proponiendo una mejora a una estrategia existente y ponerla a prueba con una nueva serie de experimentos. El único freno que encontramos algunas veces fue el tiempo que demoraban las pruebas recorriendo toda la base de conocimiento. En algunos casos, una prueba completa tomaba días aunque en el último grupo de estrategias basadas en soporte, toman menos de una hora.

Vale la pena resaltar que los problemas de performance encontrados en las pruebas fueron resueltos utilizando *caches* simples en los accesos repetidos a la base de conocimiento. En ningún caso tuvimos que usar técnicas de programación complejas o sofisticadas, ni estructuras de datos complicadas o grandes. No hay nada que oscurezca el código o lo haga difícil de entender. La cantidad de código es relativamente reducida y es fácil de abarcar y comprender por una sola persona[13].

Finalmente, el ambiente Smalltalk y las herramientas de inspección y *debugging* también fueron nuestro principal auxiliar a la hora de examinar el comportamiento de las estrategias, estudiarlas y entender qué pasaba en casos conflictivos, donde detectábamos que una estrategia hacía demasiadas preguntas. Este tipo de análisis fue la principal fuente de ideas para nuevas estrategias. Resulta difícil expresar cuánto más fácil es hacer este tipo de estudios en un ambiente de objetos "vivos", en el mismo momento en que se realiza su interacción.

[13] Esta es una métrica muy interesante, propuesta por Dan Ingalls, uno de los creadores de Smalltalk. El dice que un sistema que no pueda ser comprendido íntegramente por una sola persona, está condenado al fracaso porque será imposible de mantener. Aunque esto no ha sido probado formalmente, tenemos bastante ejemplos que nos hacen pensar que tiene razón.

Trabajo futuro

Como pasa siempre con este tipo de trabajos, quedan muchas cosas más por hacer. Apenas hemos pasado del nivel de prueba de concepto y tenemos algo que se puede usar, aunque el presente trabajo da las bases para continuar y define una línea de trabajo.

Una de las carencias más importantes en este trabajo, que se hizo sentir en varios momentos y que habría que subsanar para continuar, es la falta de herramientas (sobre todo gráficas).

En este sentido la potencia de las herramientas del ambiente Smalltalk funciona como un arma de doble filo porque el desarrollador siempre puede encontrar la forma de arreglárselas con inspectores y *debuggers*. Entonces, no hay una obligación de hacer interfaces visuales para poder seguir trabajando. Siempre está la posibilidad de seguir adelante con la programación y las pruebas, y dejar las herramientas gráficas para un "más adelante" que nunca llega.

Sería necesaria una herramienta que muestre el avance de la sesión, el progreso, los candidatos que se están analizando y el soporte de cada uno. Tal vez sería conveniente utilizar metáforas de navegación.

Al inicio del trabajo, descartamos hacer una herramienta así porque no estaban claras las nociones/metáforas de proximidad, distancia y navegación que podríamos utilizar. Pero después de empezar a utilizar fuertemente la noción de soporte, esto se podría haber definido sin problemas y actualmente contaríamos con las definiciones e ideas necesarias como para hacerlo.

Consideramos que una herramienta así sería necesaria porque contribuye mucho a la exploración de las estrategias, a entender cómo funcionan y cómo operan las ideas que las guían. Probablemente el uso de una herramienta gráfica produciría nuevas ideas y se enriquecería rápidamente con mayor funcionalidad para interactuar con sesiones y estrategias.

También sería necesaria una herramienta de configuración para la sesión, la estrategia y sus conexiones. En este momento el framework ha quedado muy de caja blanca [JF/88], es necesario que cualquier construcción y configuración se realicen a mano y para ello, primero se debe estudiar, conocer y entender las clases del modelo y cómo interactúan. Una herramienta de configuración más cerrada permitiría usos más inocentes de la combinación de partes y pruebas más amplias, sin necesidad de interiorizarse en los detalles de implementación.

En esta misma línea, también sería bueno contar con una herramienta para configurar las simulaciones, corridas de prueba y *benchmarking* de las estrategias. Nuevamente, esto se hizo programando, escribiendo manualmente (bueno, a lo sumo usando *copy&paste*) el código que prepara las condiciones del ambiente y ejecuta las pruebas. Además, los análisis posteriores de los archivos de resultados se hicieron en Excel, también manualmente, cuando hacerlos desde una herramienta integrada en el ambiente de desarrollo nos hubiera facilitado hacer mejores análisis, con una información más cercana y viva. Por dar un ejemplo muy simple, en lugar de buscar los máximos de cantidad de preguntas en Excel y luego reconstruir lo que sucede en esos casos, podríamos hacer directamente que la herramienta de análisis los seleccione y los corra de nuevo paso por paso.

Pero además de mejores herramientas para trabajar con el framework, hay otro tipo de trabajo que queda por hacer, más relacionado con el dominio de ExpertCare.

En primer lugar, hay que hacer cierto trabajo para integrar estas estrategias automáticas al producto.

Hay una parte que podríamos llamar de bajo nivel de esto, que ya ha sido realizada: las estrategias automáticas fueron desarrolladas en un ambiente separado del ambiente de desarrollo del ExpertCare, de hecho utilizamos inclusive otro dialecto. ExpertCare fue desarrollado en Dolphin Smalltalk mientras que el trabajo con las estrategias automáticas se hizo en Visual Smalltalk. Si bien ambos son dialectos de Smalltalk, hay algunas diferencias que fue necesario subsanar en una integración manual que fue realizada en noviembre de 2007.

Pero además de tener el código y los objetos en el mismo ambiente, la integración al producto requiere más decisiones, definiciones y algo de implementación. En primer lugar, es necesario decidir e implementar si las estrategias van a ser la única guía de la sesión, reemplazando directamente a las reglas de interrogatorio. Esto parece demasiado fuerte para una etapa inicial, así que probablemente se incorporen como un asistente más con un cierto peso en la recomendación que se pueda configurar. También es necesario permitir, inclusive dentro de las estrategias, que haya algunas reglas que puedan pasar por encima de la misma, en una especie de *manual override*. Esto puede ayudar a resolver rápidamente casos en que se detecte que la estrategia elegida tarda demasiado. Pero no solamente eso, sino que también pueden ser necesarias para manejar aspectos psicológicos del paciente y de la sesión. Para dar un ejemplo ingenuo, la estrategia puede sugerir preguntar si el paciente tiene fiebre alta, pero cualquier médico sabe que hay que preguntar la temperatura, aunque la información necesaria sea booleana (>39º), porque ningún paciente consideraría seriamente a un médico al que "no le importa" si tiene 39.5 o 39.8.

Otro tipo de trabajo sumamente necesario es realizar pruebas reales con las estrategias. Hasta ahora, prácticamente no han salido de la prueba de simulación, pero el paciente simulado no hace muchas de las cosas que suceden en interrogatorios con pacientes reales. El `AnswerProvider` nunca se equivoca, nunca se desdice algunas preguntas más tarde de algo que ya había dicho o amplía información previa, nunca olvida un síntoma, conoce exactamente los síntomas que tiene, etc. Todo esto está muy lejos de la realidad y, aunque consideramos que fue una prueba interesante, es imprescindible hacer pruebas reales antes de seguir avanzando.

Finalmente, aunque hemos postulado que el enfoque del trabajo y las estrategias no son directamente dependientes de la medicina o de la aplicación concreta y que esto permite la apertura a otros dominios, esa no deja de ser una hipótesis que debe ponerse a prueba también. Es necesario hacer el intento de encarar otro dominio con estas técnicas, tratando de adaptar las estrategias existentes para poder comprobar que realmente no hay acoplamientos indeseados e impensados o asunciones fuertes y ocultas que hagan que esto esté ligado de maneras ocultas al dominio de síndromes y síntomas.

Bibliografía

[PEA/84] Judea Pearl. “Heuristics – Intelligent Search Strategies for Computer Problem Solving”. Addison-Wesley Publishing Company, 1984.

[GCA/97] Pere García Calvés. “Introducción al Razonamiento Aproximado en la Inteligencia Artificial”. Apuntes ECI, Departamento de Computación FCEyN, 1997.

[MAT/98] Stan Matwin. “Data Mining y Machine Learning: conceptos, técnicas y aplicaciones”. Apuntes ECI, Departamento de Computación FCEyN, 1998.

[GHJV/95] Erich Gamma, Richard Helm, Ralph Jonson, John Vlissides. “Design Patterns – Elements of Reusable Object-Oriented Software”. Addison-Wesley Publishing Company, 1995.

[ABW/98] Sherman R. Alpert, Kyle Brown, Bobby Woolf. “Design Patterns – Smalltalk Companion”. Addison-Wesley Publishing Company, 1998.

[RN/95] Russell, Stuart y Norvig, Peter. “Artificial Intelligence – A modern approach”. Prentice-Hall, 1995.

[JF/88] Ralph E. Johnson, Brian Foote. “Designing reusable classes”. Journal of Object-Oriented Programming, 1(2). p 22-35, 1988.

[LIU/96] Chamond Liu. “Smalltalk, Objects and Design”. Ed. toExcel, 1996.

[GR/83] Adele Goldberg, David Robson. “Smalltalk-80: The Language and its Implementation”. Addison-Wesley Publishing Company, 1983.

[SUNIT] http://sunit.sourceforge.net

[SQUEAK] http://www.squeak.org

[GER/07] Ger S. C., Barchini G. E. y Álvarez M. M. “Ontología De Soporte Al Diagnóstico De Trastornos De Ansiedad”. SIS 2007-36 JAIIO, 2007.

[LAN/05] A. Fernández Landaluce, A. Andrés Olaizola, E. Mora González, B. Azkunaga Santibáñez, S. Mintegi Raso y J. Benito Fernández. “Triage Telefónico Realizado Por Médicos En Urgencias De Pediatría”. Anales de Pediatría, Barcelona, 2005.

[DONG/07] Sandy L. Dong, Michael J. Bullard, David P. Meurer, Sandra Blitz, Brian R. Holroyd, Brian H. Rowe. “The Effect Of Training On Nurse Agreement Using An Electronic Triage System”. Canadian Journal of Emergency Medicine, July 2007.

[WHE/06] Sheila Wheeler. “Telephone Triage Research and Real World Practice”. Telephone Triage Products and Services for Telehealth Industry, http://www.teletriage.com, 2006.

[HOL/07] Inger Holmström. “Decision aid software programs in telenursing: not used as intended? Experiences of Swedish telenurses”. Nursing and Health Sciences, Blackwell Publishing, 2007.

[GARG/05] Amit X. Garg, Neill K. J. Adhikari, Heather McDonald, M. Patricia Rosas-Arellano, P. J. Devereaux, Joseph Beyene, Justina Sam, R. Brian Haynes. “Effects of Computerized Clinical Decision Support Systems on Practitioner Performance and Patient Outcomes - A Systematic Review”. Journal of American Medical Association, March 2005.

[DUB/86] Catherine E. Dubeau, Anthony E. Voytovich, Robert M. Rippey. “Premature Conclusions in the Diagnosis of Iron-deficiency Anemia: Cause and Effect”. Medical Decision Making 6, 1986.

[KAW/05] Kensaku Kawamoto, Caitlin A Houlihan, E Andrew Balas, David F Lobach. “Improving Clinical Practice Using Clinical Decision Support Systems: A Systematic Review of Trials to Identify Features Critical to Success”. Information In Practice, BMJ, March 2005.

[BAR/07] Graciela Barchini, Margarita Álvarez, Susana Herrera y Melina Trejo. “El Rol de las Ontologías en los Sistemas de Información”. Revista Ingeniería Informática 14, Universidad de Concepción (Chile), Mayo 2007. http://www.inf.udec.cl/revista/

Apéndice A: Detalles de implementación

Lectura de archivos de Síntomas/Conceptos

Veamos algo de la implementación de esta lectura:

```
ConceptFileReader >>nextRecord
    | record line tag stream |
    record := ConceptRecord new.
    line := file nextLine.
    [line trimBlanks isEmpty] whileTrue: [
      file atEnd ifTrue: [^record].
      line := file nextLine].
    stream := line readStream.
    stream skipTo: $'.
    record concepto: (stream upTo: $').
    line := file nextLine.
    tag := '#UNIDAD:'.
    (line beginsWith: tag)
      ifTrue: [self dumpNextQuantifiedRecordFrom:
line in: record]
      ifFalse: [self dumpNextExistenceRecordFrom:
line in: record].
    line := file nextLine.
    ^record
```

Al principio se define un nuevo *record* para guardar la información. Se lee la próxima línea del archivo y hay un pequeño ciclo para saltear líneas vacías que puede haber entre registros. También se verifica para salir de la lectura si se llega al final del archivo. Luego de eso, se asume que en `line` tenemos la primera línea del registro, que comienza con CONCEPTO: y tiene luego el nombre del síntoma entre comillas simples. Por eso se usa un *stream* de lectura sobre esa línea de texto, haciéndolo avanzar con `skipTo:` hasta la posición de la primera comilla. Luego, con `upTo:` se lee lo que queda hasta la segunda comilla[14] y se coloca como el concepto del *record*. Se lee la siguiente línea, y aquí viene la división entre los dos tipos de síntoma: si la línea comienza con el *tag* #UNIDAD:, estamos en un síntoma cuantificado, con unidad, máximo y mínimo. De lo contrario tenemos un síntoma común donde podría haber indicadas ciertas relaciones de implicación. Se invoca un método distinto en cada caso, y luego se lee la siguiente línea para dejar listo el archivo al principio del próximo registro. Finalmente, se devuelve el *record* que se completó con los datos leídos, para su proceso.

```
ConceptFileReader >>dumpNextQuantifiedRecordFrom:
line1 in: record
    | line tag stream |
    tag := '#UNIDAD:'.
    stream := line1 readStream.
    stream skip: tag size.
```

[14] Se asume que no se utilizan comillas simples como parte del nombre

```
    record unidad: stream upToEnd trimBlanks.
    line := file nextLine.
    record minimo: line substrings second asNumber.
    line := file nextLine.
    record maximo: line substrings second asNumber.
    ^record
```

Este es el método usado para leer y volcar la información en un registro de síntoma cuantificado. Se arma un *stream* de lectura sobre la línea actual, se saltea con `skip:` el tamaño del *tag* de unidad y se coloca el resto de la línea (truncando los blancos innecesarios) como unidad del registro. Para el mínimo y máximo, se lee la línea y se aprovecha el mensaje `#substrings` para dividir la línea por palabras, sabiendo que la segunda debe ser un número.

```
ConceptFileReader >>dumpNextExistenceRecordFrom: line
in: record
    | tag stream stream2 |
    line isEmpty ifFalse: [
      tag := '#IMPLICA: '.
      stream := line readStream.
      stream skip: tag size.
      [stream atEnd] whileFalse: [| implication |
        implication := stream upTo: $,.
        stream2 := implication readStream.
        stream2 skipTo: $'.
        implication := stream2 upTo: $'].
        record addImplica: implication trimBlanks]].
    ^record
```

Este es el método usado para leer y volcar la información en un registro de síntoma común. Estos síntomas pueden tener o no información de otros síntomas que implican. Si la línea está vacía, es que no tiene esa información adicional y no hay más que hacer. Si por el contrario la tiene, se construye un *stream* de lectura sobre la línea, se saltea con `#skip:` el *tag* inicial y se comienza un ciclo de lectura de los síntomas implicados por éste hasta el final de la línea. Se considera el trozo hasta la siguiente coma, ya que la lista de implicaciones usa este separador. Dentro de cada trozo separado por comas, se lee entre las comillas simples y se agrega una nueva implicación al registro.

```
ConceptFileReader >>processRecord: aConceptRecord
    | symptom |
    aConceptRecord isQuantified
      ifTrue: [
        symptom := QuantifiedSymptom named:
aConceptRecord concepto.
        symptom
          unit: aConceptRecord unidad;
          minimum: aConceptRecord minimo;
          maximum: aConceptRecord maximo]
```

```
      ifFalse: [symptom := Symptom named:
aConceptRecord concepto].
    aConceptRecord implica do: [:string | | implied
|
      implied := Symptom named: string.
      (Symptom implications at: symptom ifAbsentPut:
[OrderedCollection new])
        add: implied.
      (Symptom implieds at: implied ifAbsentPut:
[OrderedCollection new])
        add: symptom]
```

Este es el método que se usa para procesar cada registro leído. Nuevamente, se establece la división inicial entre los síntomas cuantificados y comunes, definiendo el síntoma con la clase correspondiente y el nombre leído. En el caso del síntoma cuantificado, además, se coloca la información de unidad, mínimo y máximo. Luego se procesan las implicaciones. Por cada una, se busca (o crea) el síntoma correspondiente al nombre y luego se agregan las entradas correspondientes en los diccionarios de la clase `Symptom`, `implications` e `implieds`. Estos diccionarios tienen como clave el síntoma y como valor la colección de síntomas que la clave implica o que la implican, respectivamente. Se utiliza el método `#at:ifAbsentPut:` por comodidad, para que si la entrada no existe, la cree con una colección vacía.

Lectura del archivo de Síndromes

```
SyndromeFileReader>>nextRecord
    | record line tag stream stream2 |
    record := SyndromeRecord new.
    line := file nextLine.
    tag := '#NOMBRE: '.
    stream := line trimBlanks readStream.
    stream skip: tag size.
    record nombre: (stream upTo: $').
    line := file nextLine.
    tag := '#EXPRESION: '.
    stream := line trimBlanks readStream.
    stream skip: tag size.
    record expresion: stream upToEnd.
    line := file nextLine.
    tag := '#FRECUENCIA: '.
    stream := line trimBlanks readStream.
    stream skip: tag size.
    record frecuencia: stream upToEnd.
    line := file nextLine.
    tag := '#COMPORTAMIENTOSUGERIDO: '.
    stream := line trimBlanks readStream.
    stream skip: tag size.
    record comportamientoSugerido: stream upToEnd.
    line := file nextLine.
```

```
    tag := '#SISTEMAS: '.
    stream := line trimBlanks readStream.
    stream skip: tag size.
    [stream atEnd] whileFalse: [| system |
      system := stream upTo: $,.
      stream2 := system readStream.
      stream2 skipTo: $'.
      system := stream2 upTo: $'].
      record addSistema: system trimBlanks].
    line := file nextLine.
    ^record
```

Este es el método usado para leer y volcar la información en un registro de síndrome. Es muy similar a la que hicimos del registro del síntoma. Al principio se define un nuevo *record* para guardar la información. Se lee la próxima línea del archivo asumiendo que en `line` tenemos la primera línea del registro, que comienza con NOMBRE: y tiene luego el nombre del síndrome entre comillas simples. Por eso se usa un *stream* de lectura sobre esa línea de texto, haciéndolo avanzar con `#skip:` hasta la posición de la primera comilla. Luego, con `upTo:` se lee lo que queda hasta la segunda comilla y se coloca como el nombre del *record*. En las siguientes líneas se repite un procedimiento similar: se lee la línea, se define el *tag* que corresponde, se monta un *stream* de lectura que avanza con #skip: el tamaño del *tag*, para leer luego lo que queda hasta el final del *stream* (o sea, de la línea) en el campo correspondiente del *record*. EN la última línea del registro viene la información del sistema o los sistemas a que se adscribe este síndrome. Cuando hay más de uno, es una lista separada por comas, por lo que se usa una mecánica de lectura igual a la utilizada para leer las implicaciones en el registro de síntoma.

```
SyndromeFileReader>>processRecord: aSyndromeRecord
    | syndrome expression trio ambulance severity
time |
    syndrome := Syndrome named: aSyndromeRecord
nombre.
    syndrome frequency: aSyndromeRecord frecuencia.
    trio := aSyndromeRecord comportamientoSugerido
      asArrayOfSubstringsSeparatedBy: $|.
    time := trio first trimBlanks.
    time first = $' ifTrue: [time := time copyFrom:
2 to: time size - 1].
    ambulance := trio last trimBlanks.
    ambulance first = $'
          ifTrue: [ambulance := ambulance copyFrom:
2 to: ambulance size - 1].
    syndrome
        time: time;
        place: trio second trimBlanks;
        ambulance: ambulance.
    severity := #green.
    (time beginsWith: '15 min') ifTrue: [severity :=
#red].
```

```
    (time beginsWith: '30 min') ifTrue: [severity :=
#yellow].
    syndrome severity: severity.
    expression := ThesisExpresionParser parse:
aSyndromeRecord expresion.
    syndrome definition: (expression reductionOf:
expression).
    aSyndromeRecord sistemas
      do: [:string | syndrome addSystem: (System
named: string)]
```

Este es el método que se usa para procesar cada registro de síndrome leído. Se define el síndrome a partir del nombre leído y se le coloca la frecuencia que se había leído en el *record*. Luego se procesa el comportamiento sugerido del *record*, que tiene tres partes separadas por *pipes* (|); por ello se usa el mensaje `#asArrayOfSubstringsSeparatedBy:` para obtener un *array* de tres posiciones con cada una de las partes. La primera y la tercera (`time` y `ambulance`) pueden presentar o no comillas para delimitarlas, así que se verifica y se remueven las comillas si están presentes; después de esto se pasan las tres partes al síndrome. La severidad se determina de acuerdo a lo que figura en la última de estas partes: en principio se coloca `#green`, pero si es de 15 minutos se usa `#red` y `#yellow` si es de 30 minutos. El siguiente paso es usar el *parser* para obtener una expresión (ver la sección de Expresiones Lógicas) a partir del texto de la fórmula. Sobre la expresión obtenida, se hace una mínima reducción porque a veces figuran constantes explícitas que pueden eliminarse: en la muestra del archivo se ve uno de estos casos, el último síndrome tiene una definición que termina en … O true, que puede simplificarse trivialmente. Finalmente se agrega el o los sistemas del síndrome.

```
SyndromeFileReader>>readFile
    super readFile.
    self expandSyndromesInDefinitions.
```

Hay un detalle más que nos obliga a redefinir el método `#readFile`, para incorporar un proceso posterior. Algunos síndromes incluyen en la expresión de su definición no solamente síntomas, sino también otros síndromes. Por ejemplo,

```
    #NOMBRE: 'Syndrome 0455'
    #EXPRESION: ('Syndrome 0254' Y 'ExistenceConcept 0066') O ('Existence-
Concept 0066' Y ('ExistenceConcept 0727' O 'ExistenceConcept 0337'))
    #FRECUENCIA: media
    #COMPORTAMIENTOSUGERIDO: '30 minutos' | Hospital | 'Ambulancia con
médico'
    #SISTEMAS: 'SystemConcept 0023'
```

En estos casos, hacemos una expansión del síndrome en la expresión, reemplazando ese término por la expresión que lo define.

```
SyndromeFileReader>>expandSyndromesInDefinitions
    | syndromes |
```

```
    syndromes := Syndrome syndromes
      select: [:syn | syn definition variables
        anySatisfy: [:v | (Syndrome at: v name)
notNil]].
    syndromes do: [:syn |
      syn
        definition: (syn definition
          acceptVisitor:
SyndromeExpansionExpressionVisitor new)
          reduced]
```

El primer paso es seleccionar los síndromes en los que sea necesario aplicar este proceso. Para eso, revisamos las variables de cada definición y vemos si alguna tiene un nombre que corresponda a un síndrome en lugar de un síntoma. Luego, en cada uno de ellos hacemos la expansión usando un *Visitor* [GHJV/95] especial, el `SyndromeExpansionExpressionVisitor`. Este Visitor tiene un método en el que hace la expansión:

```
SyndromeExpansionExpressionVisitor>>visitVariable:
aVariable
    | syn |
    syn := Syndrome at: aVariable name.
    ^syn isNil
        ifTrue: [aVariable]
        ifFalse: [syn definition]
```

Los demás métodos de este `Visitor` sólo propagan la visita a subexpresiones o devuelven la subexpresión original.

Iteración de casos en un subsíndrome

```
SyndromesTraveller>>subsyndromeDo: aBlock
    | answer candidates ap clues auxiliar specific
pairs |
    answer := OrderedCollection new.
    answer addAll: Symptom weakSymptoms.
    candidates := symptoms collect: [:obs | obs
symptom].
    candidates removeAllSuchThat: [:symptom | answer
includes: symptom].
    candidates size > 1 ifFalse: [
      clues := #().
      candidates notEmpty ifTrue: [
        clues := Array with: candidates anyone.
        answer add: candidates anyone].
      ap := AnswerProvider
        syndrome: syndrome
        symptoms: symptoms
        initialSymptoms: answer
```

```
        clues: clues.
      ^aBlock value: ap].
    auxiliar := CachedStatisticalCollector new.
    specific := candidates select: [:symptom | |
systems |
      systems := auxiliar symptomsBySystem at:
symptom name.
      systems size < 5].
    specific notEmpty ifTrue: [
      specific do: [:symptom | | initial |
        clues := Array with: symptom.
        initial := answer copyWith: symptom.
        ap := AnswerProvider
          syndrome: syndrome
          symptoms: symptoms
          initialSymptoms: initial
          clues: clues.
        aBlock value: ap].
      ^self].
    pairs := OrderedCollection new.
    candidates do: [:c1 | | systems1 |
      systems1 := auxiliar symptomsBySystem at: c1
name.
      candidates do: [:c2 | | pair |
        pair := Array with: c1 with: c2.
        (c1 == c2 or: [pairs anySatisfy: [:p | p
elementsEqual: pair]])
          ifFalse: [| systems2 |
            pairs add: pair.
            systems2 := auxiliar symptomsBySystem
at: c2 name.
            (systems1 intersection: systems2) size <
5 ifTrue: [| initial |
              clues := Array with: c1 with: c2.
              initial := answer copy add: c1; add:
c2; yourself.
              ap := SingleDiagnosticAnswerProvider
                  syndrome: syndrome
                  symptoms: symptoms
                  initialSymptoms: initial
                  clues: clues.
              aBlock value: ap]]]]
```

Aquí finalmente encontramos un método complicado, aunque analizándolo veremos que es más complicado debido a no factorizar en otro nivel los varios casos posibles, sino resolverlos todos juntos aquí.

Recordemos que al llegar a este método estamos iterando un *subsíndrome* específico de un síndrome puntual de los que se deben recorrer. Lo que hay que hacer ahora es configurar un `answerProvider` para cada caso que sea interesante analizar para cubrir la definición de este *subsíndrome*. Esta definición es una conjunción de

síntomas, pero hay que elegir cuántos y cuáles se le dan al `answerProvider` como "pistas", es decir, cuáles son los que se van a considerar inicialmente dados antes de comenzar con las preguntas de la sesión. Aquí usamos algunas heurísticas, tratando de emular hasta cierto punto el comportamiento de un paciente.

Inicializamos `answer`, que es la colección donde pondremos los síntomas elegidos como iniciales. Incluimos allí todos los `weakSymptoms`, que consideramos iniciales porque están siempre presente. Mayoritariamente son los relacionados con la identificación del paciente. Incluyen, por ejemplo, varios síntomas/conceptos[15] definiendo la edad y el sexo del paciente. Inicializamos `candidates` con una observación para cada síntoma de la definición del *subsíndrome* que estamos recorriendo. Estos son los candidatos a usar como pistas, así que removemos (si hay alguno) los que están relacionados con `weakSymptoms`, ya incluidos en la lista de `answer`.

Aquí aparece un primer caso para analizar y resolver: puede quedarnos una lista de candidatos con uno o ningún síntoma después de esta operación. Este es el caso que se resuelve dentro del `candidates size > 1 ifFalse:`. Se define `clues`, si no hay candidatos es una colección vacía, y si hay uno, una colección con ese único candidato, que también se agrega a `answer`. Luego se crea el `answerProvider`, pasándole el síndrome, los síntomas que debe simular tener, los que debe indicar inicialmente y los que se consideran pistas. Una vez configurado adecuadamente, se invoca el bloque externo para ejecutar, pasando como argumento el `answerProvider`. Con eso se acaba este caso, ya que la definición del *subsíndrome* no tiene más alternativas para ensayar.

Nos queda el caso en que la definición tenga varios síntomas candidatos luego de descartar los *weak* iniciales. Lo siguiente que hacemos en este caso es ver si hay síntomas *específicos*, considerando como tales aquellos que figuran en síndromes de menos de cinco sistemas distintos. Para esto usamos el mensaje `#symptomsBySystem` del `statisticalCollector` auxiliar, que se inicializa para recabar ese tipo de información. Como hemos visto en el análisis de la información estadística de la base de conocimiento, la mayoría de los síntomas cae dentro de esta categoría y el caso más frecuente es el que cubre la alternativa siguiente, `specific notEmpty ifTrue:`.

En este caso se recorren esos síntomas específicos, considerados lo suficientemente importantes como para que un paciente real los reporte inicialmente. Por cada uno de ellos, se configura un `answerProvider` con el síndrome y los síntomas determinados por el *subsíndrome* recorrido, y se agrega el síntoma específico a su lista de síntomas iniciales y de pistas. Luego, como en el caso anterior, se invoca el bloque externo pasando como argumento este `answerProvider` y se termina el caso.

Queda un último caso, poco frecuente pero que aparece de todos modos en varios *subsíndromes*, y es que todos los síntomas de la definición aparezcan en síndromes de cinco o más sistemas. En este caso, consideramos que un solo síntoma es poca información para encaminar la sesión y configuramos el `answerProvider` con un par de síntomas como pistas. Esto es lo que se hace en el tramo final del método, con una iteración doble sobre los síntomas candidatos. Se descartan los casos en que el síntoma de las dos iteraciones es el mismo, y también aquellos en que el par de síntomas aparezca en más de cinco sistemas (ya que también dar ese par sería poca información). Con

[15] Entre los así llamados síntomas, figura Varón (y también Mujer), Adulto Senior, Bebé y otras categorizaciones. Esto es porque esos conceptos se usan en las definiciones al igual que los síntomas, y no vimos utilidad en definir otro concepto aunque suene un poco chocante en el lenguaje natural decir que ser Varón es un síntoma.

cada caso restante, se configura el `answerProvider` con el síndrome y los síntomas del *subsíndrome* recorrido y el par de síntomas escogido como pistas iniciales, invocando luego el bloque externo con el `answerProvider` como parámetro.

Estrategias Intermedias

Veamos la implementación de `GuessSystemBySeverity`:

```
GuessSystemBySeverity>>nextSymptomFor: aSession
    | answer system symptom |
    collector updateExcludedFrom: aSession.
    [answer isNil] whileTrue: [
      system := self nextSystem.
      system isNil ifTrue: [
        symptoms := #().
        ^nil].
      secondary
        symptoms: symptoms copy;
        system: system;
        severity: severity.
      symptom := secondary nextSymptomFor: aSession.
      symptom isNil
        ifTrue: [self punish: system]
        ifFalse: [^self answer: symptom]]
```

La estrategia tiene un *collector* para navegar la base de conocimiento y una estrategia auxiliar en la variable de instancia `secondary`. Esta estrategia auxiliar es la que intenta resolver, una vez (presuntamente) adivinado el sistema, cuál es el mejor síntoma para preguntar. Puede ser *MoreCorrelation* o *LessNegation*; como dijimos, encontramos muy poca diferencia entre ambas. Además, tiene un diccionario donde se registran castigos para los sistemas que han resultado malos candidatos.

En este método se comienza por actualizar el *collector* con la sesión, para que anote qué síntomas ya resultan excluidos (por haber sido preguntados en el curso de la sesión). Después viene un ciclo hasta que se encuentre una respuesta, esto es, un síntoma candidato para preguntar. Dentro del mismo se busca un sistema candidato, con `#nextSystem`. Si no hay ninguno, la estrategia no puede seguir e indica su terminación con la respuesta `nil`. En el caso normal, encuentra un sistema candidato y configura la estrategia secundaria con los síntomas que quedan por preguntar y el sistema y severidad candidatos. Luego pregunta a la estrategia secundaria por el síntoma candidato para la sesión, si la respuesta es nula es porque la estrategia no encuentra un candidato para ese sistema y esa severidad, con lo que se registra un castigo (*punishment*) para el sistema y se vuelve a intentar (posiblemente con otro sistema candidato). Si, por el contrario, la estrategia secundaria encontró un candidato, éste es devuelto como el síntoma a preguntar a la sesión.

```
GuessSystemBySeverity>>nextSystem
    | answer |
    collector sortedSeverities do: [:sev |
      severity := sev.
```

```
      answer := self nextSystemOfSeverity: sev.
      answer notNil ifTrue: [^answer]].
    ^nil
```

Este es el método que busca el sistema candidato. Recorre las severidades según un orden preestablecido que el *collector* conoce. Es el orden de gravedad decreciente, primero *red*, luego *yellow* y finalmente *green*. Mientras encuentre un sistema candidato en la severidad más grave, la iteración no llega a la siguiente. Es una forma de expresar que primero se revisan todos los sistemas candidatos de la severidad más alta, y luego los de las siguientes severidades en orden.

La implementación de la estrategia `GuessSystemByFrequency` es casi igual, excepto que usa las frecuencias ordenadas (de más frecuente a menos frecuente) en lugar de las severidades, y en el siguiente método busca los síndromes por sistema y frecuencia.

```
GuessSystemBySeverity>>nextSystemOfSeverity: aSymbol
    | dic systems pairs |
    dic := collector syndromesBySystemOfSeverity:
aSymbol.
    systems := dic keys asOrderedCollection.
    pairs := systems
      collect: [:sys | Array
        with: sys
        with: (self supportOf: sys severity:
aSymbol)].
    systems removeAllSuchThat: [:sys |
      (self punishmentFor: sys) > 10
        or: [(pairs detect: [:pair | pair first =
sys]) last < -0.1]].
    systems := systems asArray sortBy: [:a :b | |
supportA supportB |
      supportA := (pairs detect: [:pair | pair first
= a]) last.
      supportB := (pairs detect: [:pair | pair first
= b]) last.
      supportA > supportB
        or: [supportA = supportB and: [(dic at: a)
size > (dic at: b) size]]].
    ^systems notEmpty ifTrue: [systems first]
```

Aquí se busca el sistema candidato, dada una severidad. Se empieza pidiendo al *collector* los síndromes agrupados por sistema, filtrando sólo los que tengan la severidad correspondiente. Se arman pares sistema ➔ soporte, donde el soporte intenta ser una medida (que veremos luego) de la evidencia que respalda la hipótesis de que el síndrome buscado sea de ese sistema y esa severidad. De estos candidatos se remueven los que tengan un castigo alto (mayor que diez, arbitrariamente) o un soporte negativo muy bajo. Luego se ordenan los pares por soporte. A igual soporte, se decide por la cantidad de síndromes que tiene. Si quedó algún sistema que pasó los filtros anteriores, se devuelve el primero (que sería el de soporte más alto).

```
GuessSystemBySeverity>>supportOf: aSystem severity:
aSymbol
    | syms negative positive |
    syms := collector symptomsOfSystem: aSystem
severity: aSymbol.
    syms isEmpty ifTrue: [^-1.0].
    positive := negative := 0.
    syms do: [:sym | | a |
      a := session answerFor: sym.
      a notNil ifTrue: [
        a isPresent
          ifTrue: [positive := positive + 1]
          ifFalse: [negative := negative + 0.5]]].
    ^(positive - negative / syms size) asFloat
```

En este método se calcula el soporte para un sistema en la severidad elegida. Se piden al *collector* los síntomas que aparecen en la base de conocimiento con ese sistema y severidad. Si no hay, se devuelve -1 como convención, indicando que no hay soporte en absoluto para ese sistema. En el caso normal, se recorren los síntomas y se verifican las respuestas de la sesión, sumando un punto por cada síntoma que ya haya sido preguntado y respondido afirmativamente (evidencia positiva) y restando medio punto por cada síntoma preguntado y respondido por la negativa (evidencia negativa). Es una forma ingenua de computar alguna medida de la credibilidad/verosimilitud de un sistema, viendo qué síntomas de ese sistema ya han sido confirmados.

De esta forma se elige el sistema candidato, veamos cómo la estrategia secundaria intenta elegir el síntoma candidato.

```
MoreCorrelationStrategy>>nextSymptomFor: aSession
    | syms pairs |
    symptoms isEmpty ifTrue: [^nil].
    syms := (collector symptomsOfSystem: system)
intersection: symptoms.
    syms isEmpty ifTrue: [^nil].
    pairs := syms asArray collect: [:symptom | | c |
      c := collector correlationOf: symptom in:
system.
      Array with: symptom with: c].
    pairs sortBy: #second.
    ^pairs first first
```

Esto es mucho más simple. Si no queda ningún síntoma para preguntar, devuelve `nil` para indicar a la estrategia primaria que no hay respuesta. Pide a su propio *collector* los síntomas del sistema con que fue configurada previamente, y se queda con la intersección con los que no han sido preguntados. Nuevamente, si no queda ninguno, devuelve una respuesta nula a la estrategia primaria. Si hay candidatos, genera pares síntoma ➔ índice de incidencia, obteniendo los índices de cada síntoma del *collector*. Ordena por este índice y devuelve el síntoma candidato que tenga la correlación más alta.

La implementación de `LessNegationStrategy` es en un todo similar, sólo que pide al *collector* la *probabilidad de no estar* en lugar del *índice de incidencia*.

Así, ya hemos visto cómo funcionan las estrategias primarias `GuessSystemBySeverity` y su variante `GuesssystemByFrequency`, en combinación con las estrategias secundarias alternativas y casi equivalentes `MoreCorrelationStrategy` y `LessNegationStrategy`. Nos queda por analizar el funcionamiento de la última variante, la que intenta una búsqueda por pares de síntomas en lugar de síntomas aislados.

```
GuessSystemUsingPairsStrategy>>nextSymptomFor:
aSession
    | answer system strategy |
    pair isNil ifTrue: [
      collector updateExcludedFrom: aSession.
      system := self nextSystem.
      system isNil ifTrue: [
        atEnd := true.
        ^nil].
      strategy := LessNegationPairStrategy new
        symptoms: symptoms copy;
        system: system;
        severity: severity;
        auxiliar: collector.
      pair := strategy nextPairFor: aSession.
      first := true].
    pair isNil ifTrue: [
      atEnd := true.
      ^nil].
    (first and: [pair first notNil]) ifTrue: [
      first := false.
      ^self answer: pair first].
    first not ifTrue: [
      answer := pair second.
      answer notNil ifTrue: [
        pair := nil.
        ^self answer: answer]].
    pair := nil.
    atEnd := true.
    ^nil
```

Se complica un poco por el manejo de los pares, pero en líneas generales es parecida a `GuessSystemBySeverity`. Lo primero que se ve es si el par de respuesta es nulo, porque en ese caso habría que buscar un nuevo par de síntomas. Para ello, como en las estrategias anteriores, se busca un sistema candidato; si no se encuentra ninguno se sale con una respuesta nula. Con el sistema candidato se configura una estrategia auxiliar. En este caso, es `LessNegationPairStrategy` que en lugar de un solo síntoma, devuelve un par de ellos. Habiendo recibido un nuevo par, se setea la variable `first` para indicar que en el siguiente paso se debe tomar el primer elemento del par.

Si luego de la búsqueda el par sigue siendo nulo, es porque la estrategia auxiliar no pudo encontrar un buen candidato y se finaliza la sesión enviando una respuesta nula porque no quedan más síntomas adecuados para preguntar.

Si la variable `first` está en verdadero, la ponemos en falso para indicar que la próxima vez, debemos tomar el segundo elemento del par y devolvemos como síntoma candidato el primer elemento del par actual. Si por el contrario, `first` es falsa, devol-

vemos el segundo elemento y anulamos el par actual, para forzar una nueva búsqueda la próxima vez. En cualquier otro caso (que sería un imprevisto) devolvemos una respuesta nula y terminamos la sesión.

La búsqueda del sistema candidato con `#nextSystem` procede exactamente como en `GuessSystemBySeverityStrategy`, mientras que la estrategia auxiliar de `LessNegationPairStrategy` funciona igual que `LessNegationStrategy` excepto por el detalle de que busca y devuelve un par de elementos en lugar de uno sólo.

La única idea para usar esta estrategia en lugar de `GuessSystemySeverity` era aprovechar un poco mejor el tiempo (que en ese momento previo a la implementación de buenos caches era muy alto) de calcular las *probabilidades de no estar* de los síntomas en un sistema y severidad determinados. Por otra parte, esto introduce una cierta inercia, ya que lo observado en un momento dado determina dos preguntas, en lugar del comportamiento habitual de las estrategias de reevaluar la situación dentro del nuevo contexto en cada paso.

SupportSeparationStrategy

Veamos en detalle el código de la principal, la primera y la más simple de las estrategias de soporte: `SupportSeparationStrategy`. Al avanzar el trabajo y generarse nuevas estrategias derivadas de ésta, se hicieron diversas operaciones de reingeniería y *refactoring* sobre la jerarquía, por lo que el estado actual de la clase no refleja tanto la simplicidad inicial, sino que incluye diversas modificaciones incorporadas en sucesivas etapas de evolución.

Lo primero es indicar que , a diferencia de las primeras estrategias que usan muy poca información, se agregan unas cuantas variables de instancia para registrar el estado y la historia de la operación de la estrategia en la sesión. Además de la variable `symptoms` que hereda de `Strategy` y sigue teniendo la lista de síntomas que no se han preguntado, incorpora las siguientes:

- `collector` es un *statisticalCollector*, auxiliar que sirve para navegar y obtener información de la base de conocimiento.
- `session` conserva la sesión de interrogatorio en curso, para poder obtener información de contexto de la misma en cualquier momento y en cualquier método.
- `system` es el sistema candidato del momento actual, el que se ha elegido hasta ahora y se está analizando.
- `severity` es la severidad candidata del momento actual. Dentro del sistema candidato, es la severidad que se ha elegido hasta ahora como más prometedora y se está analizando.
- `history` es una colección que guarda los pares de sistema y severidad que se han analizado hasta ahora. Esto sirve en principio para no repetir el análisis de candidatos ya descartados.
- `candidates1/candidates2` son dos colecciones de síntomas candidatos que se están comparando en el momento actual. Se usan en varios cálculos auxiliares.
- `syndromes1/syndromes2` son dos colecciones de síndromes candidatos que se están comparando en el momento actual. Se usan en varios cálculos auxiliares.

- `group1/group2` son dos grupos específicos síndromes candidatos que se están comparando en el momento actual, por ejemplo dos sistemas o dos severidades. Se usan en varios cálculos auxiliares.

Ahora analicemos el código como en las estrategias anteriores, desde el punto de entrada principal:

```
SuppportSeparationStrategy>>nextSymptomFor: aSession
    session := aSession.
    self updateExcludedFrom: aSession.
    system isNil ifTrue: [^self
nextSymptomLookingForSystem].
    severity isNil ifTrue: [^self
nextSymptomLookingForSeverity].
    ^self nextSymptomLookingForSyndrome
```

Primero se registra la sesión en curso en la variable de instancia dedicada a tal fin y se actualiza la lista de excluidos del *collector* auxiliar. Luego, tenemos tres casos alternativos: que no haya un sistema candidato actualmente analizado y tendríamos que escoger uno en `nextSymptomLookingForSystem`, o que haya un sistema pero todavía no se haya elegido una severidad candidata dentro del mismo y se elegiría una en `nextSymptomLookingForSeverity`. El último caso es donde ya hay un sistema y una severidad elegidos como candidatos, y se está buscando un síndrome candidato.

Veamos el primer caso, ya que es también el que el primero se da cuando empieza a funcionar la estrategia, sin ningún candidato. Debemos buscar un sistema.

```
nextSymptomLookingForSystem
    ^self nextSymptomLookingForSystemComingFrom:
nil.
```

Esto es sólo un pase, para llamar a otro método más general usando un argumento que indica que no tenemos un origen interesante que considerar.

```
nextSymptomLookingForSystemComingFrom: aSystemOrNil
    | support candidates difSupport |
    support := self
supportForPositiveSystemsComingFrom: aSystemOrNil.
    support isNil ifTrue: [^nil].
    candidates := self systemCandidatesFrom:
support.
    candidates isCollection ifFalse: [^candidates].
    difSupport := self differencesOfSupportCausedBy:
candidates.
    ^self answer: difSupport last first
```

Aquí ya interviene fuertemente la noción de soporte: lo primero que se hace es calcular el soporte para los sistemas con evidencia positiva. En caso de que devuelva `nil`, querría decir que no hay soporte para ningún sistema y devolveríamos una res-

puesta nula. En general, nos devuelve un arreglo de pares (sistema, soporte), a partir de los cuales se filtra y se obtiene un conjunto de síntomas candidatos. Nuevamente, si eso no fuera posible daríamos una respuesta nula. Finalmente, se calculan las diferencias de soporte que genera cada candidato, en un arreglo de pares (síntoma, diferencia) ordenados de menor a mayor diferencia, por lo que devolvemos el último síntoma (que es el que origina una mayor diferencia en el soporte). Recordemos que estas diferencias son entre los dos primeros sistemas candidatos, aquellos que tienen soporte más alto y entre los que estamos tratando de decidir.

```
supportForPositiveSystemsComingFrom: aSystem
    | positive support |
    positive := self systemsWithPositiveEvidence.
    positive remove: aSystem ifAbsent: [].
    history
      select: [:pair | pair last isNil]
      thenDo: [:pair | positive remove: pair first
ifAbsent: []].
    support := positive
      collect: [:sys | Array with: sys with: (self
supportOfSystem: sys)].
    support := (support sortBy: #second) reversed.
    ^support
```

Este método primero selecciona en la variable `positive` los sistemas con evidencia positiva que los apoye. Esto refleja que, en principio, analizamos sólo síndromes que incluyan alguno de los síntomas ya mencionados por el paciente. Esto podría parecer apresurado pero se basa en que es muy raro que el paciente presente un síndrome pero mencione un conjunto de síntomas totalmente disjunto con los de ese síndrome. Representa una aplicación del razonamiento abductivo que hemos mencionado antes. Con la noción de soporte, además, cualquier síndrome que no tenga evidencia positiva tendrá un soporte de cero o negativo. Por otro lado, por la reevaluación constante de ExpertCare, si esto resultara en un callejón sin salida, se revisaría la suposición contando con más información y obteniendo una nueva lista de candidatos.

Se remueve el argumento que indica qué sistema se analizó previamente, si está en la colección. Esto fue introducido para evitar un ciclo infinito que nos dejaría analizando siempre el mismo sistema. Por la misma razón, se revisa la historia de la estrategia y se sacan de la lista todos los sistemas que han sido analizados, indicados con un par en el que el segundo elemento es nulo.

Luego se calcula una nueva lista de pares con el sistema y su soporte a partir de los sistemas candidatos. Esta se ordena de mayor a menor por soporte y se devuelve.

```
systemsWithPositiveEvidence
    | answer dic |
    answer := OrderedCollection new.
    dic := collector syndromesBySystem.
    dic keysAndValuesDo: [:sys :syndromes |
      (syndromes anySatisfy: [:syn |
        syn symptoms anySatisfy: [:obs |
          (session answerFor: obs symptom) value =
obs value
```

```
            and: [obs symptom isQuantified or: [obs
value]]]])
        ifTrue: [answer add: sys]].
    answer isEmpty ifTrue: [
      dic keysAndValuesDo: [:sys :syndromes |
        (syndromes anySatisfy: [:syn |
          syn symptoms anySatisfy: [:obs |
            | symptom v |
            symptom := obs symptom.
            v := (session answerFor: obs symptom)
value.
            v = obs value
              or: [(v notNil and: [symptom
isQuantified]) and: [obs value includes: v]]]])
          ifTrue: [answer add: sys]]].
    ^answer
```

La obtención de los sistemas con evidencia positiva parece compleja, pero está complicada por hacer dos intentos. El segundo es sutilmente diferente y sólo se hace si el primero no da resultado.

Se analiza un diccionario donde las claves son los sistemas y los valores, las colecciones de síndromes de cada sistema. Este diccionario es provisto por el `collector` auxiliar y se recorre, incluyendo en la colección de respuesta los sistemas que cumplan con la condición encerrada en un doble `anySatisfy:` que indica que se verifique si algún síntoma de algún síndrome del sistema que se está recorriendo la cumple. La condición es que haya una respuesta en la sesión sobre el mismo síntoma, y esa respuesta sea igual a lo que indica la definición y además, sea verdadero o un valor cuantificado.

El segundo intento sólo se realiza si en el primer intento la colección quedó vacía, y es un poco más amplio. Es diferente únicamente para los síntomas cuantificados y en lugar de pedir que el valor de la respuesta coincida con el valor de la definición, se fija si el valor de la definición (que es un rango de valores) incluye al valor de la respuesta.

```
supportOfSystem: aSystem
    | syndromes |
    syndromes := collector syndromesOfSystem:
aSystem.
    ^self supportOfSyndromes: syndromes
```

El soporte de un sistema se calcula pidiendo al `collector` auxiliar los síndromes de ese sistema y calculando luego el soporte de ese conjunto de síndromes con un método más general.

```
supportOfSyndromes: syndromes
    | negative positive total solver |
    syndromes isEmpty ifTrue: [^-1.0].
    positive := negative := total := 0.
    solver := SolvingExpressionVisitor on: session
context.
```

```
    syndromes do: [:syn | | exp |
      exp := solver evaluate: syn definition.
      exp == false
        ifTrue: [| score |
          score := collector severityScoreOf: syn
severity.
          total := score * 5 + total.
          negative := score * 5 + negative]
        ifFalse: [
          exp == true ifTrue: [| score |
            score := collector severityScoreOf: syn
severity.
            total := score * 5 + total.
            positive := score * 5 + positive].
          syn symptoms do: [:obs | | answer |
            total := total + 1.
            answer := session answerFor: obs
symptom.
            answer notNil ifTrue: [
              (answer value = obs value
                  or: [obs symptom isQuantified and:
[obs value includes: answer value]])
                  ifTrue: [positive := positive + 1]
                  ifFalse: [negative := negative +
0.5]]]]].
    ^(positive - negative / total) asFloat
```

Finalmente, este es el método que calcula el soporte de un conjunto de síndromes y define exactamente a qué llamamos soporte.

Lo primero es chequear si el conjunto está vacío y en ese caso consideramos que tiene un soporte negativo, devolviendo -1 por convención. Podríamos haber devuelto cero, que parece más adecuado como definición de soporte de un conjunto vacío, pero optamos por definirlo como fuertemente negativo para que, comparado con cualquier conjunto no vacío, el vacío tenga menor soporte. Esto refleja la idea más general de que no es verosímil (no debe tener soporte) que no haya un síndrome finalmente elegido en la sesión, por lo que el conjunto vacío no es un candidato viable.

Luego se inicializan en cero las variables que utilizaremos para contabilizar aportes positivos y negativos y un total que se usa para una normalización final del valor.

El método utiliza un `solver` auxiliar, que es un *visitor* sobre la jerarquía de expresiones lógicas que permite hacer la evaluación parcial de las definiciones en un contexto, devolviendo `true`, `false` o la expresión que queda sin resolver. Recorre todos los síndromes del conjunto, evaluando con este `solver` la definición de cada uno.

Aquí empieza un análisis de casos. Los casos especiales son aquellos en que la definición ya se puede resolver en `true` o `false`. En estos casos se suma al total y a la cuenta de positivo o negativo, según corresponda, un aporte de 5 veces el puntaje de la severidad del síndrome. Esto es para dar mayor peso a los síndromes de mayor severidad. Los puntajes de cada severidad son provistos por el `collector` y son de 1, 2 y 3

para las severidades `green`, `yellow` y `red`. O sea que el aporte que se suma para un síndrome ya resuelto, sea positiva o negativamente, es de 5, 10 o 15.

En el caso restante sólo sabemos que todavía no se puede resolver el valor de verdad de la definición con la información existente. En este caso se recorren todos los síntomas mencionados en la definición y se suma 1 al total por cada síntoma. El síntoma influye en el soporte si fue preguntado, así que se analiza la respuesta para ese síntoma en la sesión. Si la respuesta existe y coincide con la definición (es la misma, o en el caso de un síndrome cuantificado, su valor está dentro del rango de la definición) suma un aporte positivo de 1, mientras que en otro caso el aporte negativo es de ½. Esto da mayor peso a la evidencia positiva que a la negativa.

Finalmente se restan los aportes negativos de los positivos y se divide por el total, asegurando que el número final queda entre -1 y 1.

Esta medida es ingenua y es empírica. Está orientada por nuestras ideas sobre el problema y la adoptamos en tanto resulta efectiva. Es simplemente una función heurística posible, adecuada para comparar valores en un conjunto de síndromes y fácil de calcular. Se podría intentar hacer un ajuste fino de cada uno de los números involucrados, pero dado el contexto en que la usamos, no creemos que eso ocasione mucha variación en los resultados.

Con esto completamos el análisis de cómo se calcula el soporte de los sistemas con evidencia positiva. Vamos al siguiente paso para obtener un sistema candidato:

```
systemCandidatesFrom: support
    | candidates system1 system2 |
    candidates := OrderedCollection new.
    [candidates isEmpty] whileTrue: [
      support isEmpty ifTrue: [
        history add: (Array with: nil with: nil).
        ^nil].
      (support hasOneElement
        or: [support first last - support second
last
          > self supportDifferenceThreshold]
        or: [
          support first last > 0.0
            and: [support first last / support
second last
              > self supportRatioThreshold]])
        ifTrue: [
          system := support first first.
          severity := nil.
          ^self nextSymptomLookingForSeverity].
      system1 := support first first.
      system2 := support second first.
      support removeFirst; removeFirst.
      candidates1 := collector allSymptomsOfSystem:
system1.
      candidates2 := collector allSymptomsOfSystem:
system2.
      candidates := candidates1 copy.
      candidates addAll: candidates2.
```

```
      candidates := candidates intersection:
symptoms].
    group1 := system1.
    group2 := system2.
    syndromes1 := collector syndromesOfSystem:
system1.
    syndromes2 := collector syndromesOfSystem:
system2.
    ^candidates
```

Recordemos que el argumento `support` recibido es una colección de pares, en este caso de sistema y soporte. Este método busca construir en `candidates` una lista de los síntomas que es interesante preguntar. Inicia un ciclo mientras no haya candidatos, del que puede salir por varios casos especiales.

El primer caso especial es definitivo, es que la lista de `support` venga vacía. Esto querría decir que no hay nada que valga la pena analizar y se devuelve una respuesta nula, indicando en la historia con un par nulo que se llegó a esta situación.

El siguiente caso es cuando en la lista de soporte hay un único elemento, o cuando hay una diferencia muy grande entre el primer elemento y el siguiente. Esto se decide usando valores umbral que también tienen una base empírica. Consideramos que la diferencia es muy grande si el soporte del primer elemento tiene más diferencia que 0.35 (recordemos que son valores con módulo entre 0 y 1) o si es más de 250 veces más grande. En estos casos consideramos que el sistema está decidido y no vale la pena considerar más que el primero (que registramos en la variable de instancia `system`), y vamos a la siguiente etapa que es tratar de decidir una severidad en `nextSymptomLookingForSeverity`.

En otro caso vale la pena analizar los dos primeros sistemas, apuntando a encontrar un síntoma que nos permita decidir entre ellos. Así, vamos a trabajar en `system1` y `2` con estos primeros sistemas de la lista de soporte. Los sacamos de la lista de soporte porque si no resultan candidatos para este par, podemos seguir con sistemas que tengan un soporte más bajo.

En `candidates1` y `2` ponemos los síntomas que aparecen en las definiciones de cualquier síndrome de los sistemas 1 y 2, provistos por el `collector`, y armamos una lista consolidada de síntoma candidatos, haciendo una intersección con los de la estrategia, que son los que todavía no fueron preguntados.

Luego de esto podría pasar que no quede ningún síntoma, por haber sido todos preguntados. En ese caso pasaríamos, como dijimos antes, a la siguiente iteración del `whileTrue:`, con sistemas de soporte más bajo pero que todavía tengan candidatos.

Si, en cambio, hay síntomas candidatos entonces registramos en las variables locales `group1` y `2` a los dos sistemas sobre los que estamos tratando de decidir, y en `syndromes1` y `2` los síndromes de cada uno. Finalmente devolvemos la lista de síntomas candidatos.

```
differencesOfSupportCausedBy: candidates
    | difSupport |
    difSupport := Array streamContents: [:strm |
      candidates do: [:symptom | | array dif |
        array := Array new: 4.
        array at: 1 put: symptom.
```

```
        dif := (candidates1 includes: symptom)
          ifTrue: [self
            variationOfSupportOfSyndromes:
syndromes1
            withSymptom: symptom]
          ifFalse: [0.0].
        array at: 2 put: dif.
        dif := (candidates2 includes: symptom)
          ifTrue: [self
            variationOfSupportOfSyndromes:
syndromes2
            withSymptom: symptom]
          ifFalse: [0.0].
        array at: 3 put: dif.
        array at: 4 put: (array at: 2) + (array at:
3).
        strm nextPut: array]].
    difSupport
      sortBy: [:a :b | a last < b last
        or: [a last = b last and: [a first name <= b
first name]]].
    ^difSupport
```

El último paso para determinar el síntoma a preguntar cuando no se sabe el sistema candidato, es calcular la diferencia de soporte que puede provocar cada uno de los candidatos.

Se recorre la lista de síntomas candidatos y por cada uno se guarda un arreglo de cuatro posiciones: en la primera va el síntoma, en la segunda y la tercera la variación de soporte que produce en los conjuntos de síndromes 1 y 2 respectivamente (recordemos que en este punto, en estas colecciones estarían los síndromes del sistema 1 y 2), y en la última, la suma de ambas variaciones. Para devolver una respuesta, se ordenan estos arreglos por la última posición, que tiene la suma de ambas variaciones, con la idea de que se pueda escoger el síntoma que más produce mayor variación considerando las dos respuestas igualmente posibles. Se desempata por nombre en el orden si producen la misma variación, simplemente para asegurar algún orden repetible.

```
variationOfSupportOfSyndromes: syndromes
withSymptom: aSymptom
    | positive total negative solver total_
positive_ negative_ solver_ |
    negative := positive := total := 0.
    negative_ := positive_ := total_ := 0.
    solver := SolvingExpressionVisitor on: session
context.
    solver positive: aSymptom name.
    solver_ := SolvingExpressionVisitor on: session
context.
    solver_ negative: aSymptom name.
    syndromes do: [:syn | | exp exp_ synTotal |
      synTotal := syn satisfiers size.
```

```
      total := synTotal + total.
      total_ := synTotal + total_.
      (syn isRelatedTo: aSymptom) ifTrue: [
        exp := solver evaluate: syn definition.
        exp == false
          ifTrue: [| score |
            score := collector severityScoreOf: syn
severity.
            total := score * 5 + total.
            negative := score * 5 + negative]
          ifFalse: [
            exp == true
              ifTrue: [| score |
                  score := collector
severityScoreOf: syn severity.
                  total := score * 5 + total.
                  positive := score * 5 + positive]
              ifFalse: [
                  syn symptoms
                       do: [:obs | obs symptom =
aSymptom
                            ifTrue: [positive :=
positive + 1.0]]]].
         exp_ := solver_ evaluate: syn definition.
         exp_ == false
            ifTrue: [| score |
               score := collector severityScoreOf: syn severity.
               total_ := score * 5 + total_.
               negative_ := score * 5 + negative_]
            ifFalse: [
               exp_ == true
                  ifTrue: [| score |
                       score := collector severityScoreOf: syn
severity.
                       total_ := score * 5 + total_.
                       positive_ := score * 5 + positive_]
                  ifFalse: [
                       syn symptoms
                            do: [:obs | obs symptom = aSymptom
                                 ifTrue: [negative_ := negative_
+ 0.5]]]]]].
    ^((positive - negative / total) asFloat
      - (positive_ - negative_ / total_) asFloat)
      abs
```

Este método es muy largo, pero contiene dos repeticiones casi idénticas del cálculo de soporte que vimos antes, usando dos `solvers` auxiliares en lugar de uno. Uno de ellos "simula" una respuesta positiva para el síntoma indicada con `positive:` y el otro, `solver_`[16] usa `negative:` para considerar una respuesta negativa al mismo síntoma. Con esas consideraciones, se calculan dos juegos con las mismas variables `positive`, `negative` y `total` usadas en el cálculo del soporte, duplicadas en las que llevan un *underscore* al final, para el caso negativo. El cálculo se limita únicamente

[16] Por una limitación del parser, no pudimos utilizar solver- (solver negativo) o –solver, que hubieran sido nombres más adecuados.

a los síndromes en los que el síntoma analizado tenga algo que ver, y únicamente computa el valor relacionado con este síntoma, ya que lo que estamos buscando es la variación que provoca y no el soporte total. Finalmente se consolidan las diferencias, restando el total del caso negativo del positivo y devolviendo el valor absoluto de la diferencia.

De esta manera hemos cubierto todas las alternativas para el caso en que no tenemos predilección por un sistema candidato, que es justamente el caso inicial.

El segundo caso general que esta estrategia trata por separado es el de estar analizando un sistema candidato determinado, y tratar de decidir una severidad. Es muy similar al de elegir un sistema candidato, sólo que en este caso nos limitamos al espacio de los síndromes del sistema que nos interesa y las únicas alternativas son las tres severidades.

```
nextSymptomLookingForSeverity
    | answer |
    answer := self
nextSymptomLookingForSeverityComingFrom: nil.
    ^answer
```

Nuevamente hacemos un pase para llamar a otro método más general usando un argumento que indica que no tenemos un origen interesante que considerar.

```
nextSymptomLookingForSeverityComingFrom: symbolOrNil
    | support candidates difSupport |
    support := self
supportForPositiveSeveritiesComingFrom: symbolOrNil.
    candidates := self severityCandidatesFrom:
support.
    candidates isCollection ifFalse: [^candidates].
    difSupport := self differencesOfSupportCausedBy:
candidates.
    ^self answer: difSupport last first
```

Este método es completamente análogo al que utilizamos para buscar el síntoma cuando no conocíamos el sistema y no requiere mayor explicación. Lo mismo sucede con `#supportForPositiveSeveritiesComingFrom:`, `#severitiesWithPositiveEvidence` (limitadas a los síndromes del sistema analizado), y `#severityCandidatesFrom:` que son semejantes a sus contrapartidas referidas a los sistemas, con los ajustes apropiados para llamarse entre sí y trabajar con las severidades.

El caso de buscar un síndrome dentro del ámbito de un sistema y severidad determinadas es levemente distinto:

```
nextSymptomLookingForSyndrome
    |positive candidates difSupport frequencyScores|
    positive := self syndromesWithPositiveEvidence.
    (self checkEndConditionWith: positive)
      ifTrue: [^nil].
    candidates := Set new.
    frequencyScores := Dictionary new.
```

```
    positive do: [:syn | | score |
      score := collector frequencyScoreOf: syn
frequency.
      syn symptoms do: [:obs | | current symptom |
        symptom := obs symptom.
        (symptoms includes: symptom) ifTrue: [
          current := frequencyScores at: symptom
ifAbsent: 0.0.
          frequencyScores at: symptom put: current +
score.
          candidates add: symptom]]].
    candidates isEmpty ifTrue: [| ant |
      history add: (Array with: system with:
severity).
      ant := severity.
      severity := nil.
      ^self nextSymptomLookingForSeverityComingFrom:
ant].
    candidates size = 1
      ifTrue: [^self answer: candidates anyone].
    difSupport := candidates asArray collect:
[:symptom | | dif |
      dif := self variationOfSupportOfSyndromes:
positive withSymptom: symptom.
      Array with: symptom with: dif].
    difSupport sortBy: [:a :b |
      a last < b last or: [
        a last = b last and: [
          (frequencyScores at: a first) <
(frequencyScores at: b first) or: [
            (frequencyScores at: a first) =
(frequencyScores at: b first)
              and: [a first name < b first
name]]]]].
    ^self answer: difSupport last first
```

Se obtiene la colección de síndromes de modo similar a los otros casos, seleccionando en el método `#syndromesWithPositiveEvidence` aquellos síndromes que, teniendo el sistema y la severidad elegidos, tienen además algún síntoma con respuesta positiva en la sesión. Luego se chequea la condición de finalización sobre la colección resultante. Esto es algo que se utilizará luego en algunas subclases de esta estrategia, pero en el caso actual está definida simplemente como `false`, por lo que ese chequeo no tiene ningún efecto.

Luego se recorren las definiciones de los síndromes con evidencia positiva, y se van armando dos colecciones: una para los síntomas candidatos y otra donde se recoge, para cada uno de ellos, un puntaje basado en la frecuencia de los síndromes. Este puntaje se utilizará únicamente al final, al ordenar los candidatos, para desempatar entre dos síntomas que produzcan la misma variación de soporte.

Si la colección de síntomas candidatos resulta estar vacía, quiere decir que no hay ningún síntoma para preguntar dentro del sistema y la severidad elegidos, por lo que

de momento hay que descartarlos; esto se hace anotándolos en la `history` con un par que recuerde la combinación de sistema y severidad, y volviendo al caso en que la severidad no estuviera decidida, indicando en el parámetro de `nextSymptomLookingForSeverityComingFrom:` la severidad actualmente descartada.

El caso especial de que haya un solo síntoma candidato se resuelve de inmediato, devolviendo ese síntoma como respuesta.

En los otros casos, como hemos visto, se recorre la colección de candidatos y se recolecta la variación de soporte que provocarían. Se ordenan los resultados por variación de soporte, desempatando con el puntaje basado en la frecuencia o, en última instancia, usando el nombre. Se devuelve el síntoma candidato que produzca la variación de soporte más alta.

Estrategias de SupportImplication, Tracking y Closing

Creamos la clase `CallSessionWithImplication` como subclase de `CallSession` y redefinimos el método `#addAnswer:`.

```
CallSession>>addAnswer: anAnswer
    context at: anAnswer symptom name put: anAnswer
asObservation.
    ^answers add: anAnswer
```

Esta es la implementación básica en la clase `CallSession`. Lo único que hace es agregar al contexto de evaluación la respuesta correspondiente al síntoma preguntado, y registrarla en la colección de respuestas.

```
CallSessionWithImplication>>addAnswer: anAnswer
    | symptom value related |
    super addAnswer: anAnswer.
    symptom := anAnswer symptom.
    symptom isQuantified ifTrue: [^anAnswer].
    value := anAnswer value.
    related := value
      ifTrue: [
        symptom implicationsClosure reject: [:sym |
          answerProvider symptoms
            anySatisfy: [:obs | obs symptom = sym]]]
      ifFalse: [
        symptom implicatorsClosure reject: [:sym |
          answerProvider symptoms
            anySatisfy: [:obs | obs symptom =
sym]]].
    related do: [:sym | | obs |
      obs := Observation value: value of: sym.
      context at: sym name ifAbsentPut: [obs].
      strategy removeSymptom: sym].
    ^anAnswer
```

Esta es la versión del método en la subclase que toma en cuenta las implicaciones entre síntomas. Lo primero que hace es invocar al método de su superclase y luego, si el síntoma es cuantificado, simplemente termina ya que, como dijimos, las reglas de implicación no se aplican a los síntomas cuantificados.

Luego, consideramos dos casos: si la respuesta fue positiva, tomamos la clausura de síntomas implicados (`implicationsClosure`) y si es negativa, la clausura de síntomas que lo implican (`implicatorsClosure`). En ambos casos excluimos aquellos síntomas que ya hayan sido preguntados, para que la información obtenida a través de la relación de implicación nunca pueda reemplazar una respuesta dada directamente por el paciente.

Finalmente, para cada uno de los síntomas, generamos una observación con el mismo valor que la respuesta dada y la agregamos al contexto de evaluación de las definiciones que guarda la sesión. Además, sacamos estos síntomas de la lista de la estrategia, para que no los considere como síntomas candidatos para preguntar, ya que acabamos de proporcionar la información que necesita sobre los mismos.

Es conveniente hacer un cambio similar en la estrategia, cuando se calculan las diferencias de soporte evaluando posibles respuestas positivas y negativas para el síntoma actual.

```
variationOfSupportOfSyndromes: syndromes
withSymptom: aSymptom
    | positive total negative solver total_
positive_ negative_ solver_ implications |
    syndromes isEmpty ifTrue: [^-1.0].
    negative := positive := total := 0.
    negative_ := positive_ := total_ := 0.
    solver := SolvingExpressionVisitor on: session
context.
    implications := Array withAll: aSymptom
implicationsClosure.
    implications := implications collect: [:sym |
sym name].
    solver positive: implications.
    solver_ := SolvingExpressionVisitor on: session
context.
    implications := Array withAll: aSymptom
implicatorsClosure.
    implications := implications collect: [:sym |
sym name].
    solver_ negative: implications.
    syndromes do: [:syn | | exp exp_ synTotal |
        [ . . . ]
```

Este método es casi igual al de la superclase, excepto que “simula” para las respuesta positivas y negativas, la inclusión de las clausuras transitivas de la relación de implicación correspondientes. Es equivalente a lo que pasaría si en la sesión se incluye cada respuesta como vimos arriba. El resto del método es igual y tal vez podría haberse refactorizado para no tener que repetir el código.

Estos son los únicos cambios necesarios para utilizar consistentemente la información de implicación entre los síntomas.

La estrategia `SupportImplicationTrackin` agrega una variable de instancia más, llamada `sysCandidates`, donde se guarda la lista inicial de sistemas con evidencia positiva usando un método de acceso con la técnica de *lazy initialization*.

```
sysCandidates
    sysCandidates isNil ifTrue: [self
initializeSysCandidates].
    ^sysCandidates
```

```
initializeSysCandidates
    sysCandidates := self
systemsWithPositiveEvidence
```

Para calcular la lista de sistemas candidatos, se utiliza el método heredado que obtiene los sistemas con evidencia positiva en la sesión. Aplicamos la idea solamente al paso de obtener los sistemas, redefiniendo el siguiente método:

```
supportForPositiveSystemsComingFrom: aSystemOrNil
    | positive support |
    positive := self sysCandidates.
    positive isEmpty ifTrue: [
      sysCandidates := nil.
      positive := self sysCandidates.
      positive isEmpty ifTrue: [
        history add: (Array with: nil with: nil).
        ^nil]].
    positive remove: aSystemOrNil ifAbsent: [].
    history
      select: [:pair | pair last isNil]
      thenDo: [:pair | positive remove: pair first
ifAbsent: []].
    support := positive
      collect: [:sys | Array with: sys with: (self
supportOfSystem: sys)].
    support := (support sortBy: #second) reversed.
    support isEmpty ifTrue: [
      sysCandidates := nil.
      positive := self sysCandidates.
      positive remove: aSystemOrNil ifAbsent: [].
      history
        select: [:pair | pair last isNil]
        thenDo: [:pair | positive remove: pair first
ifAbsent: []].
      positive isEmpty ifTrue: [history add: (Array
with: nil with: nil)].
      ^nil].
    (support first second > 0.0 andNot: [support
last second > 0.0])
```

```
      ifTrue: [support := support select: [:pair |
pair second > 0.0]].
    self triggerEvent: #systemSupportChanged: with:
support.
    ^support
```

En la implementación básica de la superclase, al principio se obtiene la lista `positive` evaluando cada vez `#systemsWithPositiveEvidence`, mientras que aquí se utiliza la variable `sysCandidates`. La primera vez se usa el acceso *lazy* y se inicializa de manera equivalente a la de la superclase, pero las siguientes veces no vuelve a calcular la lista, cosa que daría un conjunto posiblemente mayor de sistemas, sino que sigue refinando sobre la lista inicial.

El resto de las diferencias con el método básico son para cuidarse de que la lista pueda quedar vacía o sólo incluyendo sistemas con soporte negativo. En estos casos se fuerza una re-evaluación de la lista de sistemas candidatos, ya que indica que mantenerse en la pista inicial no dio resultado. Esto se hace poniendo `nil` en la variable de instancia `sysCandidates` y volviendo a llamar al acceso *lazy*, que volverá a calcular todos los sistemas con evidencia positiva.

Implementamos la `SupportImplicationtrackingClosingStrategy`, que hereda de la anterior y agrega una variable de instancia llamada `chosen` para registrar el candidato único cuando lo encuentra.

Cambia la condición de finalización de la estrategia, para indicar que termina si encontró este candidato único:

```
atEnd
    ^super atEnd | chosen notNil
```

Y este es el momento en que se puede detectar que queda un único candidato:

```
checkEndConditionWith: positive
    | syndromes |
    syndromes := positive collect: #syndrome in: Set
new.
    syndromes size = 1 ifTrue: [
      chosen := syndromes anyone.
      ^true].
    ^false
```

Este método es llamado desde `#nextSymptomLookingForSyndrome`, el momento en que se busca un candidato para decidir entre síndromes. La implementación básica heredada simplemente responde `false` de manera constante, mientras que aquí responderemos `true` (indicando que sí, tenemos una condición de finalización) en caso de encontrar que los candidatos entre los que tratamos de decidir corresponden al mismo síndrome. Para esto, se recolecta el síndrome básico de la lista recibida (recordemos que en esa lista hay distintos *subsíndromes* obtenidos de definiciones alternativas para los síndromes) en un conjunto, que no acepta elementos repetidos. Si al final resulta que quedó un solo elemento en el conjunto, es porque todos los candidatos corres-

ponden al mismo síndrome, y entonces se considera que tenemos al *candidato único*, y lo colocamos en la variable `chosen`, indicando finalización porque no tiene sentido seguir haciendo más preguntas.

Estrategias MainSyndrome y MainScoringSyndrome

La estrategia *MainSyndrome* intenta calcular la variación de soporte unificando el aporte de los distintos *subsíndromes*. Concretamente, si un síntoma aparece repetido en varias definiciones alternativas, por el efecto multiplicador de una conjunción, sólo lo contabilizamos una vez:

```
SupportMainSyndromeStrategy>>
variationOfSupportOfSyndromes: syndromes
withSymptom: aSymptom
     | positive total negative solver total_ positive_ negative_
solver_ implications visited visited_ |
     syndromes isEmpty ifTrue: [^-1.0].
     negative := positive := total := 0.
     negative_ := positive_ := total_ := 0.
     solver := SolvingExpressionVisitor on: session context.
     implications := Array withAll: aSymptom implicationsClosure.
     implications := implications collect: [:sym | sym name].
     solver positive: implications.
     solver_ := SolvingExpressionVisitor on: session context.
     implications := Array withAll: aSymptom implicatorsClosure.
     implications := implications collect: [:sym | sym name].
     solver_ negative: implications.
     visited := Dictionary new.
     visited_ := Dictionary new.
     syndromes do: [:syn | | exp exp_ synTotal |
        synTotal := syn satisfiers size.
        total := synTotal + total.
        total_ := synTotal + total_.
        (syn isRelatedTo: aSymptom) ifTrue: [
           exp := solver evaluate: syn definition.
           exp == false
              ifTrue: [| score |
                 score := collector severityScoreOf: syn severity.
                 total := score * 5 + total.
                 negative := score * 5 + negative]
              ifFalse: [
                 exp == true
                    ifTrue: [| score |
                         score := collector severityScoreOf: syn
severity.
                         total := score * 5 + total.
                         positive := score * 5 + positive]
                    ifFalse: [| visitedSymptoms |
                         visitedSymptoms := visited at: syn
syndrome ifAbsentPut: [Set new].
                         syn symptoms do: [:obs | | symptom |
                               symptom := obs symptom.
                               (symptom = aSymptom and:
[(visitedSymptoms includes: symptom) not])
                                     ifTrue: [
                                          visitedSymptoms
add: symptom.
```

```
                                        positive := positive +
1.0]]]].
          exp_ := solver_ evaluate: syn definition.
          exp_ == false
             ifTrue: [| score |
                score := collector severityScoreOf: syn severity.
                total_ := score * 5 + total_.
                negative_ := score * 5 + negative_]
             ifFalse: [
                exp_ == true
                   ifTrue: [| score |
                        score := collector severityScoreOf: syn
severity.
                        total_ := score * 5 + total_.
                        positive_ := score * 5 + positive_]
                  ifFalse: [| visitedSymptoms |
                      visitedSymptoms := visited_ at:
syn syndrome ifAbsentPut: [Set new].
                      syn symptoms do: [:obs | | symptom |
                            symptom := obs symptom.
                            (symptom = aSymptom and:
[(visitedSymptoms includes: symptom) not])
                                  ifTrue: [
                                        visitedSymptoms
add: symptom.
                                        negative_ := negative_ +
0.5]]]]]].
     ^((positive - negative / total) asFloat
        - (positive_ - negative_ / total_) asFloat)
        abs
```

Aquí intentamos resaltar en el código las secciones donde se diferencia el código del método propio de esta clase con respecto al heredado de su superclase. Están relacionadas con lo que explicamos arriba, contabilizar únicamente una vez por síndrome principal (*main syndrome*) la variación de soporte producida por un síntoma.

Implementamos una subclase de la estrategia *MainSyndrome*, que se llama *MainScoringSyndrome* y una nueva subclase de `CallSession` llamada `CallSessionScoring` que implementa variaciones del cierre de sesión:

```
CallSession>>shouldContinueWith: candidates
    | full |
    full := diagnoses select: [:d | d isFull].
    candidates do: [:d | | syn |
      syn := d syndrome.
      full do: [:diag | (syn canCompleteFrom: diag
syndrome) ifTrue: [^true]]].
    ^false
```

Esta es la implementación básica en la superclase, que a partir de una lista de síndromes candidatos, se fija si hay alguno que pueda completarse a partir de los ya completos, e indica que corresponde continuar si es así.

```
CallSessionScoring>>shouldContinueWith: candidates
    | full maxScore finalists |
```

```
    full := diagnoses select: [:d | d isFull].
    maxScore := full maxValue: [:diag | self
scoreOfSyndrome: diag syndrome].
    finalists := candidates
      select: [:diag | (self scoreOfSyndrome: diag
syndrome) > maxScore].
    finalists isEmpty ifTrue: [^false].
    finalists do: [:d | | syn |
      syn := d syndrome.
      full do: [:diag | (syn canCompleteFrom: diag
syndrome) ifTrue: [^true]]].
    ^false
```

En la nueva subclase se obtiene el máximo de *score* entre los diagnósticos completos, para luego filtrar entre la lista de síndromes candidatos los que superan ese puntaje. Si no hay ninguno, ya se puede dar por terminada la sesión, lo que se indica devolviendo falso. En caso contrario, se revisa si los síndrome que pueden superar el puntaje pueden ser completados a partir de los que ya están completos, y si no es así, también da por terminada la sesión. Ya hemos discutido previamente la razón por la que no consideramos interesante seguir buscando en un espacio de síndromes que no tenga ningún contacto con los que ya están completos.

```
CallSessionScoring>>scoreOfSyndrome: aSyndrome
    ^aSyndrome symptoms sum: [:obs | self
scoreOfSymptom: obs symptom]
```

El puntaje de un síndrome se define simplemente como la suma de los puntajes de los síntomas que aparecen en su definición.

```
CallSessionScoring>>scoreOfSymptom: aSymptom
    aSymptom isWeak ifTrue: [^0.0].
    ^(auxiliar symptomsBySyndrome at: aSymptom name)
size
```

El puntaje de un síntoma se calcula con la ayuda de un *statisticalCollector* auxiliar, y es la cantidad de síndromes en los que aparece.

En la estrategia, éste es el uso del puntaje en el cálculo del soporte:

```
SupportMainScoringSyndromeStrategy>>
supportOfSyndromes: syndromes
    | negative positive total solver visited |
    syndromes isEmpty ifTrue: [^-1.0].
    positive := negative := total := 0.
    solver := SolvingExpressionVisitor on: session context.
    visited := Dictionary new.
    syndromes do: [:syn | | exp |
      exp := solver evaluate: syn definition.
      exp == false
        ifTrue: [| score |
          score := self scoreOfSyndrome: syn.
          total := score * 5 + total.
          negative := score * 5 + negative]
        ifFalse: [
```

```
            exp == true
               ifTrue: [| score |
                  score := self scoreOfSyndrome: syn.
                  total := score * 5 + total.
                  positive := score * 5 + positive]
               ifFalse: [| visitedSymptoms |
                  visitedSymptoms := visited at: syn syndrome
ifAbsentPut: [Set new].
                  syn symptoms do: [:obs | | answer symptom |
                       symptom := obs symptom.
                       (visitedSymptoms includes: symptom) ifFalse:
[| score |
                            visitedSymptoms add: symptom.
                            score := self scoreOfSymptom:
symptom.
                            total := total + score.
                            answer := session answerFor: symptom.
                            answer notNil ifTrue: [
                                 (answer value = obs value
                                      or: [obs symptom
isQuantified
                                           and: [obs value
includes: answer value]])
                                      ifTrue: [positive :=
positive + score]
                                      ifFalse: [negative :=
negative + (score * 0.5)]]]]]]].
     ^(positive - negative / total) asFloat
```

En las partes donde se utilizaba el puntaje indicado por la severidad, ahora se usa el puntaje calculado por la sesión como se mostró arriba.

No reproducimos el código por su longitud, pero en el método que computa la diferencia de soporte para un síntoma, se aplica la misma modificación.

```
SupportMainScoringSyndromeStrategy>>
scoreOfSymptom: aSymptom
    ^session scoreOfSymptom: aSymptom
```

```
SupportMainScoringSyndromeStrategy>>
scoreOfSyndrome: aSyndrome
    ^session scoreOfSyndrome: aSyndrome
```

El cálculo del puntaje para síntomas y síndromes, simplemente se deriva a la sesión.

Estrategias OnlyPositive *y* OnlyPositiveClosing

La estrategia *OnlyPositive* es una subclase de `SupportMainSyndromeStrategy` y toma, para calcular diferencias de soporte entre candidatos, únicamente los síndromes con evidencia positiva.

```
SupportOnlyPositiveStrategy>>
systemCandidatesFrom: support
    | candidates |
```

```
    candidates := super systemCandidatesFrom:
support.
    candidates isCollection ifFalse: [^candidates].
    syndromes1 := self
syndromesWithPositiveEvidenceIn: syndromes1.
    syndromes2 := self
syndromesWithPositiveEvidenceIn: syndromes2.
    ^candidates
```

En el método que obtiene la lista de síntomas candidatos para decidir entre sistemas, se aplica un filtro adicional sobre `syndromes1` y `2` para que queden únicamente los síndromes con evidencia positiva.

Notemos que primero se invoca con `super` al método definido en la superclase, que ya deja en ambas listas los síndromes del primer y segundo sistema candidatos. El chequeo que verifica si `candidates` es una colección es para los casos especiales en que el método deja `nil` o un candidato único en la lista, que indican una finalización de la operación de la estrategia.

De manera similar se implementa `#severityCandidatesFrom:`. Las colecciones de `syndromes1` y `2` son luego utilizadas en el método `#differencesOfSupportCausedBy:` y definen los grupos de síndromes en donde se busca la mayor diferencia de soporte.

Luego quisimos combinarlo con la idea de *Closing* de evitar la falsa competencia entre *subsíndromes* del mismo síndrome. Del mismo modo que ya la habíamos combinado con *Tracking*, hicimos una subclase de `SupportOnlyPositiveStrategy` llamada `SupportOnlyPositiveClosingStrategy` que agrega la variable de instancia `chosen` y redefine `#atEnd` y `#checkEndConditionWith:`.

```
SupportOnlyPositiveClosingStrategy>>
checkEndConditionWith: positive
    | syndromes |
    positive isEmpty ifTrue: [^false].
    syndromes := positive collect: #syndrome in: Set
new.
    (syndromes size = 1
      or: [positive conform: [:syn | (session
diagnoseOf: syn) isFull]])
      ifTrue: [
        chosen := syndromes anyone.
        ^true].
    ^false
```

Si la lista de positivos está vacía, no podemos terminar todavía. En otro caso, como hicimos antes, recolectamos los síndromes principales de los *subsíndromes* en la lista en un conjunto. Si al final quedó un solo elemento en el conjunto, lo ponemos como elegido (`chosen`) e indicamos que tenemos una condición de finalización. Agregamos un caso más de finalización, si todos los síndromes que quedan en la lista están completos en la sesión; en ese caso devolvemos uno cualquiera de ellos y terminamos. Este caso es discutible, pero lo agregamos para comprobar su funcionamiento.

Estrategias derivadas de OnlyPositiveClosing

La estrategia `SupportOnlyPositiveClosingDifSevStrategy` es una subclase de `SupportOnlyPositiveClosingStrategy` que no agrega ninguna variable de instancia e implementa sólo dos métodos para lograr el efecto que describimos arriba.

```
SupportOnlyPositiveClosingDifSevStrategy>>
nextSymptomLookingForSeverityComingFrom: symbolOrNil
     | support candidates difSupport |
     support := self supportForPositiveSeveritiesComingFrom:
symbolOrNil.
     candidates := self severityCandidatesFrom: support.
     candidates isCollection ifFalse: [^candidates].
     difSupport := self
differencesOfSupportWithDifferenceCausedBy: candidates.
     ^self answer: difSupport last first
```

Este método es igual al heredado, salvo por el detalle de que llama a `#differencesOfSupportWithDifferencesCausedBy:` en lugar de a `#differencesOfSupportCausedBy:`. Ese método, que reproducimos a continuación, es donde se implementa el ordenamiento por máxima diferencia en lugar de por suma entre los aportes de ambos candidatos.

```
SupportOnlyPositiveClosingDifSevStrategy>>
differencesOfSupportWithDifferenceCausedBy:
candidates
     | difSupport |
     difSupport := Array streamContents: [:strm |
        candidates do: [:symptom | | array dif |
           array := Array new: 4.
           array at: 1 put: symptom.
           dif := (candidates1 includes: symptom)
              ifTrue: [self
                 variationOfSupportOfSyndromes: syndromes1
                 withSymptom: symptom]
              ifFalse: [0.0].
           array at: 2 put: dif.
           dif := (candidates2 includes: symptom)
              ifTrue: [self
                 variationOfSupportOfSyndromes: syndromes2
                 withSymptom: symptom]
              ifFalse: [0.0].
           array at: 3 put: dif.
           array at: 4 put: ((array at: 2) - (array at: 3))
abs.
           strm nextPut: array]].
     difSupport sortBy: #last.
     ^difSupport
```

En la línea resaltada, el método común coloca la suma de la segunda y la tercera posición del arreglo en la última, que es la que se usa para ordenar las filas (y por lo tanto, los candidatos).

Por la forma en que se realizó esta implementación, sólo surte algún efecto cuando hay que decidir el síntoma candidato entre dos grupos de síndromes de distinta

severidad dentro del mismo sistema, con miras a desempatar entre las dos severidades y elegir una como candidato, para luego continuar la búsqueda del síndrome.

Agregamos la estrategia `SupportOnlyPositiveStrictClosingStrategy`, otra subclase de `SupportOnlyPositiveClosingStrategy` de implementación muy sencilla y que no requiere de ninguna variable de instancia adicional.

```
SupportOnlyPositiveStrictClosingStrategy>>
systemCandidatesFrom: support
    | candidates |
    candidates := super systemCandidatesFrom:
support.
    ^self filteredCandidatesFromSystem: candidates
```

Este método en la superclase sólo refinaba las listas de `síndromes1` y `2` para quedarse con los que tienen evidencia positiva. Ahora, se invoca un método auxiliar que filtra también los conjuntos de síntomas candidatos.

```
SupportOnlyPositiveStrictClosingStrategy>>
filteredCandidatesFromSystem: precandidates
    | candidates |
    precandidates isCollection
      ifFalse: [^precandidates].
    syndromes1 := self
syndromesWithPositiveEvidenceIn: syndromes1.
    candidates1 := candidates1 select: [:symptom |
      (symptoms includes: symptom) and: [
        syndromes1
          anySatisfy: [:syn | syn symptoms
            anySatisfy: [:obs | obs symptom =
symptom]]]].
    candidates1 isEmpty ifTrue: [| positive |
      positive := self sysCandidates.
      positive remove: group1 ifAbsent: [].
      history add: (Array with: group1 with: nil).
      ^self nextSymptomLookingForSystemComingFrom:
group1].
    syndromes2 := self syndromesWithPositiveEvidenceIn: syndromes2.
    candidates2 := candidates2 select: [:symptom |
      (symptoms includes: symptom) and: [
        syndromes2
          anySatisfy: [:syn | syn symptoms
            anySatisfy: [:obs | obs symptom = symptom]]]].
    candidates2 isEmpty ifTrue: [| positive |
      positive := self sysCandidates.
      positive remove: group2 ifAbsent: [].
      history add: (Array with: group2 with: nil).
      ^self nextSymptomLookingForSystemComingFrom: group2].
    candidates := candidates1 copy.
    candidates addAll: candidates2.
    ^candidates
```

El primer chequeo en este método es, como hemos visto en otros casos, verificar que la lista de precandidatos sea una colección para no atender los casos especiales en que pueda ser `nil` o un solo síntoma, en los que la decisión ya está hecha por otro mecanismo.

Luego, como en la superclase, se filtra el grupo de `syndromes1` para retener sólo los síndromes con evidencia positiva. La diferencia estriba en que ahora la lista de síntomas `candidates1` pasa también por un filtro basado en `syndromes1`, en el `select:` que primero verifica si el candidato está todavía en la lista de síntomas que la estrategia podría preguntar y después, en el doble `anySatisfy:`, verifica que alguno de los `syndromes1` lo tenga en su definición.

Al hacer este nuevo filtrado, podría pasar que nos quedemos sin candidatos, si ninguno cumple con las nuevas condiciones. En ese caso, damos por concluido el análisis del primer sistema, indicando con el agregado en la historia de `group1` (que en este caso es el sistema), removiéndolo de la lista de sistemas candidatos e invocando nuevamente el método de búsqueda que utilizamos cuando no tenemos sistema candidato, `#nextSymptomLookingForSystemComingFrom:`.

El procedimiento se realiza exactamente de la misma forma con el segundo grupo, `syndromes2` y `candidates2`. Finalmente, se arma la lista de candidatos uniendo los que provienen de cada uno de los dos grupos.

De manera completamente análoga se definen los métodos `#severityCandidatesFrom:` y `#filteredCandidatesFromSeverity:`, cuya reproducción no tiene mayor interés.

`SupportOnlyPositiveCandidatesClosingStrategy` es otra subclase de `SupportOnlyPositiveClosingStrategy` que agrega una variable de instancia, `synCandidates`, para registrar el grupo de síndromes candidatos desde el inicio, y mantenernos siempre analizando los derivados de ese grupo. Tiene un acceso con *lazy initialization* para que se calcule el conjunto en el primer acceso que se haga a la variable:

```
SupportOnlyPositiveCandidatesClosingStrategy>>
synCandidates
    synCandidates isNil
      ifTrue: [self initializeSynCandidates].
    ^synCandidates
```

```
SupportOnlyPositiveCandidatesClosingStrategy>>
initializeSynCandidates
    synCandidates := self
currentSyndromesWithPositiveEvidence
```

```
SupportOnlyPositiveCandidatesClosingStrategy>>
currentSyndromesWithPositiveEvidence
    | syndromes |
    syndromes := Array streamContents: [:strm |
      self sysCandidates
        do: [:sys | strm nextPutAll: (collector
syndromesOfSystem: sys)]].
    ^self syndromesWithPositiveEvidenceIn: syndromes
```

Aquí vemos como se obtiene el grupo de síndromes candidatos, conservando sólo aquellos que vienen de sistemas candidatos a su vez y filtrando nuevamente entre estos síndromes los que tienen evidencia positiva.

El método que obtiene los síndromes con evidencia positiva se modifica para contemplar únicamente los miembros dentro de este grupo original:

```
SupportOnlyPositiveCandidatesClosingStrategy>>
syndromesWithPositiveEvidence
    | syndromes |
    syndromes := collector syndromesOfSystem: system
severity: severity.
    ^self synCandidates intersection: syndromes
```

La lista de síndromes candidatos se actualiza con cada nueva pregunta, removiendo aquellos que ya han confirmado una definición falsa.

```
SupportOnlyPositiveCandidatesClosingStrategy>>
updateExcludedFrom: aDiagnosticSession
    super updateExcludedFrom: aDiagnosticSession.
    self synCandidates
      removeAllSuchThat: [:syndrome |
(aDiagnosticSession diagnoseOf: syndrome) isFalse]
```

Los métodos `#systemCandidatesFrom:` y `#severityCandidatesFrom:` tienen la misma implementación que en `StrictClosing`.

```
SupportOnlyPositiveCandidatesClosingStrategy>>
filteredCandidatesFromSystem: precandidates
    | candidates |
    precandidates isCollection ifFalse: [^precandidates].
    syndromes1 := self synCandidates intersection:
syndromes1.
    syndromes2 := self synCandidates intersection:
syndromes2.
    candidates1 := candidates1 select: [:symptom |
       (symptoms includes: symptom) and: [
          syndromes1
             anySatisfy: [:syn | syn symptoms
                anySatisfy: [:obs | obs symptom = symptom]]]].
    candidates1 isEmpty ifTrue: [| positive |
       positive := self sysCandidates.
       positive remove: group1 ifAbsent: [].
       history add: (Array with: group1 with: nil).
       ^self nextSymptomLookingForSystemComingFrom: group1].
    candidates2 := candidates2 select: [:symptom |
       (symptoms includes: symptom) and: [
          syndromes2
             anySatisfy: [:syn | syn symptoms
                anySatisfy: [:obs | obs symptom = symptom]]]].
    candidates2 isEmpty ifTrue: [| positive |
       positive := self sysCandidates.
       positive remove: group2 ifAbsent: [].
       history add: (Array with: group2 with: nil).
```

```
        ^self nextSymptomLookingForSystemComingFrom: group2].
    candidates := candidates1 copy.
    candidates addAll: candidates2.
    ^candidates
```

Este método es casi igual al de `StrictClosing`, con la única diferencia de utilizar los `synCandidates` con su filtrado adicional en lugar de los síndromes con evidencia positiva. El resto del método procede de la misma forma y con el mismo código, y otro tanto pasa en `#filteredCandidatesFromSeverity:`.

Estrategia LessMissingScore

Hicimos una subclase de `SupportOnlyPositiveCandidatesClosingStrategy` llamada `SupportLessMissingScoreStrategy`. Esta subclase no requiere ninguna nueva variable de instancia y sólo agrega seis métodos.

```
SupportLessMissingScoreStrategy>>
initializeSynCandidates
    synCandidates := self
      syndromesWithMoreCoincidencesIn: self
currentSyndromesWithPositiveEvidence
```

Se reimplementa la inicialización de los síndromes candidatos, agregando un filtro adicional. Antes eran todos los síndromes con evidencia positive, ahora se registra sólo el grupo de mayor coincidencia entre ésos.

```
SupportLessMissingScoreStrategy>>
syndromesWithMoreCoincidencesIn: syndromes
    | groups min |
    groups := syndromes groupBy: [:syn | self
missingScoreOf: syn].
    min := groups keys min.
    ^groups at: min
```

El grupo de mayor coincidencia se busca, como dijimos, computando el score faltante de los síndromes con evidencia positiva (que vienen como argumento en este método) y devolviendo el grupo que tenga el valor mínimo.

```
SupportLessMissingScoreStrategy>>
missingScoreOf: aSyndrome
    | missing |
    missing := aSyndrome symptoms select: [:obs |
      | answer symptom |
      symptom := obs symptom.
      answer := session answerFor: symptom.
      answer isNil].
    ^missing sum: [:obs |
          self scoreOfSymptom: obs symptom]
```

Y aquí vemos el cálculo del score faltante de un síndrome: se toman los síntomas dentro de la definición que no hayan sido preguntados, detectándolos en el primer `#select:` porque tienen respuesta nula en la sesión. Estos son los síntomas faltantes. Luego, simplemente se suma el puntaje de esos síntomas.

```
SupportLessMissingScoreStrategy>>
scoreOfSymptom: aSymptom
    ^session scoreOfSymptom: aSymptom
```

El cálculo del puntaje de cada síntoma se deriva a la sesión (que debe ser de la subclase `CallSessionScoring`). Ya hemos visto en otra estrategia cómo se realiza allí el cálculo del puntaje.

```
SupportLessMissingScoreStrategy>>
updateExcludedFrom: aDiagnosticSession
    super updateExcludedFrom: aDiagnosticSession.
    synCandidates := self
syndromesWithMoreCoincidencesIn: synCandidates
```

En cada nueva actualización de información por una nueva pregunta, también se filtra nuevamente el grupo de síndromes candidatos, utilizando el mismo criterio.

Estrategia de MoreCoincidences

Hicimos una nueva subclase de `SupportOnlyPositiveCandidatesClosingStrategy` llamada `SupportMoreCoincidenceStrategy`, que no requiere ninguna variable de instancia adicional y sólo implementa cuatro métodos.

```
SupportMoreCoincidencesStrategy>>
initializeSynCandidates
    synCandidates := self
      syndromesWithMoreCoincidencesIn: self
currentSyndromesWithPositiveEvidence
```

Esta implementación, muy similar a la de *LessMissingScore*, agrega un filtro adicional en la inicialización de la lista de síndromes candidatos, para quedarse con los que registran mayor número de coincidencias entre aquellos síndromes que tienen evidencia positiva. Pero en esta estrategia definimos de manera distinta el cálculo (y la semántica) de "síndromes con más coincidencias". En *LessMissingScore* llamábamos así a los que les faltara menor puntaje para completarse; aquí procedemos de una forma que podríamos considerar opuesta: tomamos los que más síntomas presentes tengan en su definición.

```
SupportMoreCoincidencesStrategy>>
syndromesWithMoreCoincidencesIn: syndromes
    | max groups |
```

```
    groups := syndromes groupBy: [:syn | (self
positiveSymptomsOf: syn) size].
    max := groups keys max.
    ^groups at: max
```

Nuevamente, de modo similar a *LessMissingScore*, agrupamos de acuerdo a nuestro criterio y nos quedamos con el grupo que da un mayor valor para ese criterio. El criterio es, en este caso, la cantidad de síntomas positivos:

```
SupportMoreCoincidencesStrategy>>
positiveSymptomsOf: aSyndrome
    ^aSyndrome symptoms select: [:obs | | answer
symptom |
      symptom := obs symptom.
      answer := session answerFor: symptom.
      answer notNil and: [
        answer value = obs value
          or: [symptom isQuantified and: [obs value
includes: answer value]]]]
```

Consideramos síntoma positivo al que cumple con la condición especificada por el `#select:`, que ya hemos visto en varias implementaciones anteriores. Esta condición verifica que la respuesta en la sesión para un síntoma dado coincide en valor con la que indica la definición del síndrome. Si se trata de un síntoma cuantificado, en lugar de la igualdad se utiliza la inclusión en un rango de valores definido.

```
SupportMoreCoincidencesStrategy>>
updateExcludedFrom: aDiagnosticSession
    super updateExcludedFrom: aDiagnosticSession.
    synCandidates isEmptyOrNil
      ifTrue: [self initializeSynCandidates].
    synCandidates := self
syndromesWithMoreCoincidencesIn: synCandidates
```

Finalmente, en cada actualización de información desde la sesión, se vuelve a calcular la lista de síndromes candidatos eligiendo los que registran más coincidencias con las respuestas del paciente.

Estrategias derivadas de MoreCoincidences

Hicimos una subclase de `SupportMoreCoincidenceStrategy` llamada `SupportMoreCoincidencePassThruStrategy` a la que agregamos una variable de instancia para usar como parámetro, el valor umbral de la decisión que indica a partir de cuántos síndromes consideramos que es mejor pasar a través (*pass thru*) de las etapas pre-establecidas y hacer la decisión directamente dentro del conjunto de síndromes candidatos. Esta variable se llama `minSyndromes`.

```
SupportMoreCoincidencePassThru>>initialize
```

```
    super initialize.
    minSyndromes := 2
```

Esta variable toma un valor por defecto de 2, aunque normalmente será suministrada a modo de configuración, de forma externa, cuando se construye la estrategia para una sesión de interrogatorio. Cabe aclarar que, de acuerdo a la forma en que definimos la estrategia, un valor de 1 la haría totalmente equivalente a *MoreCoincidences*, ya que sólo saltearía etapas si la cantidad de síndromes candidatos fuera 1, y ya sabemos que en ese caso la estrategia termina por acción de otra regla.

```
SupportMoreCoincidencePassThru>>
nextSymptomLookingForSeverityComingFrom: symbolOrNil
    self shouldGoToSyndromes ifTrue: [^self
nextSymptomLookingThruSyndromes].
    ^super nextSymptomLookingForSeverityComingFrom:
symbolOrNil
```

Aquí es donde claramente se impone la decisión de saltear una etapa o seguir. Se evalúa si se debe saltear etapas e ir directamente a la decisión entre síndromes. Si es así, se pasa al método `#nextSymptomLookingThruSyndromes` y de lo contrario, se utiliza el mecanismo "común" provisto por la superclase.

Este es el método que busca un candidato para decidir al severidad, pero exactamente lo mismo sucede en el método `#nextSymptomLookingForSystemComingFrom:` en donde se decide entre dos sistemas candidatos.

```
SupportMoreCoincidencePassThru>>shouldGoToSyndromes
    | syndromes |
    syndromes := synCandidates collect: [:syn | syn
syndrome] in: Set new.
    ^syndromes size < self minSyndromes
```

Aquí se ve la utilización del valor umbral. Simplemente, si la cantidad de síndromes (hablamos de síndromes y no de *subsíndromes*) es menor que el mínimo establecido, se responde que se debería pasar directamente a decidir entre síndromes.

```
SupportMoreCoincidencePassThru>>
nextSymptomLookingThruSyndromes
    | answer |
    (self checkEndConditionWith: synCandidates)
        ifTrue: [^nil].
    answer := self nextSymptomLookingInSyndromes:
synCandidates.
    ^answer
```

El método que decide entre síndromes, al que se llega pasando a través de las etapas, simplemente verifica la condición de finalización y llama a un método auxiliar, pasando el conjunto de síndromes candidatos actualizado como lista para decidir entre ellos.

```
SupportMoreCoincidencePassThru>>
nextSymptomLookingInSyndromes: syndromes
    | candidates frequencyScores difSupport |
    (syndromes conform: [:syn | (session diagnoseOf:
syn) isFull]) ifTrue: [
      chosen := syndromes anyone syndrome.
      ^nil].
    candidates := Set new.
    frequencyScores := Dictionary new.
    syndromes do: [:syn | | score |
       score := collector frequencyScoreOf: syn frequency.
       syn symptoms do: [:obs | | current symptom |
          symptom := obs symptom.
          (symptoms includes: symptom) ifTrue: [
             current := frequencyScores at: symptom ifAbsent: 0.0.
             frequencyScores at: symptom put: current + score.
             candidates add: obs symptom]]].
    candidates isEmpty ifTrue: [| ant |
       history add: (Array with: system with: severity).
       ant := severity.
       severity := nil.
       ^self nextSymptomLookingForSeverityComingFrom: ant].
    candidates size = 1 ifTrue: [^self answer: candidates anyone].
    difSupport := candidates asArray collect: [:symptom | | dif |
       dif := self variationOfSupportOfSyndromes: syndromes
withSymptom: symptom.
       Array with: symptom with: dif].
    difSupport sortBy: [:a :b |
       a last < b last or: [
          a last = b last and: [
             (frequencyScores at: a first) < (frequencyScores at: b
first) or: [
                (frequencyScores at: a first) = (frequencyScores at:
b first)
                   and: [a first name < b first name]]]]].
    ^self answer: difSupport last first
```

Salvo por un pequeño detalle al principio, que consiste en hacer un chequeo adicional para ver si entre los candidatos hay alguno completo (en cuyo caso se elige ése y se termina la operación de la estrategia), el resto del método es exactamente igual a `SupportSeparationStrategy>>nextSymptomLookingForSyndrome` y por eso no profundizamos en su análisis. Este método se refactorizó simplemente para pasar el conjunto de candidatos como argumento, mientras que en el original se analizan los síndromes con evidencia positiva del sistema y severidad elegidos como candidatos.

```
SupportMoreCoincidencePassThru>>
nextSymptomLookingForSyndrome
    | positive answer |
    positive := self syndromesWithPositiveEvidence.
    (self checkEndConditionWith: positive) ifTrue:
[^nil].
    answer := self nextSymptomLookingInSyndromes:
positive.
    ^answer
```

Este método a su vez, se redefine para aprovechar la factorización, utilizando el nuevo método con el argumento adecuado.