UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Singapur - Una herramienta basada en tecnología semántica para capturar y diseminar el conocimiento sobre mejores prácticas de una organización.
Autores: Francisco Apesteguía
Director: Dr. Alejandro Fernández
Codirector: Dra. Alicia Díaz
Asesor profesional: –
Carrera: Licenciatura en Sistemas

Resumen

En el contexto del desarrollo colaborativo de software, la falta de conocimiento acerca de lo que está realizando cada participante en la tarea compartida (workspace awareness) y de las prácticas de desarrollo adoptadas (mejores prácticas), tiene un impacto negativo tanto en el proceso de desarrollo como en la calidad del producto final. Las violaciones a las buenas prácticas se detectan mediante revisiones manuales que bien pueden no realizarse bajo ciertas situaciones o, en caso de que se lleven a cabo, suelen ser más propensas a errores a medida que va aumentando la cantidad de buenas prácticas a controlar. Contando con un modelo semántico de las buenas prácticas y de las actividades realizadas en el proyecto, el conocimiento disponible puede ser fácilmente recuperado o explotado por un agente inteligente, para prevenir estas situaciones no deseadas. La motivación de este trabajo radica en proveer una plataforma que permita modelar y gestionar las buenas prácticas (con una definición semántica detallada de las mismas y de los procedimientos que involucran) y a su vez, mediante la recolección de información de las actividades del proyecto, logre detectar en el momento la ocurrencia de violaciones a las buenas prácticas definidas, y facilite su corrección mediante la distribución del conocimiento de la organización en el momento en que es relevante a las personas involucradas, dando como resultado un proceso de desarrollo más confiable y eficiente.

Palabras Claves

Mejores prácticas – Inferencia – Gestión del Conocimiento – Web Semántica – Modelo Semántico – Calidad del Software

Conclusiones

El aporte de Singapur a la gestión de las mejores prácticas consiste en la generación automática de nueva información y su difusión, a partir de los datos de trabajo que se registran. Es fundamental el rol de las tecnologías de la Web Semántica, ya que permiten una fácil integración de los datos provenientes de distintas fuentes y realizar inferencia sobre los mismos. Los resultados obtenidos de la prueba realizada con los datos de un proyecto de desarrollo real demostraron que Singapur puede detectar las violaciones modeladas y reportarlas.

Trabajos Realizados

-Se desarrolló una aplicación web con un modelo semántico para la gestión de las mejores prácticas y la detección mediante inferencia de las violaciones a las mismas.
-Se diseño un modelo semántico para describir las buenas prácticas y sus actividades (extendiendo la ontología SEON).
-Se analizó el framework Apache Jena para soportar el modelo semántico de la aplicación.
-Se estudió un caso real para validar que la aplicación funciona correctamente.

Trabajos Futuros

-En el futuro se puede trabajar sobre un enfoque que distribuya la información necesaria para evitar que se produzcan las violaciones en vez de detectarlas luego de que se produzcan.
-Una posibilidad que surge al disponer de los datos semánticos del trabajo realizado en varios proyectos pertenecientes a distintas organizaciones, es analizarlos automáticamente para generar nuevo conocimiento sobre las cosas que funcionaron bien o mal.
-Se puede realizar una validacion automática de las nuevas reglas de detección de violaciones.

Fecha de la presentación: JUNIO 2015

Singapur - Una herramienta basada en tecnología semántica para capturar y diseminar el conocimiento sobre mejores practicas de una organización

Francisco Apesteguía

1 de junio de 2015

Índice general

1. Introducción **4**
1.1. Organización 5

2. Gestión del Conocimiento **7**
2.1. Ciclo de Gestión del Conocimiento 8
2.2. Las mejores prácticas como productos de conocimiento 9
2.3. Mejores prácticas en el contexto de un proyecto de desarrollo de software 11
2.3.1. Integración exitosa del código fuente 11
2.3.2. Evitar conflictos en las modificaciones al código fuente 11
2.3.3. Cobertura de tests completa 12
2.4. Un modelo para la gestión de mejores prácticas 12
2.4.1. Creación/Identificación 12
2.4.2. Codificación 12
2.4.3. Evaluación 13
2.4.4. Intercambio y difusión 13
2.4.5. Contextualización 13
2.4.6. Adquisición y aplicación 13
2.4.7. Retroalimentación 14

3. Requerimientos de un sistema de gestión de mejores prácticas **15**
3.1. Modelado del dominio del problema 15
3.2. Detección de violaciones a las buenas prácticas y difusión de la información 17
3.3. Roles del sistema 17
3.4. Retroalimentación 18

4. Web Semántica **19**
4.1. Componentes . . . 19
4.1.1. Sentencia . . . 20
4.1.2. Datos de instancias . . . 20
4.1.3. Ontología . . . 20
4.1.4. URI (Uniform Resource Identifier) . . . 21
4.1.5. Lenguaje . . . 21
4.1.6. Herramientas . . . 21
4.2. Modelado de la información . . . 22
4.2.1. RDF . . . 23
4.2.2. RDFS . . . 31
4.2.3. OWL . . . 32
4.3. Rol en la Gestión de Mejores Prácticas . . . 39

5. Trabajo Relacionado **40**
5.1. Escritorio Semántico y Nepomuk . . . 40
5.2. SEON (Software Evolution ONtologies) . . . 41
5.3. FedX . . . 42
5.4. TUKAN . . . 43

6. Enfoque Semántico a la Gestión de Mejores Prácticas **45**
6.1. Singapur . . . 45
6.2. Arquitectura . . . 46
6.2.1. Arquitectura interna del servidor de Singapur . . . 49

7. Modelo Semántico de Mejores Prácticas **51**
7.1. Definición de mejores prácticas . . . 52
7.1.1. Integración exitosa del código . . . 52
7.1.2. Evitar conflictos en las modificaciones al código fuente 53
7.1.3. Cobertura de tests completa . . . 54
7.2. Modelo del proyecto de desarrollo de software . . . 55
7.2.1. Modelo para la integración del código fuente . . . 59
7.2.2. Modelo de modificaciones al código fuente . . . 60
7.2.3. Modelo de cobertura de tests . . . 63
7.3. Modelo del estado de una buena práctica . . . 65
7.4. Generación de datos . . . 66
7.4.1. Generación de datos de Integración del código fuente . 66
7.4.2. Generación de datos de modificaciones no integradas al código fuente . . . 66
7.4.3. Generación de datos de cobertura de tests . . . 68

7.5. Agente generador de datos de Integración del código fuente . 68

8. Detección y Reporte de Violaciones a Mejores Prácticas 72
8.1. Motor de inferencia y sus reglas 73
8.1.1. Funciones primitivas 73
8.1.2. Reglas de inferencia para detectar violaciones en la integración del código fuente 74
8.1.3. Reglas de inferencia para detectar la edición simultánea del código fuente 80
8.1.4. Reglas de inferencia para detectar violaciones en la cobertura de tests 82
8.2. Reporte de violaciones a las mejores prácticas 85
8.2.1. Reporte genérico de violaciones detectadas por Singapur 85
8.2.2. Agente consumidor de datos 85

9. Evaluación 89
9.1. Modelado y detección de las mejores prácticas 89
9.2. Pruebas automatizadas del funcionamiento de Singapur . . . 90
9.2.1. Pruebas de integración exitosa del código fuente . . . 91
9.2.2. Pruebas de evitar conflicto en edición de archivo . . . 95
9.2.3. Pruebas de cobertura completa de tests 96
9.3. Escenario real 98

10. Conclusiones y trabajo futuro 103
10.1. Trabajo futuro 104

Capítulo 1

Introducción

En el contexto del desarrollo colaborativo de software, la falta de conocimiento acerca de lo que está realizando cada participante en la tarea compartida (workspace awareness [15]) y de las prácticas de desarrollo adoptadas (mejores prácticas), tiene un impacto negativo tanto en el proceso de desarrollo como en la calidad del producto final[22].

Las violaciones a las buenas prácticas se detectan mediante revisiones manuales que bien pueden no realizarse bajo ciertas situaciones o, en caso de que se lleven a cabo, suelen ser más propensas a errores a medida que va aumentando la cantidad de buenas prácticas a controlar. El costo de reparar algunos de estos errores de manera aislada suele ser insignificante, pero a medida que los mismos se van acumulando, puede significar una pérdida en la calidad del software[2] o de una cantidad considerable de tiempo y esfuerzo, dedicados a arreglar el problema.

La gestión de conocimiento (Knowledge Management [5][19]) estudia la recuperación, acceso y mantenimiento del conocimiento de las organizaciones, que es un recurso clave que poseen ya que les permite incrementar su productividad y competitividad. El objetivo de la Web Semántica [1] es proporcionar sistemas de gestión de conocimiento [5][19] más avanzados, permitiendo que los contenidos o documentos de la web sean legibles por una máquina.

Para lograr esto, propone organizar el conocimiento conceptualmente mediante modelos semánticos, para poder recuperarlo, procesarlo y generar nuevo conocimiento de manera automática.

Contando con un modelo semántico [1] de las buenas prácticas y de las actividades realizadas en el proyecto, el conocimiento disponible puede ser fácilmente recuperado o explotado por un agente inteligente, para prevenir

estas situaciones no deseadas.

La motivación de este trabajo radica en proveer una plataforma que permita modelar y gestionar las buenas prácticas (con una definición semántica detallada de las mismas y de los procedimientos que involucran) y a su vez, mediante la recolección de información de las actividades del proyecto, logre detectar en el momento la ocurrencia de violaciones a las buenas prácticas definidas, y facilite su corrección mediante la distribución del conocimiento de la organización en el momento en que es relevante a las personas involucradas, dando como resultado un proceso de desarrollo más confiable y eficiente.

1.1. Organización

En el primer capítulo se introduce el problema a resolver, que es la falta de conocimiento que tiene cada persona involucrada en un proyecto de desarrollo de software sobre las tareas que están llevando a cabo el resto de los integrantes y sobre las mejores prácticas adoptadas de manera implícita o explícita, y una posible solución utilizando un modelo semántico para modelar las mejores prácticas y controlar que se utilicen correctamente.

En el segundo capítulo se define la gestión del conocimiento y su ciclo, se propone un ciclo de gestión de las mejores prácticas entendidas como una forma de conocimiento que tiene en cuenta las necesidades de un grupo de desarrollo de software. Adicionalmente, se introducen tres ejemplos de mejores prácticas, que se continuarán desarrollando incrementalmente en los próximos capítulos teniendo en cuenta la temática presentada en cada uno de ellos.

En el tercer capítulo se enumeran los requerimientos de un sistema que soporte la gestión de las mejores prácticas teniendo en cuenta el ciclo de gestión desarrollado en el capítulo anterior y las características deseadas de la solución al problema de la falta del conocimiento necesario para desarrollar software con una mejor calidad.

En el cuarto capítulo se describe la Web Semántica y sus componentes, incluyendo las principales tecnologías y lenguajes que contiene, para explicar el rol que cumplen en la gestión de las mejores prácticas.

En el quinto capítulo se mencionan y describen brevemente algunos trabajos que tienen un objetivo similar o complementario a éste. Los más importantes son el escritorio semántico, de donde se toma el mecanismo de generación de los datos de las acciones realizadas por los usuarios o sistemas de apoyo al desarrollo de software, y SEON, la ontología utilizada para de-

scribir los elementos del dominio de un proyecto de desarrollo de software.

En el sexto capítulo se describe en términos generales la herramienta Singapur que permite la gestión de las mejores prácticas, mostrando su arquitectura interna y su colaboración con los otros sistemas necesarios para su funcionamiento. También se destaca el papel que cumple la tecnología semántica para su funcionamiento.

En el séptimo capítulo se detalla el modelo semántico del que dispone Singapur, teniendo en cuenta los diversos elementos que contiene (mejores prácticas, elementos del dominio de un proyecto de desarrollo de software, etc). También se explica cómo se generarían los datos, realizando extensiones a las herramientas utilizadas para el desarrollo de software, siguiendo el enfoque del escritorio semántico.

En el octavo capítulo, se describen los mecanismos utilizados para la detección y reporte de las violaciones a las buenas prácticas, y se describe el motor de inferencia mediante reglas de Apache Jena, junto con la sintaxis de las reglas para poder comprender en mayor profundidad cómo son detectadas las violaciones a las buenas prácticas.

En el noveno capítulo se describe un experimento, registrando sus resultados, para luego poder evaluar el funcionamiento de Singapur y determinar si cumplió y en qué grado con su objetivo de generar y difundir el conocimiento de las mejores prácticas y las violaciones a las mismas.

En el último capítulo se extraen conclusiones sobre el aporte de Singapur a la gestión de mejores prácticas y otras conclusiones sobre el trabajo realizado, así como también los temas a tener en cuenta como trabajo futuro.

Capítulo 2

Gestión del Conocimiento

En la actualidad se reconoce al conocimiento como un recurso muy valioso para las organizaciones, y por lo tanto la habilidad de gestionarlo es crucial [5]. El conocimiento posee las siguientes características que lo diferencian radicalmente de otros recursos importantes:

- El uso del conocimiento no lo consume.
- La transferencia del conocimiento no resulta en su pérdida.
- El conocimiento es abundante, pero la habilidad para usarlo es escasa.
- La mayor parte del conocimiento de una organización, que se encuentra en las personas que la integran, sale de la misma al final del día.

El único avance sostenible que una organización puede tener viene de su conocimiento colectivo, de cúan eficientemente lo usa, y de cúan rápidamente lo adquiere. Una organización en la actual era del conocimiento, es una que aprende, recuerda, y actúa basándose en el conocimiento que tiene disponible[3]. Por lo tanto hay una gran necesidad de tener un enfoque deliberado y sistemático para generar y compartir el conocimiento organizacional. En otras palabras, para que una organización cumpla sus objetivos en el contexto actual, debe aprender de los errores del pasado y no reinventar la rueda una y otra vez. El conocimiento organizacional no debe reemplazar al individual, sino complementarlo mejorándolo, haciéndolo más consistente y generalizando su aplicación.

La gestión del conocimiento es la coordinación deliberada y sistemática de las personas, tecnologías, procesos y estructuras que forman parte de una

organización, con el fin de dar un valor agregado a través de la reutilización e innovación[9]. Esta coordinación se alcanza mediante la creación, el intercambio y la aplicación del conocimiento, y mediante el almacenamiento de las valiosas lecciones aprendidas y las mejores prácticas en la memoria organizacional con el fin de fomentar el aprendizaje dentro de la organización.

Algunos de los objetivos principales de la gestión del conocimiento son:

- Identificar los recursos y áreas de conocimientos que son críticos para que la organización sea conciente de sus conocimientos, cumpla bien sus objetivos, y sepa por qué los cumple.
- Facilitar la transferencia del conocimiento de la gente que se retira hacia la gente que es reclutada para ocupar sus puestos.
- Minimizar la pérdida de memoria organizacional debido al retiro y la renuncia de los miembros.
- Definir un conjunto de métodos que puedan ser utilizados con individuos, grupos y la organización, para contener la pérdida potencial de capital intelectual.

2.1. Ciclo de Gestión del Conocimiento

La gestión efectiva del conocimiento requiere que una organización identifique, genere, adquiera, difunda y obtenga los beneficios del conocimiento que le provee una ventaja estratégica. Se debe hacer una distinción entre la información, que se puede digitalizar, y el conocimiento, que sólo puede existir en el contexto de un sistema inteligente. Debido a que todavía estamos lejos de la creación de verdaderos sistemas de inteligencia artificial [5], el conocimiento reside en las personas conocedoras, y no en la organización misma.

El ciclo de información del conocimiento puede ser visto como la ruta que la información recorre para transformarse en un valioso recurso estratégico para la organización, mediante un ciclo de gestión del conocimiento. Debido a que la gestión del conocimiento es un campo multidisciplinario, existen distintas visiones sobre los procesos que involucra el ciclo de gestión del conocimiento, pero todas tienen en común las siguientes fases, que conforman el ciclo integrado de gestión del conocimiento:

- Captura y/o creación de conocimiento.

- Difusión del conocimiento.
- Adquisición y aplicación del conocimiento.

En la transición del conocimiento desde su captura y/o creación a su difusión, se evalúa su utilidad. Luego se contextualiza para ser entendido (adquisición) y utilizado (aplicación). Posteriormente se retroalimenta la etapa de captura y/o creación para actualizar el conocimiento. En la figura 2.1 se ilustra el ciclo integrado de gestión del conocimiento.

La captura del conocimiento tiene que ver con identificar y codificar el conocimiento interno de la organización y el conocimiento externo del entorno. La creación del conocimiento es el desarrollo de nuevo conocimiento e innovaciones que no tenían existencia previa dentro de la organización.

Cuando el conocimiento es inventariado de esta manera, el siguiente paso crítico es evaluar su utilidad con criterios alineados a los objetivos de la organización. Se debe tener en cuenta si el conocimiento es válido, nuevo o mejor que el conocimiento previo, y si posee el suficiente valor como para que la organizacion lo difunda y lo almacene como capital intelectual.

Luego se debe contextualizar el conocimiento. Esto incluye mantener una relación entre el conocimiento y sus conocedores (sus autores y los que lo utilizan). La contextualización también incluye la identificación de los atributos fundamentales del contenido para identificar mejor a la diversidad de usuarios que puede tener. Generalmente, la contextualización será exitosa cuando el nuevo contenido es integrado sin problemas en los procesos de la organización.

Posteriormente se reitera el ciclo a medida que los usuarios comprenden y deciden hacer uso de los nuevos contenidos generados. Los usuarios validarán su utilidad y alertarán cuando se encuentren desactualizados o no sean aplicables. También aportarán nuevos contenidos, que pueden contribuir a la próxima iteración del ciclo integrado de gestión del conocimiento.

2.2. Las mejores prácticas como productos de conocimiento

Las mejores prácticas son métodos, procesos o técnicas que presentan resultados superiores a los alcanzados por otros medios. A pesar de lo que indica el nombre de mejores prácticas, éstas puede evolucionar a lo largo del tiempo alcanzando mejores resultados. Poseen un conjunto de requisitos, costos y beneficios, que se deben evaluar cuidadosamente para determinar

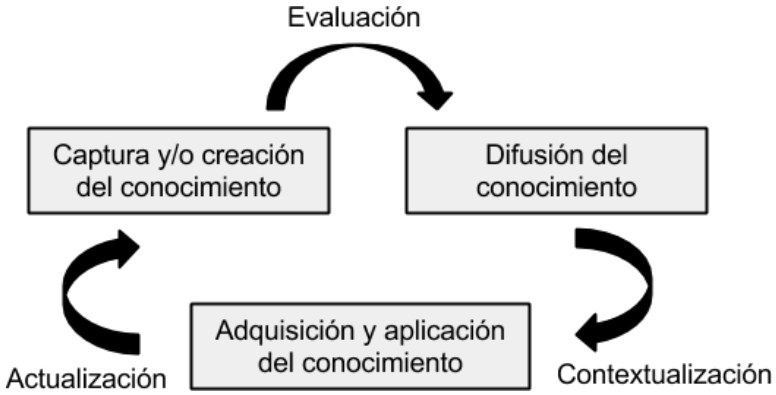


Figura 2.1: Ciclo Integrado de Gestión del Conocimiento

en qué situaciones aplicarlas. Generalmente se utilizan para alcanzar un nivel deseado de calidad como una alternativa a los estándares, y suelen estar basadas en experiencias y mediciones propias a un contexto en particular.

Debido a que son fundamentales para el aseguramiento de la calidad, y la calidad es muy importante para cualquier organización en la actualidad, brindan una gran ventaja estratégica para cumplir con los objetivos organizacionales, y por lo tanto deben ser gestionadas sistemáticamente. El costo que tiene para una organización no gestionar correctamente las mejores prácticas es muy grande. Por un lado se puede perder el conocimiento generado para solucionar un problema recurrente, teniendo que generarlo nuevamente ante cada nueva ocurrencia del mismo, perdiendo soluciones previas que pueden ser potencialmente mejores y privando a la organización de realizar una mejora contínua de sus procesos. Por otro lado, el conocimiento queda en las personas que realizan el trabajo y no en la organización misma, haciendo que su difusión y permanencia dentro de la organización dependa únicamente de los trabajadores. La consecuencia final de perder o no transmitir el conocimiento generado es la pérdida de calidad del producto o servicio brindado.

2.3. Mejores prácticas en el contexto de un proyecto de desarrollo de software

A continuación se presentan algunos ejemplos de mejores prácticas que iremos desarrollando a lo largo de este trabajo. Se plantean en el contexto de un proyecto de desarrollo de software con varios desarrolladores donde se usen herramientas de control de versiones del código fuente y un servidor de integración contínua. Todas ellas tienen en común que hacen aportes en la calidad del proceso de desarrollo del software y se encuentran en un estado muy maduro, debido a que son recomendaciones consitentes a lo largo del tiempo que vienen de técnicas y metodologías que surgieron a partir de la experiencia de desarrolladores de software de todo el mundo.

2.3.1. Integración exitosa del código fuente

Para que cada commit del código se encuentre integrado exitosamente en el servidor de integración contínua, debe tener un build en estado exitoso o debe haber un commit posterior que lo tenga.

Ésta práctica es para asegurar que luego de realizar una modificación al código fuente de la aplicación que se está desarrollando, se realizan todas las comprobaciones al código configuradas en el servidor de integración contínua[1] y que si alguna falla, es arreglada en el momento o se deshacen los cambios realizados, de manera que siempre haya una versión estable de la aplicación.

2.3.2. Evitar conflictos en las modificaciones al código fuente

Al haber más de un desarrollador modificando un archivo al mismo tiempo, los mismos deben saberlo.

El objetivo de ésta práctica es que no surjan conflictos al momento de integrar los distintos cambios que realizaron o para poder organizarse mejor en el orden que realizan las modificaciones. Surge a partir de la falta de conocimiento acerca de lo que está realizando cada participante en la tarea compartida (workspace awareness [15]) de desarrollar una aplicación.

[1] Generalmente éstas comprobaciones incluyen la detección de errores de compilación y la ejecución de tests de unidad, integración o de otro tipo.

2.3.3. Cobertura de tests completa

Cada línea de código debe estar probada en un test, es decir, que debe haber una cobertura de tests del 100 % del còdigo.

De esta manera es más probable que cada modificación que altere el comportamiento esperado de una sección de código sea detectada, pudiendo corregirla en el momento. Como resultado final se obtiene un producto de software de mejor calidad en cuanto a que reduce la presencia de errores e incrementa su confiabilidad al realizarle una modificación.

2.4. Un modelo para la gestión de mejores prácticas

A continuación se describen los pasos de un ciclo para dar soporte a la gestión de las mejores prácticas:

2.4.1. Creación/Identificación

La creación de las buenas prácticas surge de la investigación y/o desarrollo dentro del proyecto de desarrollo de software, que se da al identificar los requerimientos y la necesidad de identificar, idear o investigar procesos para cumplirlos. Parte del proceso de creación es el análisis de los requisistos y necesidades para aplicar una práctica, y sus beneficios concretos. Por ejemplo, si se desea que una aplicación sea muy confiable (que tenga la menor cantidad de fallos posibles frente a las distintas situaciones que puedan presentarse), una buena práctica que se puede encontrar fácilmente investigando en internet es la de codificar pruebas automatizadas sobre los módulos desarrollados, que se puedan ejecutar luego de cada cambio para identificar si alguna funcionalidad presenta problemas.

2.4.2. Codificación

Al ser las mejores prácticas un conocimiento de tipo explícito[2], se pueden codificar en documentos detallando el procedimiento paso a paso, junto con

[2]El conocimiento explícito es aquel conocimiento que ha sido o puede ser articulado, codificado y almacenado en algún tipo de medio. Puede ser transmitido inmediatamente a otros. Por ejemplo, la información contenida en enciclopedias es un conocimiento de tipo explícito.

las condiciones que se deben ir cumpliendo en cada uno de ellos y de qué manera se debe actuar frente a los distintos inconvenientes que puedan surgir. También se pueden agregar diagramas, otros recursos multimediales, descripciones formales en distintos lenguajes y referencias a bibliografía o documentación para ayudar en su comprensión. En la codificación se debe tener en cuenta que debe ser fácil de comprender por las personas que lo leerán.

2.4.3. Evaluación

Las mejores prácticas deben ser evaluadas para verificar que se cumplen los objetivos esperados y que no surgen problemas o situaciones no deseadas, con el fin de decidir si deben ser almacenadas y difundidas. Por ejemplo, hay ciertas buenas prácticas que sólo funcionan bien si los desarrolladores tienen un nivel avanzado de programación o si el proyecto cumple con algunas características especiales. De una buena evaluación depende que se utilicen o no las mejores prácticas descriptas y de que finalmente, se obtengan o no los resultados esperados.

2.4.4. Intercambio y difusión

Se deben almacenar las buenas prácticas que pasan la evaluación en un repositorio, como también deben poder ser accedidas por cualquiera que las necesite. La difusión se debe realizar a las personas involucradas con las tareas que tratan las buenas prácticas, y deben poder disponer de ellas en los momentos que las necesiten.

2.4.5. Contextualización

De cada una de las buenas prácticas se deben conocer sus autores, referentes de el o los temas que trata y las personas que han ganado una cierta experiencia en su uso. Además, se deben identificar los aspectos clave de la buena práctica y adaptarla al uso que se le va a dar, es decir, que la gente que la va a llevar a cabo debe poder hacerlo sin mayores inconvenientes. Para esto se debe tener en cuenta el lenguaje y la terminología que que se utiliza, la complejidad de las tareas que involucra, y la relación costo beneficio que supone.

2.4.6. Adquisición y aplicación

La adquisición y aplicación del conocimiento se da en la difusión de las buenas prácticas a la gente que las necesita, o en la medida que se encuentran

los problemas que solucionan las buenas prácticas y la gente consulta el repositorio de buenas prácticas o a los expertos en el tema.

2.4.7. Retroalimentación

Las personas que utilizan las buenas prácticas y las que perciben su impacto, deben proveer una devolución positiva o negativa para actualizar el conocimiento que se tiene de las buenas prácticas y poder modificarlas, iniciando nuevamente el ciclo de gestión de las buenas prácticas. Ésto permite entrar en un ciclo de mejora contínua de los procesos de la organización, brindando grandes ventajas estratégicas a la misma.

Capítulo 3

Requerimientos de un sistema de gestión de mejores prácticas

La motivación de este trabajo radica en proveer una plataforma que permita modelar y gestionar las buenas prácticas (con una definición semántica detallada de las mismas y de los procedimientos que involucran) y a su vez, mediante la recolección de información de las actividades del proyecto, logre detectar en el momento la ocurrencia de violaciones a las buenas prácticas definidas, y facilite su corrección mediante la distribución del conocimiento de la organización en el momento en que es relevante a las personas involucradas, dando como resultado un proceso de desarrollo más confiable y eficiente.

3.1. Modelado del dominio del problema

Un sistema de gestión de mejores prácticas debe modelarlas permitiendo su descripción, almacenamiento, modificación y recuperación. También se debe modelar un ciclo de vida con distintos estados para poder diferenciar las distintas etapas por las que pasan, como es una buena práctica que se está investigando, una que se descartó (junto con los motivos), una que se utiliza hace mucho tiempo y posee un gran nivel de madurez, una que se aplicó con éxito en un caso pero no se sabe bien si funcionaría en otros casos, etc.

La información indispensable para describir una buena práctica es:

- Título
- Descripción
- Requerimientos
- Beneficios
- Desventajas
- Autor/es
- Referentes con experiencia en su uso
- Estado actual en el ciclo de vida
- Comentarios (obtenidos de la retroalimentación)
- Descripción de las violaciones que pueden surgir y por cada una, las situaciones que la originan

Además, se debe disponer de una descripción de los elementos prosentes en el dominio de un proyecto de desarrollo de software. Como por ejemplo las personas involucradas, el código, las tareas, etc. A diferencia de las mejores prácticas, la mayor parte de este tipo de información no debe ser gestionada directamente por un usuario del sistema, sino que debe haber agentes que importen automáticamente la información de otros sistemas. Ciertos datos como las personas involucradas en el proyecto u otros datos que pueden no estar disponibles en ningún sistema deben poder ser gestionados por un usuario. Un ejemplo de un agente de importación automática de la información, puede ser el caso del código fuente y sus versiones, que se debe leer desde el sistema de manejo de versiones y ser agregado a Singapur.

Teniendo un modelo de buenas prácticas y un modelo de los elementos de un proyecto de desarrollo de software, se debe poder modelar las situaciones en que ocurre una violación a una buena práctica en función del modelo de un proyecto de desarrollo de software y las violaciones en sí. Para lograr esto, es necesario que el modelo permita una gran flexibilidad en cuanto a la definición e integración de datos debido a que las fuentes de información pueden ser muy diversas, variables en el tiempo e incluso contener distintas definiciones para un mismo concepto. Lo que es más, debería ser posible definir violaciones a las mejores prácticas en función de conceptos abstractos cuyas instancias más concretas todavía no se encuentran definidas o pueden tener diferencias respecto del concepto original.

3.2. Detección de violaciones a las buenas prácticas y difusión de la información

Teniendo en cuenta el modelo descripto anteriormente, Singapur debe detectar las violaciones a las mejores prácticas en el momento en que ocurren, instanciarlas en el modelo e informar a los usuarios involucrados para que puedan corregir la situación. También debe encargarse de actualizar las violaciones que son corregidas para que se sepa que se revirtió la situación no deseada.

En base al modelo de las violaciones, debe ser posible agregar agentes que lean los datos de las violaciones ocurridas para integrar éste conocimiento con las herramientas utilizadas en el desarrollo de software. Por ejemplo, en el caso de una buena práctica que recomiende que no haya dos programadores modificando al mismo tiempo la misma clase, se debería poder extender el IDE[1] utilizado para el desarrollo para que al ver en el modelo de Singapur que hay una violación que informa que además de uno mismo hay más personas modificando una clase, aparezca un cartel customizado (dentro del mismo contexto del IDE) informando la situación. De esta manera se brinda al programador una información muy valiosa y de una manera muy eficaz[2] que le permite evitar futuros inconvenientes.

3.3. Roles del sistema

En Singapur hay dos roles posibles para los usuarios:

- El usuario común, cuya información y la de sus acciones se almacena en el modelo, y si se encuentra envuelto en una violación a las mejores prácticas recibe la notificación del sistema. También puede dar una retroalimentación sobre su experiencia en la utilización de una buena práctica.

[1] Un Ambiente Integrado de Desarrollo (Integrated Development Environment) es una aplicación que integra una gran variedad de herramientas para facilitar el desarrollo de software. Entre éstas herramientas generalmente se incluyen editores de código fuente, compiladores, depuradores y autocompletado inteligente de código.

[2] Al poder visualizar la información de la violación en el mismo contexto de trabajo y en el mismo momento en que ocurre, teniendo además la posibilidad de presentarla de la manera que se considere más conveniente, resulta muy difícil que el usuario no se entere de la ocurrencia de la violación a las mejores prácticas y cómo se debe resolver.

- El usuario gestor de mejores prácticas, incluye las responsabilidades del usuario común y además es responsable de realizar la evaluación y el seguimiento de las mejores prácticas a través de su ciclo de vida, teniendo en cuenta las retroalimentaciones recibidas.

3.4. Retroalimentación

Los usuarios de las buenas prácticas y la gente que evalúa los procesos en que se utilizan deben poder dar una retroalimentación de los resultados obtenidos y los problemas o inconvenientes detectados, con el fin de mejorar y actualizar contínuamente la calidad del conocimiento sobre las mejores prácticas.

Capítulo 4

Web Semántica

La Web es uno de los repositorios públicos de información más grandes del mundo. Su origen cambió radicalmente la forma en que se comunican las personas entre sí, y forma parte de un cambio del mundo desarrollado hacia una sociedad en la que el conocimiento tiene un rol fundamental[6]. Éste cambio llevó a replantear las funciones de las computadoras. Inicialmente se las utilizaba para realizar cálculos numéricos, y en la actualidad su función principal es el procesamiento de información.

Hoy en día la Web consta de miles de millones de sitios web, lo que representa una gran cantidad de información. Desafortunadamente, la mayoría de los contenidos que se encuentran en la web hoy en día están pensados para ser encontrados y consumidos por humanos (incluso los que son generados dinámicamente), y resulta difícil desarrollar software que realice estas tareas, debido a que los programas no están diseñados para interpretar los conceptos y las relaciones contenidas en la información que procesan. Por lo tanto, es muy difícil que las aplicaciones actuales aprovechen a la Web como fuente de información de manera automática.

La Web Semántica [1] surge como respuesta a esta problemática y propone estructurar la información, aprovechando técnicas de Inteligencia Artificial, para que pueda ser procesada fácilmente por una máquina.

4.1. Componentes

La Web Semántica está formada por un conjunto de componentes principales que son las sentencias, los identificadores, los lenguajes y las ontologías, y un conjunto de herramientas asociadas para gestionarlos[17].

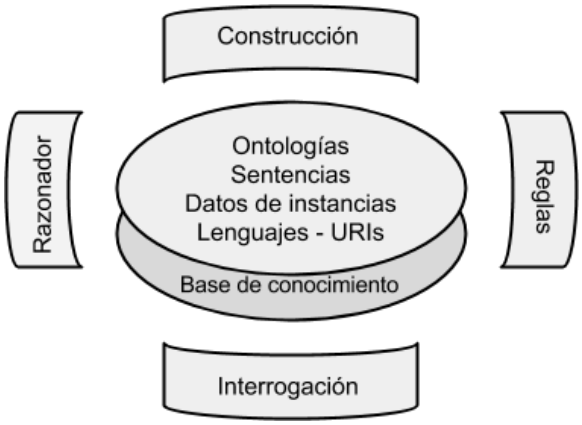


Figura 4.1: Componentes y herramientas de la Web Semántica.

4.1.1. Sentencia

Las sentencias son la base de la Web Semántica. Se encuentran formadas por múltiples elementos que generalmente forman una tripleta, que consta de un sujeto, un predicado y un objeto, por ejemplo, Juan es Humano. Las sentencias tienen una estructura simple que posee un gran poder de expresión, para realizar descripciones complejas, se pueden realizar composiciones de miles e incluso millones de sentencias relacionadas. Son utilizadas para definir la estructura de la información y las instancias específicas de los datos. Relacionándolas entre sí, e interconectando distintos juegos de datos, forman la red de datos que constituye la Web Semántica.

4.1.2. Datos de instancias

Son el conjunto de sentencias que representan información de instancias específicas en vez de conceptos abstractos. Es análogo a las instancias u objetos en el paradigma de programación orientada a objetos. Por ejemplo, Juan es una instancia, mientras que Persona es un concepto abstracto.

4.1.3. Ontología

Una ontología consta de un conjunto de sentencias que definen conceptos, relaciones y restricciones. Es análoga a un diagrama de clases en el paradigma de programación orientada a objetos. Existen muchas ontologías que pueden utilizarse, o pueden extenderse o adaptarse a las necesidades

específicas que puedan surgir. Una ontologia muy difundida y que utilizaremos para los ejemplos es FOAF[1]. Una ontología efectiva fomenta la comunicación entre las aplicaciones desde la perspectiva del modelo de dominio de la misma[17].

4.1.4. URI (Uniform Resource Identifier)

Una URI es un nombre único (en Internet) que identifica un elemento contenido en una sentencia. Sirve para determinar si dos elementos son el mismo o no. Puede incluir una URL (Uniform Resource Locator) que puede ser desreferenciada para obtener información adicional sobre el recurso. También posibilita tener espacios de nombres para que no haya conflictos entre distintas organizaciones que crean datos del mismo tipo.

4.1.5. Lenguaje

Las sentencias son expresadas de acuerdo a un lenguaje de Web Semántica, que consta de un conjunto de palabras clave que proveen instrucciones a diversas herramientas. Cada uno tiene su grado de complejidad y expresión semántica.

4.1.6. Herramientas

Para formar una red semántica, se necesitan herramientas de distintos tipos. Las herramientas de construcción permiten contruir aplicaciones que utilicen datos semánticos, las de interrogación explorar los datos semánticos, los razonadores agregar inferencia y los motores de ejecución de reglas expandir los datos semánticos. En un nivel superior se encuentran los frameworks que integran todas estas herramientas.

Herramientas de construcción

Éstas herramientas permiten a las aplicaciones construir una red semántica mediante la creación de sentencias e instancias de una ontologia. Varias herramientas proveen una interfaz gráfica para explorar y modificar los datos semánticos, otras proveen una API[2] para integrarlas con cualquier

[1] FOAF (Friend Of A Friend, literalmente Amigo de un Amigo) es una ontología que describe a las personas, sus actividades y sus relaciones con otras personas y objetos.

[2] Una Interfaz de Programación de Aplicaciones, abreviada como API (del inglés: Application Programming Interface), es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

aplicación.

Herramientas de interrogación

Las herramientas de interrogación navegan una red semántica para devolver una respuesta a una petición. Se pueden utilizar varios métodos de interrogación que van desde la simple navegación de un grafo, a la búsqueda mediante un lenguaje complejo de consultas. Éstas herramientas son de fundamental importancia en la Web Semántica debido a que su ventaja respecto de los otros modelos de datos, radica en la capacidad del software de interpretar los datos semánticos[11].

Razonadores

Los razonadores agregan la posibilidad de realizar inferencia sobre los datos semánticos. La inferencia agrega relaciones a los datos, en base a deducciones lógicas, que ofrecen clasificación y realización. La clasificación agrega relaciones entre conceptos, por ejemplo, si Juan es una persona también es un ser vivo. La realización permite identificar cuando dos conceptos o instancias representan lo mismo, por ejemplo, Juan Carlos Perez es la misma persona que Juan C. Perez. Hay distintos tipos de razonadores que ofrecen distintos niveles de razonamiento y generalmente se pueden conectar a otras herramientas.

Motores de Ejecución de Reglas

Los motores de ejecución de reglas ofrecen inferencia que va más allá de lo que puede ser deducido lógicamente. Agregan un poderoso mecanismo para construir conocimiento. Las reglas permiten la integración automática de distintas ontologías u otras tareas de modificaciones lógicas al conocimiento como realizar agregación de datos, y son consideradas una parte de la representación del conocimiento. Cada motor de reglas utiliza un lenguaje para expresar sus reglas.

4.2. Modelado de la información

Todos los sistemas deben ser diseñados con algún tipo de modelo, y las características de dicho modelo dependen de los objetivos del sistema y del contexto para el cual el mismo es diseñado. En los sistemas de información, el modelado de la misma es fundamental debido a que su representación

determina en gran medida las operaciones que el sistema puede realizar y las preguntas que puede responder[17]. En un sistema semántico es todavía más importante debido a que si se modelan los conceptos con el rigor suficiente, es posible reusarlos en otros sistemas.

La información de la Web Semántica se encuentra modelada principalmente con tres lenguajes complementarios, RDF (Resource Description Framework), RDFS (RDF Schema) y OWL (Web Ontology Language).

4.2.1. RDF

RDF define el modelo de datos subyacente y provee una base para las funcionalidades más sofisticadas que presentan soluciones de alto nivel.

En la Web Semántica, la información es representada como un conjunto de aserciones llamadas sentencias, que están compuestas de 3 partes: sujeto, predicado y objeto. El sujeto es la cosa que describe la sentencia y el predicado describe una relación entre el sujeto y el objeto. El objeto de la sentencia puede ser un sujeto o un dato, como un número o una cadena de caracteres.

Ejemplos de sentencias:

- Andres conoce a Matias.
- Andres tiene apellido Perez.
- Matias conoce a Juan.
- Roberto trabaja con Juan.

Grafo RDF

Los nodos de un grafo RDF (Figura 4.2.1) son los sujetos y objetos de las sentencias que arman el grafo. Hay dos tipos de nodos, los recursos y los valores literales. Los últimos representan valores concretos como números y cadenas de caracteres, y no pueden ser sujetos de las sentencias, sólo objetos. Los recursos, en cambio, representan todo lo demás, y pueden ser tanto sujetos como objetos. Un recurso representa cualquier cosa que pueda ser nombrada o identificada, es decir, que es un nombre que representa un objeto, una acción o un concepto. Los nombres de los recursos pueden ser cualquier tipo de URI.

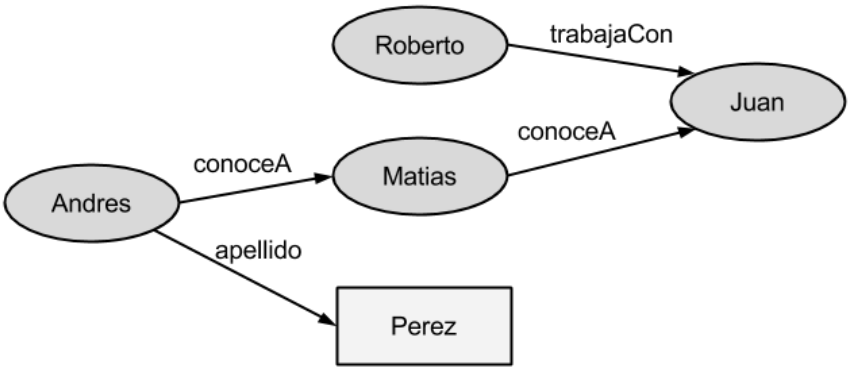


Figura 4.2: Representación de grafo del conjunto de sentencias de ejemplo.

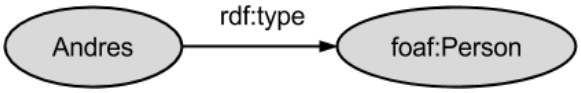


Figura 4.3: A Andres se le asigna el tipo foaf:Person.

Los predicados, o propiedades, representan las conexiones entre recursos y son recursos en sí mismos, por lo tanto se pueden hacer sentencias sobre los predicados como cualquier otro recurso. De la misma manera que los sujetos, los predicados son representados con URIs.

RDF define un predicado llamado rdf:type que se use para agrupar recursos. En la figura 4.2.1 asumimos que Andres, Matias, Roberto y Juan son personas, pero eso no se encuentra modelado, para ello se utiliza el predicado rdf:type que asocia a un recurso con otro recurso que representa el concepto de persona como en la figura 4.2.1.

La noción de tipo del predicado rdf:type es muy permisiva, en sí es un predicado como cualquier otro que se pueda definir. Puede haber más de un predicado de este estilo (o ninguno) relacionando a un mismo recurso con distintos conceptos.

Serializaciones de RDF

Los grafos RDF son herramientas muy poderosas para representar información, pero no son apropiados para intercambiar datos entre aplicaciones[17]. La serialización hace que RDF sea práctico para el intercambio proveyendo

una manera de convertir entre el modelo abstracto y un formato concreto. Hay varios formatos de serialización igualmente expresivos, pero los más populares son RDF/XML, Turtle (Terse RDF Triple Language) y N-Triples.

Si bien todos los formatos representan los mismos conceptos (sentencias, URIs y valores literales), se diferencian en que cada uno los representa de una manera distinta, lo que genera que sean más o menos convenientes para distintas situaciones.

RDF/XML

Es una sintaxis XML para representar tripletas RDF y es el único formato estándar de intercambio de datos RDF, por lo tanto debe ser soportado por todas las aplicaciones que pretendan que sus datos sean publicados en la Web Semántica[17]. Si bien hay una línea que deben seguir las herramientas que serializan datos mediante RDF/XML, pueden haber ciertas diferencias entre las distintas implementaciones. Un inconveniente que tiene la serialización en RDF/XML surge debido a la estructura de árbol de XML, que resulta incompatible con un grafo RDF, por lo tanto se crea un nodo raiz RDF para su representación en XML.

En la figura 4.2.1 se ve que el nodo raíz es el tag rdf:RDF, que contiene una serie de elementos rdf:Description. Dentro del nodo raíz se declaran los namespaces, y debido a que RDF/XML es el estándar para el intercambio de datos RDF, el documento que define al RDF mismo se encuentra en formato RDF/XML (se puede bajar de internet en la url http://www.w3.org/1999/02/22-rdf-syntax-ns.rdf).

Comentarios

Los comentarios son representados como en cualquier otro documento XML. Empiezan con la secuencia <!—, y finalizan con —>. Los mismos no son parte del grafo.

Sentencias

En un documento RDF/XML, las sentencias sobre recursos se agrupan en elementos rdf:Description, que poseen un atributo rdf:about especificando el sujeto de todas las sentencias en él. Cada uno de los elementos dentro de rdf:Description definen el predicado y el objeto de la sentencia. El nombre de los tags internos representa el predicado de la sentencia. A continuación se puede ver la forma general de una sentencia en RDF/XML.

```
<rdf:RDF
  xmlns:personas="http://ejemplo.edu/personas#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/">

  <rdf:Description rdf:about="http://ejemplo.edu/personas#Roberto">
      <personas:trabajaCon rdf:resource="http://ejemplo.edu/personas#Juan
          "/>
  </rdf:Description>
  <rdf:Description rdf:about="http://ejemplo.edu/personas#Matias">
      <foaf:knows rdf:resource="http://ejemplo.edu/personas#Juan"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://ejemplo.edu/personas#Andres">
      <foaf:surname>Perez</foaf:surname>
      <foaf:knows rdf:resource="http://ejemplo.edu/personas#Matias"/>
  </rdf:Description>
</rdf:RDF>
```

Figura 4.4: Representación en RDF/XML del grafo RDF de la figura 4.2.1.

```
<rdf:Description rdf:about="sujeto">
    <predicate rdf:resource="objeto" />
    <predicate>valor literal</predicate>
</rdf:Description>
```

Recursos

Los recursos se tratan de manera diferente dependiendo de si son el sujeto o el objeto de una sentencia. Los sujetos de las sentencias se designan con el atributo rdf:about del tag rdf:Description, y los objetos aparecen en el atributo rdf:resource de los tags de los predicados. Como RDF/XML es XML, usa las convenciones estándar de namespaces para abreviar URIs completas que aparecen como elementos XML.

Literales

Los valores literales en RDF/XML (Figura 4.2.1) aparecen como el contenido de un elemento de predicado. Se les puede asignar un tipo de dato usando el estándar XSD (XML Schema Datatypes). De hecho, cualquier URI puede ser usada como tipo de dato para un alor literal, lo que permite definir nuevos tipos de datos.

Se puede usar el atributo xml:lang para indicar el lenguaje del texto en una cadena de caracteres y usar el atributo rdf:datatype para indicar cómo tratar los valores literales. El conjunto de lenguajes validos está limitado a los definidos en la RFC 3066 (disponibles en http://www.isi.edu/in-notes/rfc3066.txt).

Abreviaciones

RDF/XML permite abreviaciones a la sintaxis para asignar tipos a los recursos. En la figura 4.2.1 se ven dos maneras equivalentes de asignar un tipo a un recurso, el elemento <foaf:Person rdf:ID="#Roberto"/>representa la misma sentencia que rdf:Description, que es que hay un recurso llamado "http://ejemplo.edu/personas#Roberto" que es de tipo foaf:Person.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
  xmlns:animales="http://ejemplo.edu/animales/">

  <rdf:Description rdf:about="http://ejemplo.edu/animales/Animal−123456">
      <animales:cantidadDePiernas rdf:datatype="http://www.w3.org/2001/
          XMLSchema#int">4</animales:cantidadDePiernas>
      <animales:nombreCientifico>Canis lupus familiaris</animales:
          nombreCientifico>
      <animales:nombreComun xml:lang="es">Perro</animales:nombreComun
          >
      <animales:nombreComun xml:lang="en">Dog</animales:nombreComun>
  </rdf:Description>
</rdf:RDF>
```

Figura 4.5: Valores literales en RDF/XML.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:base="http://ejemplo.edu/personas/">

  <foaf:Person rdf:ID="#Roberto">
  <rdf:Description rdf:about="#Roberto">
      <rdf:type rdf:resource="foaf:Person" />
  </rdf:Description>
</rdf:RDF>
```

Figura 4.6: Abreviación en RDF/XML.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix personas: <http://ejemplo.edu/personas/> .

# Esto es un comentario

personas:Roberto personas:trabajaCon personas:Juan .
personas:Matias foaf:knows personas:Juan .
personas:Andres foaf:knows personas:Matias ;
                foaf:surname "Perez" .
```

Figura 4.7: Datos serializados en Turtle.

Turtle (Terse RDF Triple Language)

Comparado con otros formatos de serialización, Turtle tiene una sintaxis más legible para las personas[17]. No es XML, fue diseñado específicamente para RDF, y por lo tanto puede representar un grafo RDF. En la figura 4.2.1 se pueden ver los datos de la figura 4.2.1 serializados en Turtle.

Comentarios

Turtle usa el caracter # al principio de una línea para indicar un comentario.

Sentencias

El formato de sentencias de turtle es simple. El sujeto, el predicado y el objeto son escritos en una línea, separados por un espacio en blanco, y la sentencias se termina con un punto. En la línea "personas:Roberto personas:trabajaCon personas:Juan ." del ejemplo de la figura 4.2.1 se puede ver que personas:Roberto es el sujeto, personas:trabajaCon es el predicado, personas:Juan es el objeto, y la sentencia termina con un punto.

Éste formato provee un método para abreviar múltiples sentencias sobre el mismo sujeto, que consiste en agregar ";" al final de la sentencia, indicando que los próximos dos elementos serán el predicado y el objeto de una sentencia que tiene el mismo sujeto que la sentencia previa. Se puede

observar un ejemplo en la figura 4.2.1. De esta manera Turtle resulta más legible y fácil de escribir.

Se puede utilizar un mecanismo de abreviación similar cuando varias sentencias tienen el mismo sujeto y predicado. Utilizando una coma al final de la sentencia, se indica que el próximo elemento es el objeto de una sentencia que tiene el mismo sujeto y predicado que la sentencia previa. Por lo tanto el ejemplo de la figura 4.2.1 se puede resumir a:

```
personas:Andres foaf:knows personas:Matias, personas:Roberto, personas:Juan .
```

Recursos

Los recursos pueden ser escritos de dos maneras. Las URIs aparecen completas entre los símbolos "<" y ">", o con un prefijo predefinido. En la figura 4.2.1 se puede observar la definición y utilización de prefijos en las sentencias:

```
@prefix personas: <http://ejemplo.edu/personas/> .

personas:Roberto personas:trabajaCon personas:Juan .
```

La primera línea declara el prefijo, que informa al parser de Turtle que para este documento cuando se encuentre personas:, debe ser expandido a la URI completa asociada con el prefijo. Ésta declaración del prefijo también muestra cómo se expresan las URIs completas en Turtle. En la segunda línea se ve cómo se usa el prefijo, y podría haber sido escrita sin usar el prefijo de la siguiente manera:

```
<http://ejemplo.edu/personas/Roberto> <http://ejemplo.edu/personas/
    trabajaCon> <http://ejemplo.edu/personas/Juan> .
```

Los prefijos hacen de Turtle un formato mucho más legible y fácil de escribir.

Literales

Los valores literales en Turtle se encuentran entre dobles comillas. Al igual que en XML, se puede especificar el tipo de datos XSD y el lenguaje de un literal. Los tipos de datos se indican agregando "textless datatype URI>" al final del literal. Los idiomas se especifican agregando "@idioma" al final del literal. En el siguiente ejemplo se puede ver cómo se especifican tanto el lenguaje como el tipo de datos de los literales.

```
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns\#> .
@prefix animales: <http://ejemplo.edu/animales/> .

animales:Animal animales:nombreComun "Perro"@es , "Dog"@en ;
            animales:cantidadDePiernas "4"^^<http://www.w3.org/2001/
                XMLSchema\#int> ;
            animales:nombreCientifico "Canis lupus familiaris" .
```

Abreviaciones

Turtle provee un mecanismo de abreviación en la designación del tipo de un recurso, la letra "a" se puede utilizar en lugar de "rdf:type". Ésta abreviación sirve para facilitar la lectura y escritura de los documentos Turtle. En el siguiente ejemplo se ve cómo se especifica el tipo de un recurso de manera completa y de manera abreviada (respectivamente), siendo ambas equivalentes.

```
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns\#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix : <http://ejemplo.edu/personas/> .

:Juan rdf:type foaf:Person .
:Juan a foaf:Person .
```

4.2.2. RDFS

RDF provee un modelo extremadamente expresivo para capturar información, sin embargo, no permite capturar el significado de la información

que modela[17]. Para ésto, es necesario desarrollar un vocabulario común, que tenga un significado y sea utilizado de forma consistente para describir recursos. RDFS (RDF Schema) es un lenguaje con el cual se pueden desarrollar éstos vocabularios compartidos, describiendo las clases de los recursos y las propiedades que se pueden usar en un modelo RDF. Permite armar jerarquías de clases y propiedades, definir dominios y rangos esperados de las propiedades, realizar aserciones sobre los recursos miembros de una clase y especificar tipos de datos.

Todos los recursos en RDFS son considerados miembros de la clase de todos los recursos RDF, y por lo tanto son instancias. Adicionalmente se puede describir a esas instancias mediante sentencias sobre ellas, usando propiedades, o haciéndolas miembro de una clase definida en un vocabulario RDFS.

RDFS es uno de los elementos fundamentales en la construcción de ontologías en la Web Semántica, y es el primer paso hacia la incorporación de semántica en un modelo RDF.

4.2.3. OWL

OWL (Web Ontology Language) extiende el vocabulario RDFS con recursos que pueden ser utilizados para construir ontologías más expresivas para la Web. Introduce restricciones adicionales para la estructura y los contenidos de los documentos RDF, que hacen que el procesamiento y razonamiento sean decidibles[23] [3].

Las ontologías en OWL se utilizan para modelar el conocimiento de un dominio y poseen las siguientes características, asunciones y estructuras:

- **Conocimiento distribuido**

  Los contenidos de la Web son inherentemente distribuidos, y por lo tanto, sus descripciones contenidas en la Web Semántica también lo son. OWL soporta este tipo de modelo de conocimiento distribuido porque está construido sobre RDF, que permite declarar y describir recursos de manera local o hacer referencia a recursos remotos. OWL también provee un mecanismo para importar y reutilizar ontologías en un ambiente distribuido.

- **Asunción de Mundo Abierto**

[3] En lógica, el término decidible se refiere a la existencia de un método efectivo para determinar si un objeto es miembro de un conjunto.

Para proveer una base sobre la cual realizar inferencias válidas en el modelo de conocimiento distribuido de la Web Semántica, es importante hacer la asunción del mundo abierto.

Ésta asunción tiene un impacto significativo en cómo se modela e interpreta la información. Lo que plantea ésta asunción es que la veracidad de una sentencia es independiente de si la misma es conocida o no, es decir, que no saber si existe una sentencia que diga explícitamente que algo es verdadero, no significa que ese algo sea falso. Con la asunción del mundo abierto, toda la información nueva es siempre aditiva, puede ser contradictoria, pero nunca puede borrar información previa.

- **No asunción de nombres únicos**

  La naturaleza distribuida de la descripción de recursos en la Web Semántica hace que no sea razonable asumir que todos utilicen la misma URI para identificar a un recurso específico. Generalmente cuando un recurso es descrito por varios usuarios en distintas ubicaciones, cada uno usa su propia URI para representar el recurso.

  La no asunción de nombres únicos significa que, a menos que se especifique lo contrario, dos recursos identificados con URIs distintas no son el mismo. Éste escenario es bastante diferente al de muchos sistemas tradicionales, y tiene un impacto en la capacidad de realizar inferencia. La redundacia y ambigüedad de los datos son problemas comunes en los sistemas de gestión de información, y ésta estrategia permite resolverlos de manera sencilla debido a que se puede especificar que dos recursos son el mismo sin destruir, borrar ni actualizar información.

Elementos de una ontología

Las ontología en OWL normalmente se guardan como documentos en la Web. Cada documento consta de un encabezado (opcional), anotaciones, definiciones de clases y propiedades (formalmente llamadas axiomas), hechos sobre individuos y definiciones de tipos de datos. Debido a que OWL se basa en el modelo de RDF, no hay una separación explícita entre la ontología y los datos que describe la misma. Por lo tanto es una práctica común mantener separadas las ontologías de los datos que describen.

A continuación se describen los elementos que forman una ontología, incluyendo aquellos que se reutilizan y extienden de RDFS:

- **Encabezado**

```
@prefix ode: <http://ontologiaDeEjemplo.org/>.

ode: rdf:type owl:Ontology;
     rdfs:comment ''Esta es una ontologia de ejemplo.'';
     owl:imports <http://ejemplo.org/importacion-de-ejemplo>.
```

Figura 4.8: Documento de ontología con un encabezado que contiene un comentario y una sentencia de importación.

El encabezado de una ontología es un recurso que representa a la ontología misma. No es un requisito que un documento de ontología tenga un encabezado, pero aporta información útil. El encabezado describe la ontología y generalmente contiene comentarios, etiquetas, información de la versión y sentancias de importación de ontologías.

La propiedad owl:imports especifica el conjunto de ontologías que son referenciadas en el documento de la nuevo ontología. Las sentencias con owl:import proveen a las herramientas la información necesaria para construir un modelo completo de todos los recursos referenciados en una ontología. El grado en que éstas sentencias son utilizadas depende de la herramienta.

Las propiedades owl:priorVersion, owl:backwardCompatibleWith y owl:incompatibleWith, proveen la base para la gestión de versiones.

En la figura 4.2.3 se puede ver un documento de ontología con un encabezado que contiene un comentario y una sentencia de importación.

- **Anotaciones**

  Las anotaciones son sentencias que describen recursos usando propiedades de anotaciones, que no poseen una semántica. Las propiedades de anotaciones predefinidas en OWL se pueden ver en la tabla 4.2.3, cualquiera de ellas puede ser utilizada para realizar anotaciones sobre clases, instancias, propiedades y axiomas.

- **Clases e individuos**

  Una de las forma más básicas de describir un objeto es colocándolo en una categoría, o clase, de cosas con las que comparte características

PROPIEDAD	DESCRIPCION
rdfs:label	Una etiqueta o descripción breve del recurso.
rdfs:comment	Un comentario del recurso.
owl:versionInfo	Información sobre la ontología o la versión del recurso.
rdfs:seeAlso	Se utiliza para especificar que otro recurso puede contener más información sobre el recurso.
rdfs:isDefinedBy	Se utiliza para especificar que otro recurso define a un recurso.

Cuadro 4.1: Anotaciones predefinidas en OWL.

comunes. En OWL, se puede definir clases y especificar la membresía de un objeto a dicha clase usando los recursos owl:Class y rdf:type. El recurso owl:Class representa la clase que contiene todas las clases de OWL. Cada clase en OWL debe ser miembro de owl:Class, y cada recurso que tiene un rdf:type de owl:Class es una clase. El recurso rdf:type es una propiedad que asigna membresia a una clase a un recurso específico.

Los miembros, o instancias, de una clases son referidos en OWL como individuos. El conjunto de individuos que son miembros de una clase es considerado como la extensión de la clase. El hecho de que un individuo pertenezca a un clase no le prohíbe pertenecer a cualquier otra clase, lo que significa que dos clases pueden tener exactamente la misma extensión y sin embargo representen clases únicas y distintas. Éste concepto es importante porque enfatiza que la equivalencia de extensión no es suficiente para asegurar la equivalencia de las clases. Dos clases pueden tener la misma extensión en un contexto particular, sin embargo, si son conceptos diferentes, sus extensiones pueden diferir en otro contexto.

El concepto de clase en OWL es ligeramente distinto al de la Programación Orientada a Objetos, en la cual los objetos derivan toda su información de sus clases. En OWL, si bien las clases describen un conjunto de individuos con características comunes, la estructura del individuo no es necesariamente determinada por las clases a las que pertenece, sino que puede tener cualquier estructura a pesar de ellas.

En el siguiente ejemplo se puede ver cómo se utilizan los recursos owl:Class y rdf:type para representar la clase Canino y Humano, y las instancias Pipo (que es un canino) y Roberto (que es un humano). No se realiza ninguna descripción de la clase, sólo se limita a declarar una

URI, del ejemplo sólo se puede saber que existe una clase Canino y una clase Humano, y que cada una posee una instancia:

```
@prefix ej: <http://ejemplo.org/>.

# Canino y Humano son clases OWL
ej:Canino rdf:type owl:Class.
ej:Humano rdf:type owl:Class.

# Pipo es una instancia de la clase Canino
ej:Pipo rdf:type ej:Canino.
# Roberto es una instancia de la clase Humano
ej:Roberto rdf:type ej:Humano.
```

Una definición de clase en OWL consta de algunas anotaciones opcionales seguidas de un conjunto de sentencias que restringen la pertenencia a la clase. Éstas restricciones representan descripciones de la clase y forman la base para la definición de la misma. Entre las posibles restricciones se encuentran la subclasificación, la enumeración explícita de miembros, las propiedades de restricción y las operaciones de conjuntos basadas en clases.

Para crear una subclase se utiliza la propiedad rdfs:subClassOf de la siguiente manera:

```
@prefix ej: <http://ejemplo.org/> .

ej:Mamifero rdf:type owl:Class .

ej:Canino rdf:type owl:Class;
          rdfs:subClassOf ej:Mamifero .

ej:Humano rdf:type owl:Class;
          rdfs:subClassOf ej:Mamifero .

ej:Pipo rdf:type ej:Canino .
ej:Roberto rdf:type ej:Humano .
```

Donde queda implícito que los miembros de ej:Canino y ej:Humano

también son miembros de ej:Mamifero, y que las propiedades y restricciones de ej:Mamifero son heredadas por ej:Canino y ej:Humano.

Para comprender las implicaciones de las relaciones de subclasificación en un entorno de gestión de conocimiento, considere una base de datos relacional con un esquema que represente a humanos y caninos como dos tablas separadas. Las aplicaciones que necesiten extraer el conjunto de mamíferos de la base de datos, debe combinar las entradas de ambas tablas. Por lo tanto, la semántica de la relación entre humanos y caninos queda capturada en la aplicación en vez de en el modelo. Un desarrollador entendió que los humanos y los caninos son mamíferos y escribió una consulta que combina ambas tablas para producir el resultado deseado.

En cambio, utilizando una Ontología, se puede extraer de la aplicación la semántica de ésta relación, y ponerla en el modelo de dominio, donde puede ser accedida por cualquier aplicación. Éste enfoque simplifica el desarrollo de aplicaciones porque separa a las aplicaciones del modelo de dominio, por lo tanto se pueden realizar cambios a la jerarquía sin requerir cambios a las aplicaciones que lo usan.

En OWL, hay dos clases fundamentales de las que todas las otras clases se derivan: owl:Thing y owl:Nothing. El recurso owl:Thing representa la clase de todos los individuos, y todos los recursos que son instancia de una clase son implícitamente miembros de owl:Thing. El recurso owl:Nothing representa la clase vacía, una clase que no tiene miembros. Desde un punto de vista taxonómico, owl:Thing es la clase más generalizada posible, y owl:Nothing es la más específica posible, porque es tan exclusiva que nada puede ser considerado un miembro.

- **Propiedades**

  En OWL las propiedades se utilizan para establecer relaciones entre recursos. Las dos clases fundamentales para esto son owl:ObjectProperty, la clase de todas las relaciones entre individuos, y owl:DatatypeProperty, la clase de todas las relaciones entre individuos y un valor literal.

  En el siguiente ejemplo se definen las propiedades ej:nombre y ej:raza, que son owl:DatatypeProperty y owl:ObjectProperty respectivamente:

```
@prefix ej: <http://ejemplo.org/> .

ej:nombre rdf:type owl:DatatypeProperty .
ej:raza rdf:type owl:ObjectProperty .
```

```
ej:Pipo ej:nombre "Pipo";
        ej:raza ej:GoldenRetriever.
ej:Roberto ej:nombre "Roberto" .
```

OWL permite describir las relaciones entre el dominio y el rango entre propiedades y clases o tipos de datos usando las propiedades rdfs:domain y rdfs:range.

- rdfs:domain especifica el tipo de todos los individuos que son el sujeto de sentencias que usan la propiedad que se describe.
- rdf:range especifica el tipo de todos los individuos o el tipo de datos de todos los valores literales que son el objeto de sentencias que usan la propiedad descripta.

OWL provee varias formas de describir las propiedades. De la misma manera que las clases, las propiedades pueden organizarse en taxonomías usando la propiedad rdfs:subPropertyOf.

- **Tipos de datos**

  Los tipos de datos representan rangos de valores que son identificados por una URI. OWL permite utilizar un conjunto predefinido de tipos de datos, mayormente definidos en el namespace xsd (XML Schema Definition). Los más utilizados son:

  - Valores numéricos: xsd:integer, xsd:float, xsd:real, xsd:decimal
  - String: xsd:string, xsd:token, xsd:language
  - Booleano: xsd:Boolean
  - URI: xsd:anyUri
  - XML: xsd:XMLLiteral
  - fecha: xsd:dateTime

  En la versión 2 de OWL se pueden definir nuevos tipos de datos de dos maneras, creando un nuevo rango de datos posibles, o en término de otros tipos de datos.

4.3. Rol en la Gestión de Mejores Prácticas

Tener un modelo de un dominio particular permite desarrollar herramientas que ayuden a resolver los problemas que surgen en el mismo. Utilizando un modelo semántico se tienen ciertas ventajas adicionales que no se poseen en ningún otro modelo disponible en la actualidad, y son las siguientes:

- Modelado de conceptos: El modelo de la Web Semántica define los conceptos y sus relaciones de una forma muy potente y expresiva, a diferencia de los otros modelos donde se definen tipos de datos o arreglos de datos y el significado se lo da el programador o cualquier persona que los interprete[17].
- Modelo de dominio fuera de la aplicación: El modelo de la aplicación (los conceptos que maneja) queda definido en la base de datos semántica en vez de codificado dentro de la aplicación.
- Fácil modificación de datos: Los cambios a la estructura de los datos resulta muchos más sencillos que en la mayoría de los otros modelos debido a que no hay un esquema fijo sino relaciones entre sujetos y objetos.
- Fácil integración de datos: La integración de los datos que vienen de distintas fuentes se simplifica[21] al poder declarar equivalencias entre distintas clases que representan el mismo concepto.
- Inferencia: Se puede aprovechar la inferencia lógica en los datos para descubrir nuevas relaciones de forma automática.

Por lo tanto si se realiza un modelo semántico del dominio de un proyecto de desarrollo de software y sus buenas prácticas, puede ser utilizado para detectar automáticamente (mediante inferencia lógica) la ocurrencia de violaciones a las buenas prácticas.

Capítulo 5

Trabajo Relacionado

En esta sección se describen brevemente los trabajos o proyectos que tienen un objetivo similar o complementario al de Singapur.

5.1. Escritorio Semántico y Nepomuk

Internet y la Web revolucionaron la comunicación, y con su adopción masiva surgió un nuevo problema: la sobrecarga de información debida a la falta de organización de la misma. La Web Semántica promete esta organización de la información, pero todavía no se adoptó masivamente en las computadoras personales[8].

El Escritorio Semántico consiste en aprovechar las tecnologías de la Web Semántica (ontologías, metadatos y protocolos) en las computadoras personales para poder organizar mejor la información personal, y mejorar la distribución y colaboración de la información en la Web. Por lo tanto, al contar con un modelo semántico para la información, resulta más sencillo compartir los datos entre distintas aplicaciones y automatizar el procesamiento de los mismos. Las nuevas posibilidades que brinda el Escritorio Semántico no se limitan sólo al procesamiento de los datos, sino también a la presentación de los mismos, ya que se puede reemplazar completamente la interfaz del usuario por una que integre de manera consistente toda la información disponible.

Nepomuk es una especificación del Escritorio Semántico que brinda un ambiente colaborativo que soporta la administración e intercambio de información mediante relaciones sociales y organizacionales. Es un proyecto de software libre finalizado en 2008, cuyos objetivos principales eran diseñar e implementar un conjunto de métodos, estructuras de datos y herramientas

para convertir la computadora personal en un ambiente colaborativo con el fin de aumentar el conocimiento disponible organizando la información creada por un grupo de personas. Como parte del proyecto se desarrolló una versión de Nepomuk en Java, para realizar pruebas de concepto. KDE realizó una implementación de Nepomuk en C++ que se incluye en la versión 4 de su entorno de escritorio.

La relación de este trabajo con el Escritorio Semántico, es que toma su enfoque de recolección de datos para generar la información de las acciones o tareas realizadas por los integrantes del proyecto de desarrollo de software que se almacenan en la base de datos de Singapur.

5.2. SEON (Software Evolution ONtologies)

SEON es un conjunto de ontologías creadas con el objetivo de facilitar la implementación de herramientas que ayuden a los ingenieros de software en la gestión del software en todo su ciclo de vida, y describe la mayoría de los conceptos importantes de este dominio, junto con sus relaciones. Éste conjunto de ontología fue publicada en el 2012 por el Laboratorio de Arquitectura y Evolución del Software del Departamento de Informática de la Universidad de Zurich.

Las ontologías se encuentran organizadas en una serie de capas incrementales que representan distintos niveles de abstracción en un esquema piramidal:

- La capa superior está compuesta de conceptos generales o independientes del dominio, los atributos que los describen y sus relaciones. Las clases definidas en este nivel se relacionan con los conceptos fundamentales para la evolución del software, como por ejemplos las actividades, los stakeholders o los archivos.
- La segunda capa define los conceptos pertenecientes al dominio de una forma menos abstracta que la capa superior, es decir que las descripciones deben estar en un nivel que permita realizar análisis y obtener resultados. Describe los conceptos que surgen de los distintos subdominios, como por ejemplo el control de versiones de archivos, el seguimiento de bugs y tareas, mediciones, etc.
- Las capas inferiores representan los elementos concretos del dominio. En estas capas se describen los lenguajes de programación concretos

como Java, C y C++, o los resultados obtenidos de realizar una medición. La extensión de esta capa permite describir nuevas tecnologías o sistemas involucrados en la evolución del software, que pueden ser representados de manera abstracta en las capas superiores.

Debido a que la información modelada en una ontología no está pensada para ser leída por una persona, se agregó una capa de lenguaje natural que es transversal a la pirámide de SEON. Ésta capa se compone de anotaciones, que son etiquetas legibles para un ser humano, nombrando cada una de las clases y propiedades descriptas.

Las ontologías de SEON se utilizan en el modelo de Singapur para describir los diversos elementos existentes en un proyecto de desarrollo de software.

5.3. FedX

En los últimos años, la Web evoluciona cada vez más de una red de documentos a una red de datos[16]. Éste desarrollo comenzó cuando los principios de Linked Data fueron formulados con la visión de crear un conjunto de datos globalmente interconectados. La conexión de datos semánticos relacionados es fundamental para los objetivos de la iniciativa de Linked Open Data, que es un proyecto que apunta a conectar los datos RDF distribuidos en la web. Actualmente, la nube de Linked Open Data contiene más de 200 juegos de datos interconectados.

Debido a esta iniciativa, surge la posibilidad de realizar consultas sobre múltiples juegos de datos distribuidos. Para poder unir la información distribuida y realizar consultas sobre ella, es necesario contar con estrategias de procesamiento de consultas eficientes. El primer enfoque para resolver este problema era integrar los distintos juegos de datos en un servidor local centralizado, pero debido a la naturaleza distribuida de la Web Semántica, este enfoque fue descartado migrando hacia la ejecución de consultas distribuidas sobre los distintos juegos de datos con el fin de realizar una integración virtual. Desde el punto de vista del usuario, ésto significa que los datos de distintos juegos de datos pueden ser consultados de manera transparente como si estuvieran en la misma base de datos.

FedX es un motor de procesamiento de consultas SPARQL federadas (sobre distintas BBDD semánticas distribuidas) optimizado mediante nuevas

técnicas de procesamiento y agrupación. Sus principales objetivos son proveer un soporte eficiente y completo para el estándar SPARL en ambientes distribuidos, flexibilidad en cuanto a la especificación de juegos de datos (que se puedan agregar y quitar dinámicamente), que se pueda implementar usando el mismo estándar SPARQL y minimizar el flujo de datos entre los distintos endpoints SPARQL debido a que éstos pueden tener que viajar por Internet.

Éste motor de procesamiento de consultas se puede utilizar en Singapur para trabajar en un entorno con datos semánticos distribuidos, integrándolos de manera virtual en un único repositorio.

5.4. TUKAN

Los ambientes de trabajo compartidos permiten a las personas mantener una gran cantidad de conocimiento sobre las interacciones realizadas por cada uno con el ambiente de trabajo. Éste conocimiento se denomina Workspace Awareness[15], y es una pieza fundamental para que los grupos de trabajo colaboren eficientemente[4]. Los sistemas de colaboración en tiempo real que proveen un entorno virtual compartido, pierden muchas posibilidades que se dan en los ambientes físicos compartidos.

En la actualidad, el desarrollo de software se lleva a cabo en equipos, y varias metodologías de desarrollo de software modernas hacen énfasis en esto introduciendo formas especiales de colaboración, como Pair Programming y Adaptive Software Development Process. De cualquier manera, la programación es implícitamente una actividad realizada por (varios) individuos, como escribir un libro o componer música (Weinberg, 1971). Por lo tanto, la discrepancia entre el trabajo aislado y grupal es inherente al desarrollo de software. Los ambientes que quieran dar soporte a los programadores en su tarea de programar, deben proveer diferentes modos de colaboración acorde a las necesidades del proceso de desarrollo de software y facilitar las transiciones entre ellas.

Los equipos de programación medianos o largos, suelen encontrarse distribuidos geográficamente. Ésto introduce nuevos desafíos a la organización del trabajo de programación. Generalmente se utilizan sistemas de control de versiones y videoconferencias, pero éstas herramientas no proveen Workspace Awareness del trabajo de los integrantes del equipo.

En este paper se proponen distintos modos de colaboración que brin-

dan Workspace Awareness en un entorno de trabajo colaborativo, cada uno de estos modos de colaboración cumple con las distintas necesidades que pueden surgir en las distintas etapas de la colaboración. En particular se propone tener awareness de los cambios que están siendo realizados por otras personas en el código, para evitar los cambios conflictivos del código y la resolución del mismo problema en forma simultánea por varios integrantes.

El objetivo de éste trabajo se relaciona con el objetivo de Singapur, debido a que es un caso particular de una buena práctica (saber cuando hay otras personas trabajando en la misma pieza de código), que nosotros tomamos de forma generalizada a cualquier buena práctica de cualquier tipo.

Capítulo 6

Enfoque Semántico a la Gestión de Mejores Prácticas

6.1. Singapur

Singapur es una herramienta que permite gestionar las mejores prácticas, para ello contiene un modelo semántico que permite definir buenas prácticas y un conjunto de violaciones[1] a las mismas para detectar cuando no se cumplen.

Para poder realizar esta tarea, las distintas herramientas que son utilizadas para llevar a cabo el desarrollo de software (agentes generadores de datos) envían una representación semántica de las acciones realizadas, o eventos que ocurren, a la aplicación Singapur. Dichas acciones pueden ser de distintos tipos como por ejemplo que un desarrollador realice un commit al repositorio de código, que un analista escriba un documento, o que un tester ejecute un caso de prueba. Las mejores prácticas no se limitan solo a las acciones directamente relacionadas con la programación, sino que pueden ser de cualquier tipo, el límite está impuesto por el modelo del que se dispone, que puede ser fácilmente extendido, y la capacidad de recolectar los datos necesarios de las acciones realizadas a medida que van ocurriendo, aunque no siempre es necesario detectar las violaciones en el momento en que ocurren.

[1] Se optó por definir las violaciones a las buenas prácticas (en contraposición a definir el estado esperado), debido a que de esta manera resulta más sencillo informar al usuario cómo resolver cada situación no deseada en particular, en vez de describir sólo la forma correcta de llevar a cabo una tarea e informar cuando hay un desvío.

A medida que se ingresan los datos, Singapur detecta si ocurrió alguna violación a las buenas prácticas mediante un agente explotador de los datos, las instancia en el modelo e informa a los usuarios involucrados para que puedan corregir la situación. También actualiza las violaciones que son corregidas para que se sepa que se revirtió la situación no deseada.

El modelo semántico se adapta perfectamente a este escenario debido a tres grandes motivos:

- El primero es que desacopla la definición de los conceptos (buenas practicas, violaciones, etc) de la aplicación.
- El segundo es que la integración de datos de distintas fuentes con destintas definiciones resulta más sencilla que con otros modelos debido a la posibilidad de declarar equivalencias entre distintas clases que representan el mismo concepto.
- El tercero es que permite aplicar un razonador a los datos para detectar automáticamente las violaciones a las buenas prácticas a partir de un conjunto de reglas.

Para integrar el conocimiento generado en Singapur (las violaciones que ocurrieron a las buenas prácticas y cómo corregirlas) con las herramientas de trabajo, se debe integrar un agente explotador de datos en las mismas. De ésta manera se puede brindar a los trabajadores una información muy valiosa y de una manera muy eficaz[2].

Desde una interfaz gráfica se pueden gestionar las buenas prácticas y las reglas para detectar violaciones a las mismas, y visualizar las violaciones a las buenas prácticas que surgieron, junto con su descripción y los datos necesarios para poder tomar las acciones correctivas necesarias.

6.2. Arquitectura

Singapur utiliza una arquitectura cliente-servidor donde el servidor es el encargado de almacenar el modelo y proveer una interfaz para realizar las operaciones necesarias. Los clientes pueden ser de dos tipos, generadores

[2] Al poder visualizar la información de la violación en el mismo contexto de trabajo y en el mismo momento en que ocurre, teniendo además la posibilidad de presentarla de la manera que se considere más conveniente, resulta muy difícil que el usuario no se entere de la ocurrencia de la violación a las mejores prácticas y cómo se debe resolver.

o consumidores de datos. Éstos componentes trabajan en conjunto para detectar las violaciones a las mejores prácticas (como se puede ver en la figura 6.2):

- **Singapur:** Es una aplicación Web donde se pueden gestionar las mejores prácticas y sus violaciones. Es muy importante que sea una aplicación Web para facilitar su utilización de manera remota, ya que en la actualidad muchos equipos de desarrollo se encuentran distribuídos geográficamente[12]. Contiene la base de datos semántica (Apache TDB Triplestore), y utiliza el framework Apache Jena para su integración y para realizar la detección de las violaciones con su motor de ejecución de reglas. Éste servidor expone una conjunto de servicios web que permiten a los clientes tanto consultar como insertar datos.

- **Agentes consumidores de datos:** Son los que leen los datos generados. Algunos ejemplos de agentes consumidores de datos pueden ser un cliente utilizado para mostrar las violaciones que surgieron, o cualquier otro agente que lea las violaciones que ocurrieron involucrando a una persona cualquiera para brindar información contextualizada en el entorno de trabajo.

- **Agentes generadores de datos:** Éstos agentes son los que generan los datos que se almacenan en el servidor. Algunos de ellos se integran en las herramientas utilizadas en el proyecto de desarrollo de software para obtener los datos de las acciones realizadas o eventos ocurridos. Se pueden integrar en un Servidor de Integración Contínua o en otras herramientas como se puede ver en la figura 6.2. También puede haber agentes que en vez de agregar datos a medida que se van generando, realicen importaciones de datos ya existentes desde nuevas fuentes de datos, o realicen inserciones con una determinada frecuencia. Al desacoplar los agentes generadores de datos, se tiene una gran flexibilidad sobre la manera de generar los datos. Un caso especial de agente generador de datos es el razonador, que genera nuevos datos a partir de la información generada previamente, convirtiéndolo también en un agente consumidor de datos.

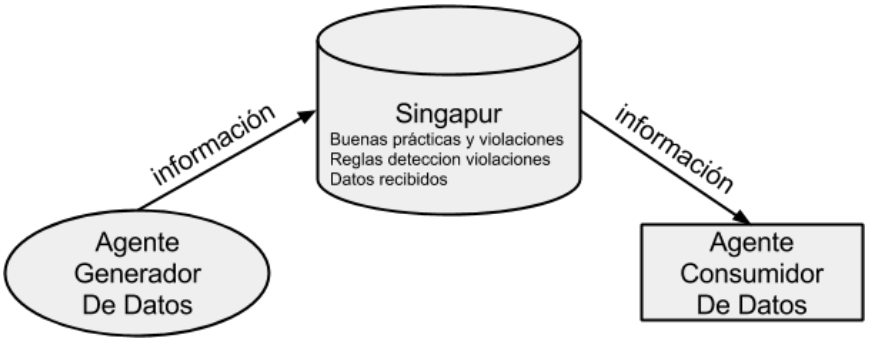


Figura 6.1: Arquitectura general de Singapur y sus componentes.

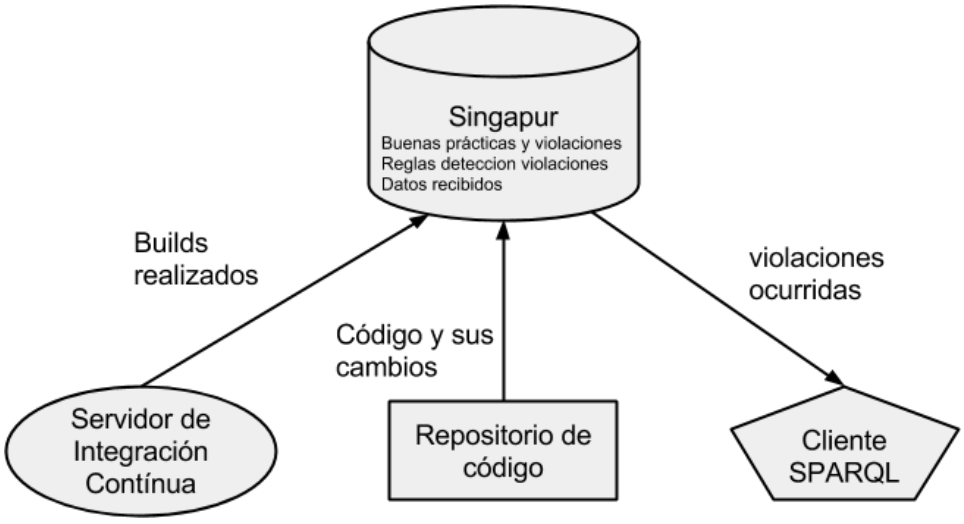


Figura 6.2: Arquitectura general de Singapur con algunos ejemplos de agentes generadores y consumidores de datos.

6.2.1. Arquitectura interna del servidor de Singapur

La implementación del servidor de Singapur se realizó con Spring MVC, que permite desarrollar una aplicación Web[3] desacoplando sus componentes internos, lo que facilita luego la posibilidad de pedir información en distintos formatos. Ésto es particularmente útil para la negociación de contenidos en las consultas SPARQL que se realizan a la base de datos semántica, es decir, que fácilmente se puede obtener los resultados de las consultas en los distintos formatos deseados (RDF/XML, turtle, html, etc).

Para conectarse con el servidor web de Singapur, los agentes consumidores o generadores de datos sólo necesitan un cliente http para realizar las peticiones web.

El framework Apache Jena se encarga de realizar las tareas necesarias para acceder y modificar el grafo semántico. Permite realizar consultas, inserciones y también se encarga del almacenamiento y persistencia en la base de datos semántica Apache TDB Triplestore. El framework incluye distintos tipos de razonadores (y la posibilidad de extenderlos) que permiten ejecutar reglas, en este caso utilizamos un razonador de propósito general que será descripto en detalle en el capítulo 8.

[3]Se denomina aplicación web a aquellas herramientas que los usuarios pueden utilizar accediendo a un servidor web a través de Internet o de una intranet mediante un navegador. Algunas de sus ventajas principales son la practicidad del navegador web como cliente ligero, la independencia del sistema operativo y la facilidad para actualizar y mantener aplicaciones web sin distribuir e instalar software a miles de usuarios potenciales.

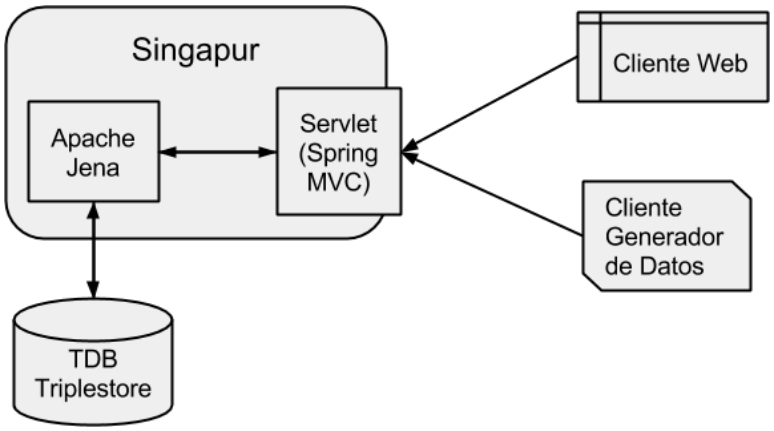


Figura 6.3: Arquitectura interna de los componentes de Singapur y sus relaciones entre ellos y con los componentes externos.

Capítulo 7

Modelo Semántico de Mejores Prácticas

Singapur cuenta con una base de datos semántica (Apache Jena y TDB) en el que se persiste el modelo semántico de las mejores prácticas y los elementos del dominio de un proyecto de desarrollo de software, y por lo tanto posee un conjunto de características muy importantes:

- Las relaciones de las instancias de violaciones (o de cualquier otro tipo) con otras instancias o datos, se pueden agregar, quitar o modificar de forma sencilla e inmediata.
- En lugar de tener que desarrollar la lógica necesaria para detectar las violaciones, se puede utilizar la inferencia lógica (mediante reglas) sobre la información almacenada en la base de datos semántica.
- La integración de los datos de acciones realizadas o eventos ocurridos, que vienen de distintas fuentes, se simplifica al poder declarar equivalencias entre distintas clases que representan el mismo concepto. Lo que permite declarar una regla que se va a aplicar sobre clases desconocidas en el momento.

Singapur cuenta con su propia ontología donde define las buenas prácticas y las violaciones a las mismas, que se ilustra en la figura 7.1.3.

Las clases que se describen en la ontología de Singapur son:

- **Mejores prácticas:** Tienen un título, una descripción, requerimientos, beneficios, desventajas, estado, autores, referentes, comentarios y el conjunto de clases de violaciones que las afectan.

- **Violaciones:** Poseen un título, una descripción, la fecha de detección, un conjunto de personas involucradas en la generación de la violación y un conjunto de recursos asociados a la misma. Dentro de la ontología no se describen ni las personas ni los recursos, lo que se describe son las relaciones con ellos, debido a que lo importante es la relación en sí y no la descripción de personas o recursos.

  Hay dos subclases de violaciones. La subclase ViolacionInevitable, representa las que su aparición no necesariamente significa que no se respetó una buena práctica (en este trabajo la utilizamos para los casos en que la aparición de una violación depende de si se hace un paso siguiente o no, debido a que los motores de razonamiento actuales no pueden realizar razonamientos en base al tiempo transcurrido desde un evento) y la subclase ViolacionResoluble que son las que pueden ser corregidas mediante un conjunto de acciones y poseen una fecha de resolución.

Este modelo puede ser extendido para agregar tipos de violaciones específicos como se muestra en la figura 7.1.3.

7.1. Definición de mejores prácticas

Para definir una práctica primero hay que describirla, podemos continuar con los ejemplos de la sección 2.3:

7.1.1. Integración exitosa del código

Título: Integración exitosa del código fuente.

Descripción: Para que cada commit del código se encuentre integrado exitosamente en el servidor de integración contínua, debe tener un build en estado exitoso o debe haber un commit posterior que lo tenga.

Requerimientos: Se puede aplicar en el contexto de un proyecto de desarrollo de software en que se utiliza una herramienta de control de versiones del código fuentes y un servidor de integración contínua.

Beneficios: Ésta práctica es para asegurar que luego de realizar una modificación al código fuente de la aplicación que se está desarrollando, se realizan todas las comprobaciones al código configuradas en el servidor de

integración contínua[1] y que si alguna falla, es arreglada en el momento o se deshacen los cambios realizados, de manera que siempre haya una versión estable de la aplicación[13]. Además, permite que se encuentren posibles problemas antes de proseguir con otra tarea.

Desventajas: Puede llegar a consumir más tiempo integrar cada uno de los commits individualmente que de a varios.

Otros datos: Adicionalmente se deben completar los campos autores, referentes, comentarios y estado.

Violaciones: Para poder crear una instancia de BuenaPráctica en la base de datos falta definir el conjunto de clases de violaciones que la afectan. En este caso para que no se cumpla la buena práctica debe darse alguna de las dos situaciones siguientes:

- Un commit no está cubierto por ningún build en el servidor de integración contínua, es decir, que no se ejecutó ningún build con ese commit o cualquiera que le siga.
- Un commit (y todos los commits posteriores) fallaron la ejecución del build en el servidor de integración contínua.

Las dos situaciones anteriores son subclases de violaciones que nombraremos NoHayUnBuild y NoHayUnBuildExitoso respectivamente, como se puede ver en la figura 7.1.3. La primer subclase de violación, NoHayUnBuild, extiende de ViolacionInevitable y de ViolacionResoluble debido a que siempre hay un periodo de tiempo en que se realizó el commit y todavía no se realizó el build correspondiente, y se puede resolver realizando un build. La segunda subclase de violación, NoHayUnBuildExitoso, extiende ViolacionResoluble debido a que mayormente se puede evitar realizando previamente las verificaciones necesarias. En el capítulo 8 detallaremos la declaración de las reglas que permiten detectar las violaciones y crear nuevas instancias de las subclases de violaciones mencionadas previamente.

7.1.2. Evitar conflictos en las modificaciones al código fuente

Título: Evitar conflictos en las modificaciones al código fuente.

[1] Generalmente éstas comprobaciones incluyen la detección de errores de compilación y la ejecución de tests de unidad, integración o de otro tipo.

Descripción: Al haber más de un desarrollador modificando un archivo al mismo tiempo, los mismos deben saberlo para que no surjan conflictos al momento de integrar los distintos cambios.

Requerimientos: Se puede aplicar en el contexto de un proyecto de desarrollo de software que cuenta por lo menos con 2 desarrolladores trabajando en el mismo módulo.

Beneficios: Evita que surjan conflictos debido a la falta de conocimiento del trabajo de los demás al momento de integrar los distintos cambios que realizaron distintos desarrolladores al mismo tiempo.

Desventajas: Ninguna.

Otros datos: Adicionalmente se deben completar los campos autores, referentes, comentarios y estado.

Violaciones: Para que no se cumpla la buena práctica debe darse la situación en que hay dos o más desarrolladores modificando un mismo archivo de manera local (sin haber realizado un commit del mismo). A ésta subclase de violación la nombraremos ConflictoEnEdiciónDeArchivo, como se puede ver en la figura 7.1.3.

7.1.3. Cobertura de tests completa

Título: Cobertura de tests completa.

Descripción: Cada línea de código debe estar probada en un test, es decir, que debe haber una cobertura de tests del 100 % del código en cada revisión. Es importante que los tests se incorporen en la misma revisión que los cambios para que cualquier versión del software que se quiera utilizar tenga asegurado el nivel de calidad.

Requerimientos: Se puede aplicar en el contexto de un proyecto de desarrollo de software que cuenta con control de versiones del código fuente, tests automatizados sobre el código fuente y una herramienta de medición de la cobertura de código por parte de los tests.

Beneficios: Fomenta la cobertura total de tests, por lo que se tiene una mayor calidad en el software debido a que es más probable que se detecten

errores o comportamientos no deseados del sistema.

Desventajas: Se necesita disponer de recursos (desarrolladores y tiempo) para desarrollar y mantener actualizados los tests.

Otros datos: Adicionalmente se deben completar los campos autores, referentes, comentarios y estado.

Violaciones: Para que no se cumpla la buena práctica debe darse alguna de las siguientes situaciones:

- No hay una medición realizada sobre la cobertura de los tests de una clase para una revisión particular. A ésta subclase de violación la nombraremos NoHayMedicionDeCoberturaDeTests, como se puede ver en la figura 7.1.3. Extiende la clase ViolacionInevitable debido a que siempre hay un lapso de tiempo entre que se realiza un commit y se ejecuta la herramienta que valida la cobertura de la nueva revisión, y extiende ViolacionResoluble debido a que se resuelve una vez que se agrega la medición de la cobertura.
- Una clase tiene una cobertura menor al 100 %. A ésta subclase de violación la nombraremos NoHaySuficienteCoberturaDeTests, como se puede ver en la figura 7.1.3. Extiende directamente de la clase Violación debido a que se puede evitar revisando la cobertura antes de realizar un commit, y no se puede resolver porque esa revisión en particular ya no se puede cambiar y queda con una cobertura menor a la deseada.

7.2. Modelo del proyecto de desarrollo de software

Para poder detectar las violaciones mencionadas anteriormente, se necesita disponer de un modelo que describa los diversos elementos existentes en un proyecto de desarrollo de software. Para esto utilizaremos la ontología SEON (Software Evolution ONtologies), que fue creada con el objetivo de facilitar la implementación de herramientas que ayuden a los ingenieros de software en la gestión del software en todo su ciclo de vida, y describe la mayoría de los conceptos importantes de este dominio. En la figura 7.2 se ilustran los conceptos descriptos en la ontología SEON.

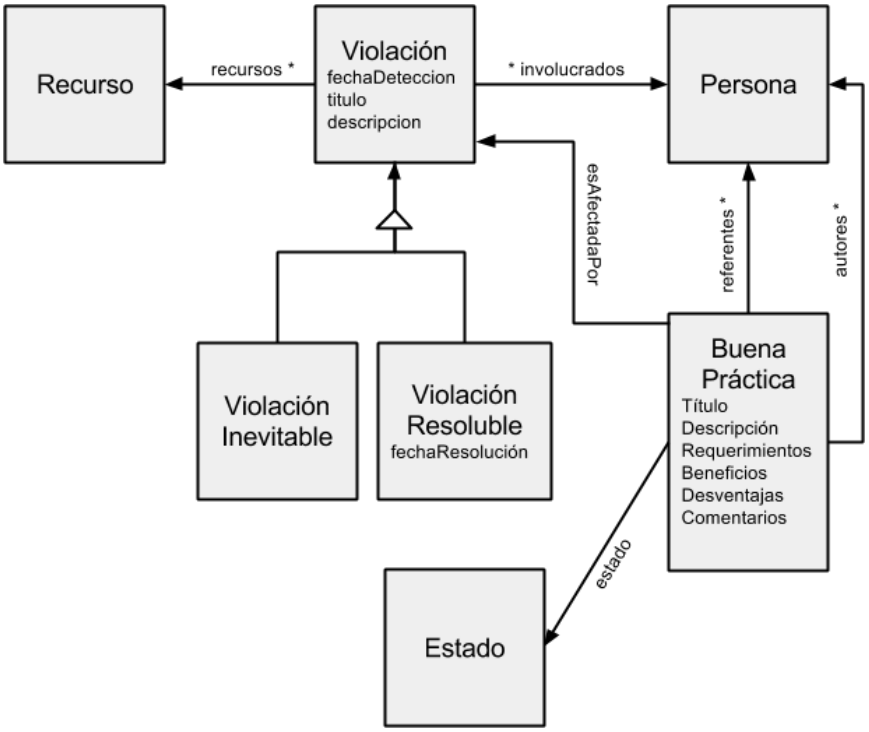


Figura 7.1: Representación del modelo provisto en la ontología de Singapur de buenas prácticas y violaciones.

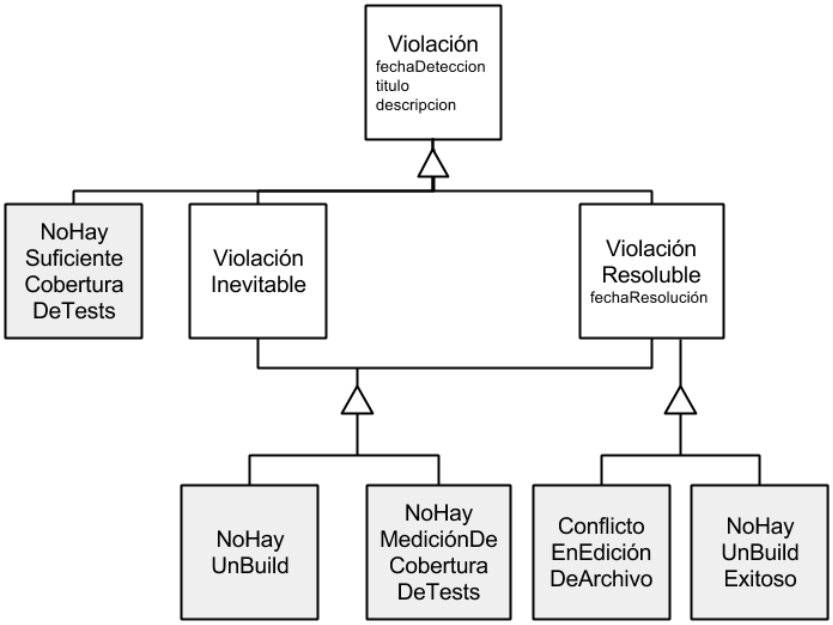


Figura 7.2: Ejemplo de extensión del modelo de violaciones.

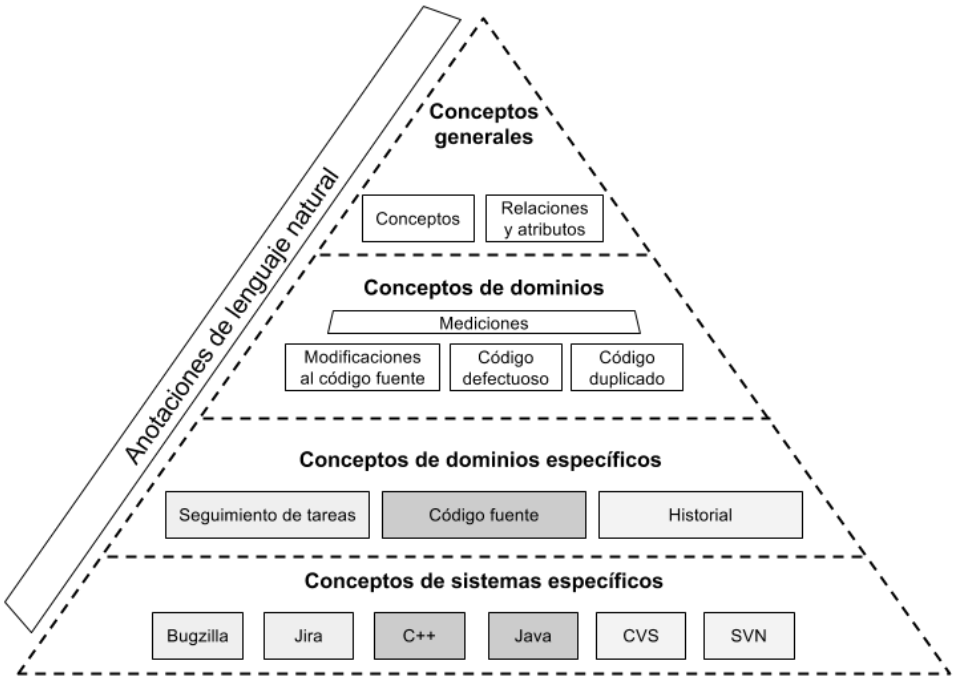


Figura 7.3: Representación de las capas de conceptos que se modelan en la ontología SEON (Software Evolution ONtologies).

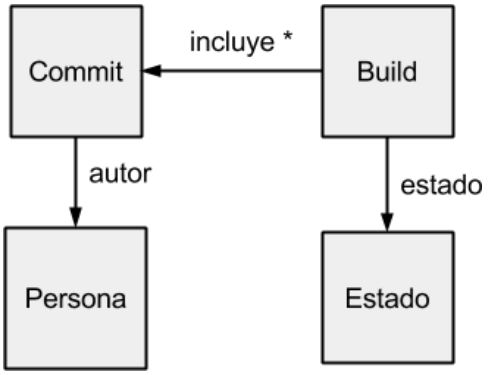


Figura 7.4: Modelo para describir la integración del código fuente dentro de un proyecto de desarrollo de software.

7.2.1. Modelo para la integración del código fuente

Para poder describir la integración del código fuente, y por lo tanto detectar si no se realizó correctamente, es necesario contar con la información de los commits realizados por lo desarrolladores y con las integraciones realizadas en el servidor de integración contínua. En la figura 7.2.1 se describe un modelo que cuenta con los commits realizados (cada uno con su respectivo autor), y los builds con los commits que incluye y su estado (que puede ser Exitoso o Fallido). Fue necesario extender la ontología SEON, ya que la misma no describe los builds realizados.

Las situaciones que se deben detectar son las siguientes:

- Un commit no está incluído en ningún build en el servidor de integración contínua, es decir, que no se ejecutó ningún build con ese commit o cualquiera que le siga. Ésta situación se ejemplifica en el Commit 4 de la figura 7.2.1, que es el último commit realizado y no se encuentra incluído en ningún build.

- Un commit está incluído en un build con estado fallido en el servidor de integración contínua, y no tiene commits siguientes, o los siguientes commits se encuentran incluídos en builds fallidos o no se encuentran incluídos en ningún build. Ésta situación se ejemplifica en el Commit 3 de la figura 7.2.1, que está incluído en el Build 2 que tiene estado fallido, y el commit siguiente (que es el Commit 4) no está incluído en ningún build.

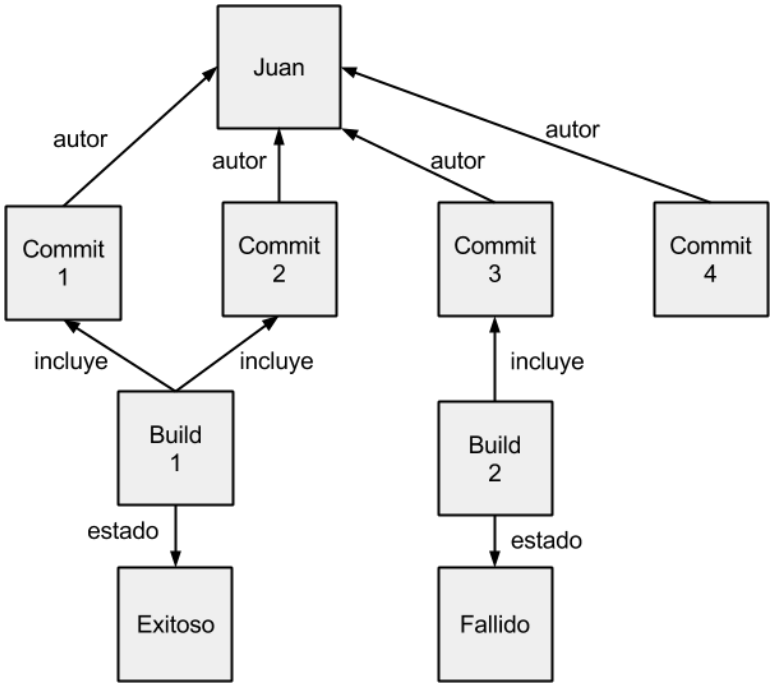


Figura 7.5: En este ejemplo se puede ver un caso de cada una de las situaciones no deseadas que pueden surgir en la integración del código fuente. El Commit 3 está incluído en el Build 2 que tiene estado fallido, y el commit siguiente (que es el Commit 4) no está incluído en ningún build.

Una vez que se detectan las violaciones ocurridas, se instancian (con la subclase de violación que se muestra en la figura 7.1.3 correspondiente a cada situación) haciendo referencia al commit que no se integró exitosamente y a la persona involucrada en la violación, que en este caso es el autor del commit mencionado previamente. En la figura 7.2.1 se pueden ver las violaciones que se instanciarían en el ejemplo mostrado en la figura 7.2.1.

7.2.2. Modelo de modificaciones al código fuente

Para detectar cuando hay más de un desarrollador modificando la misma porción de código, es necesario conocer qué modificaciones están realizando los desarrolladores al código. En la figura 7.2.2 se muestra el modelo que describe el código, los desarrolladores y las modificaciones que están realizando en el momento. Una entidad de código puede ser un método, una clase, una

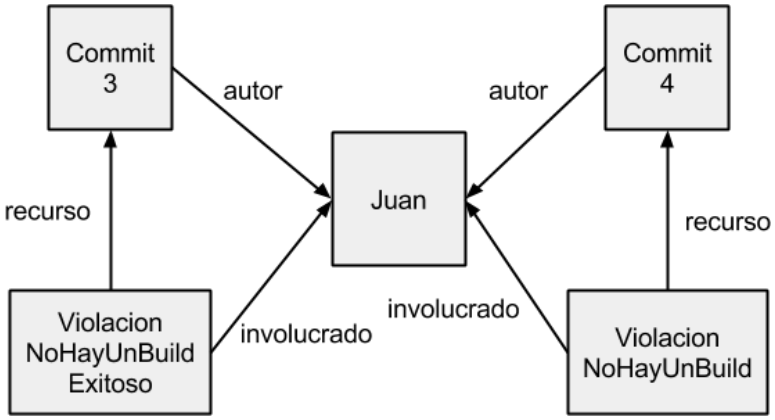


Figura 7.6: Instancias de violación que se generarían al detectar la situación no deseada del ejemplo mostrado en la figura 7.2.1.

Figura 7.7: La entidad de código se organiza recursivamente, y puede estar siendo modificada por una o varias personas.

estructura de datos, una librería, una rutina, una fórmula, o cualquier entidad que se utilice en cualquier paradigma de programación. La ontología SEON describe éste modelo.

La situación que se busca detectar es que haya dos o más desarrolladores modificando una misma porción de código en el mismo momento. En la figura 7.2.2 se ilustra esta situación, debido a que Juan y Carlos están modificando ClaseDos al mismo tiempo, y en la figura 7.2.2 se puede ver la violación que se debe generar al detectar éste posible conflicto en las modificaciones realizadas a ClaseDos.

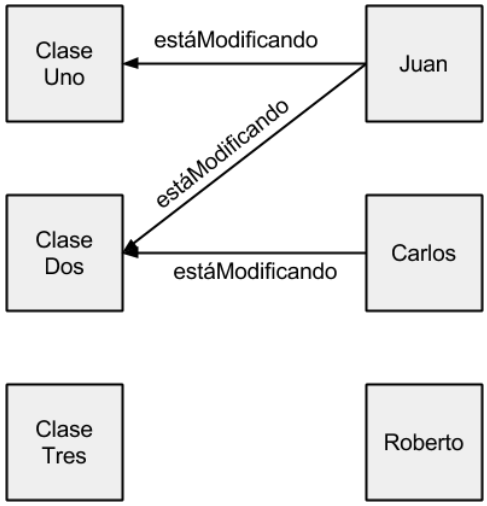


Figura 7.8: En este ejemplo se puede ver la situación no deseada en que los desarrolladores Juan y Carlos están modificando ClaseDos al mismo tiempo.

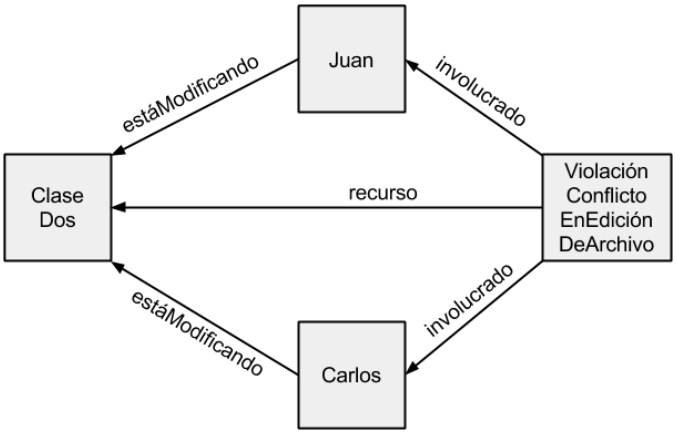


Figura 7.9: Violación que se instanciaría al detectar el posible conflicto en la edición de ClaseDos (involucrando a Juan y a Carlos) del ejemplo visto en la figura 7.2.2.

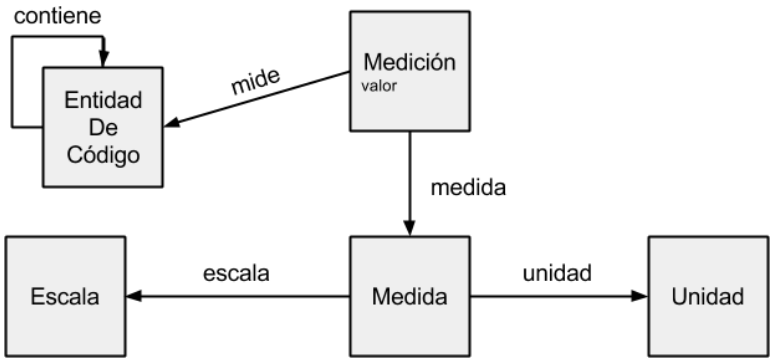


Figura 7.10: Clases utilizadas para representar el código y las mediciones.

7.2.3. Modelo de cobertura de tests

Para poder detectar que haya una cobertura completa de tests automatizados sobre el código fuente, es decir, que el 100 % de las líneas de código se encuentren verificadas en algún test, el modelo debe incluir información del código y su cobertura de tests. La ontología SEON modela el código y los conceptos necesarios para realizar mediciones de distintos tipos, por lo tanto se necesitó instanciar la unidad Porcentaje (y su escala del 0 al 100), y el tipo de medida CoberturaDeTests. En la figura 7.2.3 se muestran las clases de la ontología SEON que se utilizan.

Las situaciones que se quiere detectar son:

- Cuando no hay una medición de la cobertura de tests de una entidad de código.
- Cuando el valor de la medición es menor al 100 %.

En la figura 7.2.3 se puede observar las instancias de medida, escala y unidad necesarias para realizar una medición de la cobertura de tests de una clase, y los ejemplos de las situaciones no deseadas, que son ClaseUno y ClaseDos respectivamente. En la figura 7.2.3 se muestran las violaciones que se instanciarían al detectar éstas situaciones.

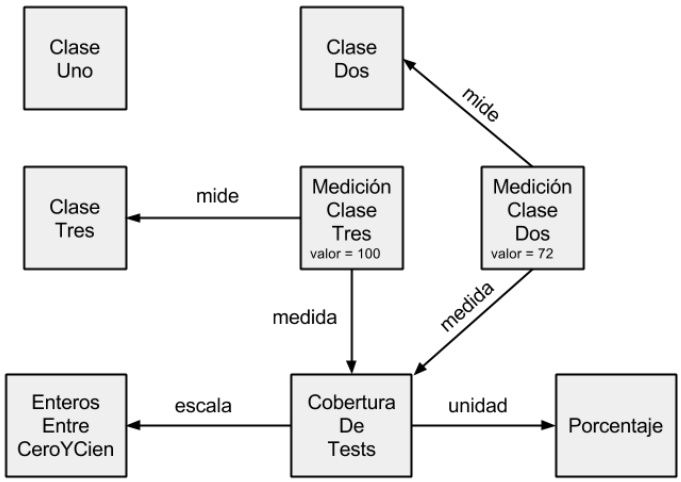


Figura 7.11: CoberturaDeTests es una medida que tiene de unidad a Porcentaje y de escala a EnterosEntreCeroYCien. ClaseUno no tiene ninguna medición de CoberturaDeTests realizada, ClaseDos tiene una medición con valor de 72 % y ClaseTres tiene una medición con valor de 100 %.

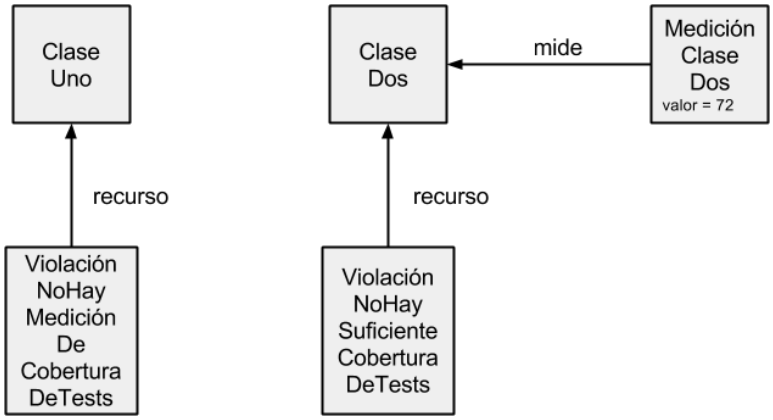


Figura 7.12: En esta figura se ven las instancias de violaciones que corresponden a ClaseUno, que no tiene ninguna medición de CoberturaDeTests realizada, y a ClaseDos, que tiene una medición con valor de 72 %.

7.3. Modelo del estado de una buena práctica

Las buenas prácticas, durante su gestión, van pasando por distintos estados dentro de su ciclo de vida. Es importante que este ciclo de vida se encuentre modelado[7], para que la aplicación maneje éste concepto, es decir, que sepa si una buena práctica está madura, es experimental, o se encuentra en evaluación para eventualmente realizar distintas acciones si es necesario, o simplemente, para poder informárselo al usuario.

Éste ciclo de vida puede modelarse (y modificarse dinámicamente) con la Ontología Para Máquinas de Estado de Peter Dolog[2]. En la figura 7.3 se puede ver una representación gráfica del ciclo de vida de las buenas prácticas que proponemos (aunque puede utilizarse cualquiera que se desee):

- El primer estado del ciclo es "Propuesta", representa una versión que se encuentra en desarrollo y/o investigación.
- Una vez que se tiene una primera versión de la buena práctica que se quiere probar se pasa al estado "En Evaluación", donde se aplica por primera vez con el propósito de verificar que se pueda utilizar y que se obtengan los resultados esperados.
- Si no se obtienen los resultados esperados, y se determina que no hay una corrección posible, se pasa al estado "Descartada", finalizando su ciclo de vida.
- En caso de que se pueda corregir, se realizan las modificaciones necesarias y se vuelve a probar.
- Cuando se considera que se tiene una versión relativamente estable, se pasa al estado "Estable" y se siguen realizando las correcciones que se van encontrando al probar con los distintos escenarios que puedan surgir.
- Luego de que se aplicó muchas veces, teniendo una gran experiencia en su uso, y no se encuentran escenarios en que se obtengan resultados distintos a los esperados, se pasa al estado "Madura".
- Eventualmente, una buena práctica puede quedar obsoleta debido a la aparición de otra buena práctica o a cambios tecnológicos, pasando al estado "Obsoleta".

[2] http://people.cs.aau.dk/ dolog/fsm/

7.4. Generación de datos

Para la obtención de datos se utiliza el enfoque del escritorio semántico, es decir, que las aplicaciones que se utilizan para realizar el trabajo (IDE, procesador de texto, herramientas de integración de código, etc) generan metadatos con la información que manejan y los envían[20] al servidor de Singapur. Para lograr esto se deben realizar pequeñas extensiones a las aplicaciones mediante plugins u otras estrategias de customización, por lo tanto, es importante elegir herramientas que posean dicha capacidad o que ya dispongan de mecanismos para notificar de alguna manera de los eventos ocurrido o acciones realizadas.

7.4.1. Generación de datos de Integración del código fuente

En el caso de la Integración exitosa del código fuente, es necesario que el repositorio de código envíe al servidor de Singapur la información de los commits realizados y que el servidor de integración contínua envíe la información de los builds realizados. Además, debe haber una manera de relacionar los commits que genera el repositorio de código con los que percibe el servidor de integración contínua, para eso se puede utilizar el número de commit que es un identificador natural y único para cada commit. La misma situación surge con la identificación de las personas que realizan los commits y los builds, éste caso es más complejo debido a que depende de cada herramienta la identificación de sus usuarios, es decir, que no existe un identificador natural y único[3], por lo tanto su resolución depende de cada situación particular y se encuentra limitada por las herramientas usadas en cada caso.

7.4.2. Generación de datos de modificaciones no integradas al código fuente

Para poder detectar cuando hay más de una persona modificando el mismo archivo o sección de código, es necesario contar con la información del código fuente. En este caso hay dos opciones, se puede tomar a nivel de archivos o a nivel de entidades de código, que pueden ser métodos,

[3]Un identificador natural de las personas puede ser su nombre y apellido, pero presenta el problema de que no es un identificador único. Otro identificador natural dentro de un mismo sistema puede ser el nombre de usuario, pero éste puede variar entre distintos sistemas, es decir, que no se puede asegurar que un mismo nombre de usuario represente a la misma persona en distintos sistemas.

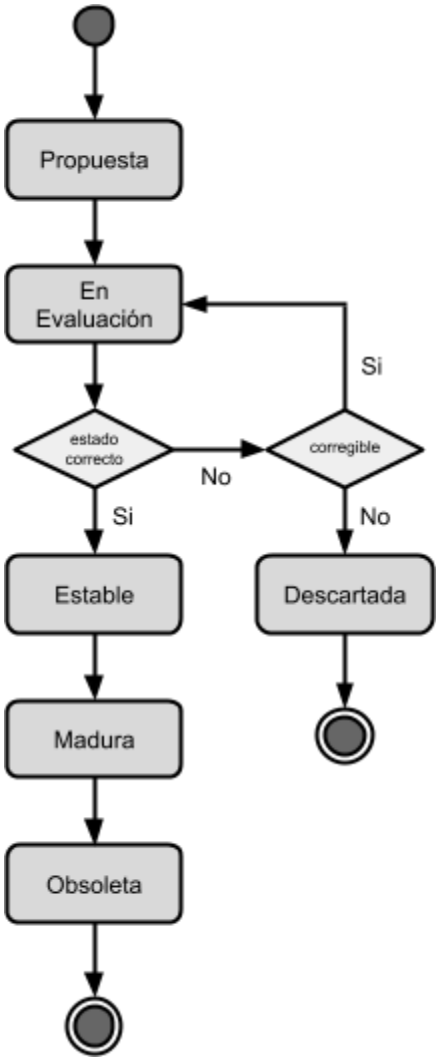


Figura 7.13: Ciclo de vida propuesto para las buenas prácticas.

clases, funciones, procedimientos o cualquier otro tipo de entidad de código utilizada. En este caso puede haber una gran cantidad de enfoques para la generación de la información, puede ser una extensión al repositorio del código fuente o una herramienta aparte que toma la información del repositorio y genera los datos semánticos para mandarlos al servidor de Singapur. Por otro lado, es necesario saber cuando una persona está realizando modificaciones al código (sin haber subido los cambios al repositorio), para ésto también se pueden tomar varios enfoques, por ejemplo puede ser mediante una extensión al IDE utilizado para el desarrollo o un programa aparte que utilice el repositorio para saber cuáles archivos tienen cambios sin enviar al repositorio. Sea cual sea el enfoque, también está el problema de la identificación de las personas realizando las modificaciones.

7.4.3. Generación de datos de cobertura de tests

Se puede extender la herramienta que mide los porcentajes de cobertura de cada clase para que genere los metadatos de las mediciones realizadas. Es necesario que ésta herramienta sea capaz de identificar las clases de manera unívoca y de la misma forma que lo haga la herramienta o extensión que genere la información de las clases. Una clave natural y única para las clases suele ser una ruta o paquete seguidi por el nombre de la clase, de esta manera se puede relacionar una clase con una medición en particular que tiene un porcentaje de cobertura asociado.

7.5. Agente generador de datos de Integración del código fuente

Se puede implementar un agente generador de datos de integración del código fuente aprovechando las facilidades que brinda el servidor de integración contínua Jenkins. Por un lado provee una API web para consultar los datos de los builds en distintos formatos. A continuación se muestra un ejemplo de la respuesta a una petición de información de un build en particular (cuyo identificador es parte de la url):

```
{
  "result" : "SUCCESS",
  "timestamp" : 1426876390000,
  "changeSet" : {
```

```
    "items" : [
      {
        "date" : "2015-03-20T18:27:42.971390Z",
        "revision" : 13761,
        "user" : "benitezl"
      },
      {
        "date" : "2015-03-20T18:26:58.654970Z",
        "revision" : 13760,
        "user" : "benitezl"
      }
    ]
  }
}
```

El atributo result se corresponde con el estado del build al finalizar, timestamp es la fecha en que finalizó su ejecución (es la cantidad de milisegundos transcurridos desde el 1 de enero de 1970) y changeSet representa al conjunto de commits nuevos que incluye éste build. De cada build se tiene la fecha en que se realizó, la versión que representa y el usuario que lo hizo.

Ésta respuesta puede ser convertida a tripletas en algún lenguaje de serialización de RDF y enviadas al servidor de Singapur para que las agregue en su base de datos semántica y detecte la ocurrencia de violaciones. Para realizar esto se puede utilizar el siguiente script en bash:

```
#!/bin/bash

# Se guarda en la variable BUILD el numero de build pasado como
    parametro
BUILD=$1

# URL del servidor de integracion continua
JENKINS_URL=http://jenkins

# URL del servidor de Singapur
SINGAPUR_POST_URL=http://singapur/post

# Con este comando se pide a Jenkins la informacion necesaria de un
    build en particular (pasado como parmetro) y se guarda en el
```

```
    archivo build_info.
wget -q -O build_info "$JENKINS_URL/job/$PROYECTO/$BUILD/api/json?
    pretty=true&tree=result,timestamp,changeSet[items[revision,user,date]]"

# Se utiliza el comando jq para interpretar y extraer la informacion
    obtenida del build
BUILD_RESULT=`cat build_info | jq '.result'`
BUILD_TIMESTAMP=`cat build_info | jq '.timestamp'`
REVISIONS_COUNT=`cat build_info | jq '.changeSet.items | length'`
BUILD_DATE=`date -d @${BUILD_TIMESTAMP:0:10} +"%Y-%m-%dT%
    H:%M:%S"`

if [ -z "${REVISIONS_COUNT}" ]; then
    REVISIONS_COUNT=0
fi

INCLUDES=""

for (( i=0; i<$REVISIONS_COUNT; i++ ))
do
        REVISION=`cat build_info | jq .changeSet.items[$i].revision`
        AUTOR=`cat build_info | jq .changeSet.items[$i].user`
        COMMIT_DATE=`cat build_info | jq .changeSet.items[$i].date`

        #Genera la version
        VERSION_ID=`uuidgen`
        echo "<http://singapur/version/$VERSION_ID> a seonh:Version ; seon:
            hasIdentifier $REVISION ." >> $OUTPUT_FILE

        #Genera el commit
        COMMIT_ID=`uuidgen`
        echo "<http://singapur/commit/$COMMIT_ID> a seonh:Commit ;
            seonh:constitutesVersion <http://singapur/version/$VERSION_ID>
             ; seon:isCarriedOutBy $AUTOR^^xsd:string ; sgr:hasTimestamp \"
            ${COMMIT_DATE:1:19}\"^^xsd:dateTime . " >> $OUTPUT_FILE

        INCLUDES=" $INCLUDES; sgr:includes <http://singapur/commit/
            $COMMIT_ID> "
done

#Genera el build
case $BUILD_RESULT in
   "\"SUCCESS\"") BUILD_RESULT_TTL="; sgr:hasStatus sgrs:Successful";;
   "\"FAILURE\"") BUILD_RESULT_TTL="; sgr:hasStatus sgrs:Failure";;
esac
```

```
echo "<http://singapur/build/$PROYECTO/$c> a sgr:Build
    $BUILD_RESULT_TTL $INCLUDES ; sgr:hasTimestamp \"$BUILD_DATE
    \"^^xsd:dateTime ." >> $OUTPUT_FILE

curl --form upload=@$OUTPUT_FILE --form press=OK
    $SINGAPUR_POST_URL
```

A continuación se muestran las tripletas que se terminarían generando con el script y almacenando en Singapur para el caso anterior:

```
<http://singapur/version/3d269b4c-4eab-4e71-a4de-6cc9892f3ba7> a seonh:
    Version ; seon:hasIdentifier 13761 .

<http://singapur/commit/dc4098f1-8ea2-4105-87ad-153966898a40> a seonh:
    Commit ; seonh:constitutesVersion <http://singapur/version/3d269b4c-4
    eab-4e71-a4de-6cc9892f3ba7> ; seon:isCarriedOutBy "benitezl"^^xsd:
    string ; sgr:hasTimestamp "2015-03-20T18:27:42"^^xsd:dateTime .

<http://singapur/version/1dc6ebd1-c44f-4ef3-9538-595e1f52d08d> a seonh:
    Version ; seon:hasIdentifier 13760 .

<http://singapur/commit/78bfd6cd-912b-4226-a04c-027b3e00531e> a seonh:
    Commit ; seonh:constitutesVersion <http://singapur/version/1dc6ebd1-c44f
    -4ef3-9538-595e1f52d08d> ; seon:isCarriedOutBy "benitezl"^^xsd:string ;
    sgr:hasTimestamp "2015-03-20T18:26:58"^^xsd:dateTime .

<http://singapur/build/mobile-red-social/475> a sgr:Build ; sgr:hasStatus sgrs
    :Successful ; sgr:includes <http://singapur/commit/dc4098f1-8ea2
    -4105-87ad-153966898a40> ; sgr:includes <http://singapur/commit/78
    bfd6cd-912b-4226-a04c-027b3e00531e> ; sgr:hasTimestamp
    "2015-03-20T15:33:10"^^xsd:dateTime .
```

Finalmente, se puede utilizar el plugin de Jenkins llamado Post Build Task para que luego de realizar un build invoque al script anterior con el número del nuevo build realizado. De ésta manera, cada vez que se termina de ejecutar un build, se agrega automáticamente en Singapur su descripción con datos semánticos.

Capítulo 8

Detección y Reporte de Violaciones a Mejores Prácticas

Para detectar la ocurrencia de violaciones a las mejores prácticas se utiliza un razonador sobre la base de datos semántica debido a que posee las siguientes ventajas:

- No es necesario realizar ninguna implementación para el mecanismo genérico de detección de las violaciones.
- Las reglas no quedan dependiendo de una implementación en particular sino de un lenguaje.
- Los distintos requerimientos que puedan surgir, tanto funcionales como no funcionales, se pueden resolver con cambios del motor de ejecución de las reglas. Por ejemplo los tiempos de ejecución o la posibilidad de trabajar con varias fuentes de datos en vez de tener un único repositorio centralizado.
- Las reglas se escriben sobre datos semánticos, lo que brinda todas las ventajas mencionadas anteriormente sobre los modelos semánticos.

Jena[1] incluye un razonador de propósito general basado en reglas que implementa la inferencia sobre el grafo RDF que se almacena en el triplestore. Para poder detectar la ocurrencia de violaciones a las mejores prácticas y crear las instancias de violaciones en el modelo, es necesario agregar un

[1] Framework que permite escribir, extraer y procesar grafos RDF.

conjunto de reglas en el razonador de Jena que utiliza Singapur. A medida que se van generando los datos semánticos sobre las acciones realizadas, el motor de ejecución de reglas realiza el razonamiento que permite detectar e instanciar las violaciones ocurridas con los datos mostrados en el capítulo 7. Adicionalmente, se debe detectar cuando se corrige la situación para eliminar la instancia de violación generada o agregarle una fecha de corrección.

Singapur provee una interfaz Web para gestionar las reglas correspondientes a cada práctica. Dichas reglas deben tener en cuenta la detección de la ocurrencia de una violación, la creación de la instancia de la violación y la detección de que se corrigió la violación y su correspondiente eliminación de la base de datos semántica.

8.1. Motor de inferencia y sus reglas

Para ejecutar las reglas Jena posee dos motores de inferencia, uno de Encadenamiento Hacia Adelante[2] con el algoritmo RETE[3], y otro de Encadenamiento Hacia Atrás[4]. Para nuestra aplicación utilizaremos ambos motores de inferencia en un esquema híbrido que utiliza el motor de Encadenamiento Hacia Adelante para preparar los datos, y el de Encadenamiento Hacia Atrás para responder las consultas (los datos que cumplen los objetivos).

Una regla consta de una lista de términos (premisas), una lista de términos de encabezado (conclusiones), un nombre opcional y una dirección de encadenamiento opcional. En la figura 8.1.1 se describe la sintaxis de las reglas de Jena.

8.1.1. Funciones primitivas

Las primitivas que pueden ser invocadas desde las reglas de inferencia se implementan en Java como un objeto que se almacena en un registro. Se pueden crear nuevas primitivas y registrarlas.

[2]Es uno de los métodos principales de razonamiento, consiste en usar reglas de inferencia para obtener más información de los datos disponibles hasta que se alcance un objetivo[18]

[3]Es un algoritmo de reconocimiento de patrones utilizado para implementar sistemas de reglas productivos[18], debido a que sacrifica el uso de memoria para obtener una mejor performance.

[4]Es otro de los métodos principales de razonamiento, y consiste en ir hacia atrás desde los objetivos[18].

Cada primitiva puede ser utilizada en el cuerpo de la regla, en el encabezado de la regla, o en ambos. Si se utiliza en el cuerpo de la regla actúa como una validación, y si se utiliza en el encabezado, actúa como una acción.

En la tabla 8.1.1 se describen algunas de las primitivas disponibles en Jena.

Nueva función primitiva para generar el identificador de una nueva instancia

La inferencia se utiliza mayormente para instanciar nuevas relaciones entre entidades ya existentes, para poder utilizarla para instanciar nuevas entidades (nuevas instancias de violaciones) fue necesario definir una nueva función primitiva (builtin) denominada addUUID. Dicha función persiste una nueva tripleta en la base de datos semántica generando un identificador aleatorio para el sujeto, y se puede invocar varias veces dentro de las acciones realizadas en una regla (término de encabezado), generando siempre el mismo identificador. Ésto permite definir en varios términos de encabezado una instancia y todas sus propiedades.

8.1.2. Reglas de inferencia para detectar violaciones en la integración del código fuente

Se deben implementar reglas para detectar las siguientes situaciones, que incluyen las no deseadas al realizar la integración del código (mencinadas previamente en la sección 7.2.1) y detectar cuándo se dejan de cumplir las violaciones, para luego ejecutar la acción correspondiente:

- Un commit no está incluído en ningún build en el servidor de integración contínua, es decir, que no se ejecutó ningún build con ese commit o cualquiera que le siga. Ésta situación se ejemplifica en el Commit 4 de la figura 7.2.1, que es el último commit realizado y no se encuentra incluído en ningún build.

  En este caso se debe instanciar una violación de clase NoHayUnBuild con la fecha actual, el commit como recurso asociado y la persona que realizó el commit como involucrado.

  Para esto, se puede utilizar la siguiente regla:

```
[addNoBuildviolation:
        (?commit rdf:type seonh:Commit),
```

FUNCION	DESCRIPCION
isLiteral(?x) notLiteral(?x)	Valida si el argumento recibido es o no un valor literal.
equal(?x,?y) notEqual(?x,?y)	Valida si x=y o si x!=y. La igualdad es semántica.
le(?x, ?y) ge(?x, ?y)	Valida si x<=y o x>=y. Sólo devuelve verdadero si tanto x como y son números o fechas.
sum(?a, ?b, ?c)	Le setea a c el valor (a+b).
difference(?a, ?b, ?c)	Le setea a c el valor (a-b).
min(?a, ?b, ?c)	Le setea a c el valor mínimo entre a y b.
max(?a, ?b, ?c)	Le setea a c el valor máximo entre a y b.
product(?a, ?b, ?c)	Le setea a c el valor (a*b).
quotient(?a, ?b, ?c)	Le setea a c el valor (a/b).
strConcat(?a1, .. ?an, ?t)	Concatena la forma canónica (si algún argumento es una URI lo convierte a un String) de todos los argumentos excepto el último, en un valor literal, y se lo asigna a t.
now(?x)	Asigna a x un valor del tipo xsd:dateTime con la fecha actual.
noValue(?x, ?p, ?v)	Devuelve verdadero si no hay una tripleta conocida (x, p, v) en el modelo o las deducciones del motor de encadenamiento hacia adelante.
remove(n, ...)	Elimina la sentencia (tripleta) obtenida por el enésimo término de la regla.
drop(n, ...)	Elimina la sentencia (tripleta) obtenida por el enésimo término de la regla, sin disparar la ejecución de más reglas.
isDType(?l, ?t) notDType(?l, ?t)	Valida si el literal l es o no una instancia del tipo de dato definido por el recurso t

Cuadro 8.1: Algunas funciones primitivas disponibles en el framework Jena.

```
Regla      :=   regla-simple .
           o    [ regla-simple ]
           o    [ nombreRegla : regla-simple ]


regla-simple :=  termino, ... termino -> termino-encabezado, ... termino-
     encabezado // regla de Encadenamiento Hacia Adelante
           o    termino-cuerpo-encabezado <- termino, ... termino // regla de
                Encadenamiento Hacia Atras


termino-encabezado   :=   termino
           o    [ regla-simple ]


termino    :=    (nodo, nodo, nodo)        // patron de tripleta
           o     (nodo, nodo, functor)        // patron de tripleta extendido
           o     builtin(nodo, ... nodo)      // invocar funcin primitiva
                 procedural


termino-cuerpo-encabezado :=   (nodo, nodo, nodo)            // patron de
     tripleta


functor    :=    nombreDeFunctor(nodo, ... nodo) // literal estructurado


nodo       :=    uri-ref                  // Ejemplo: http://foo.com/eg
           o     prefix:localname         // Ejemplo: rdf:type
           o     <uri-ref>                // Ejemplo: <ejemplo:miuri>
           o     ?nombredevariable        // variable
           o     'un literal '            // valor de una cadena de caracteres
           o     ' literal '^^tipo         // un valor literal tipado, se
                 soportan los tipos definidos en xsd:*.
           o     nmero                     // Ejemplo: 42.5
```

Figura 8.1: Descripción informal de la sintaxis de las reglas utilizadas por el razonador de propósito general de Jena.

```
        noValue(?build sgr:includes ?commit),
        (?commit seon:isCarriedOutBy ?commitAuthor),
        now(?now),
        -> addUUID(sgr:violation/$0 rdf:type sgr:NoBuildViolation),
        addUUID(sgr:violation/$0 sgr:hasOffendingResource ?commit),
        addUUID(sgr:violation/$0 sgr:involves ?commitAuthor),
        addUUID(sgr:violation/$0 sgr:detected ?now)]
```

El primer término detecta los sujetos que son del tipo seonh:Commit, en el segundo se toman sólo los que no se encuentran incluídos en ningún build, en el tercero se toman los autores de los commits y en el cuarto se agrega la fecha actual.

Luego, se ejecutan los términos de encabezado (por cada commit que cumple los requisitos anteriores). En el primero se agrega una tripleta (con el nuevo identificador generado reemplazando $0) declarando la nueva instancia del tipo sgr:NoBuildViolation, en el segundo se agrega la relación entre la violación y el commit que se encuentra violando las buenas prácticas, en el tercero se agrega la relación entre la violación y el autor del commit, y finalmente a la violación se le agrega la fecha actual.

- Un commit está incluído en un build con estado fallido en el servidor de integración contínua, y no tiene commits siguientes, o los siguientes commits se encuentran incluídos en builds fallidos o no se encuentran incluídos en ningún build. Ésta situación se ejemplifica en el Commit 3 de la figura 7.2.1, que está incluído en el Build 2 que tiene estado fallido, y el commit siguiente (que es el Commit 4) no está incluído en ningún build.

  En este caso se debe instanciar una violación de clase NoHayUnBuildExitoso con la fecha actual, el commit como recurso asociado y la persona que realizó el commit como involucrado.

  Para esto, se puede utilizar la siguiente regla:

```
[addNoSuccessfulBuildviolation:
        (?commit rdf:type seonh:Commit),
        (?build rdf:type sgr:Build),
        (?build sgr:includes ?commit),
        noValue(?build sgr:hasStatus sgrs:Successful),
```

```
        (?commit seon:isCarriedOutBy ?commitAuthor)
        now(?now),
        -> addUUID(sgr:violation/$0 rdf:type sgr:
            NoSuccessfulBuildViolation),
        addUUID(sgr:violation/$0 sgr:hasOffendingResource ?commit),
        addUUID(sgr:violation/$0 sgr:involves ?commitAuthor),
        addUUID(sgr:violation/$0 sgr:detected ?now)]
```

El primer término detecta los sujetos que son del tipo seonh:Commit, el segundo detecta los sujetos del tipo sgr:Build, el tercero relaciona los builds con los commits que se encuentran involucrados con ellos, el cuarto detecta builds que no tienen estado exitoso, el quinto agrega al autor del commit y el sexto agrega la fecha actual.

Luego, se ejecutan los términos de encabezado (por cada commit que cumple los requisitos anteriores). En el primero se agrega una tripleta (con el nuevo identificador generado reemplazando $0) declarando la nueva instancia del tipo sgr:NoSuccessfulBuildViolation, en el segundo se agrega la relación entre la violación y el commit que se encuentra violando las buenas prácticas, en el tercero se agrega la relación entre la violación y el autor del commit, y finalmente a la violación se le agrega la fecha actual.

- Un commit que se encuentra asociado a una violación de clase NoHayUnBuild y tiene un build realizado o hay un commit posterior que lo tiene.

  En este caso se debe eliminar la violación asociada al commit.

  Para esto, se puede utilizar la siguiente regla:

```
[removeNoBuildviolation:
        (?commit rdf:type seonh:Commit),
        (?build rdf:type sgr:Build),
        (?build sgr:includes ?commit),
        (?build sgr:hasStatus ?status),
        (?violation rdf:type sgr:NoBuildViolation),
        (?violation sgr:hasOffendingResource ?commit),
        (?violation sgr:involves ?commitAuthor),
        (?violation ?predicate ?object),
        -> drop(4,5,6,7)]
```

El primer término detecta los sujetos que son del tipo seonh:Commit, el segundo detecta los sujetos del tipo sgr:Build, el tercero relaciona los builds con los commits que se encuentran involucrados con ellos, el cuarto recupera el estado del build, el quinto detecta los sujetos del tipo sgr:NoBuildViolation, el sexto los relaciona con el commit que originó la violación, el séptimo recupera el autor del commit y el octavo recupera cualquier otra relación que pueda tener la violación detectada previamente.

Luego, se ejecuta el término de encabezado que elimina todas las tripletas relacionadas con la violación del commit, debido a que éste ahora está incluido en un build.

- Un commit que se encuentra asociado a una violación de clase NoHayUnBuildExitoso y tiene un build con estado exitoso o hay un commit posterior que lo tiene.

  En este caso se debe eliminar la violación asociada al commit.

```
[removeNoSuccessfulBuildviolationPrevious:
        (?violation rdf:type sgr:NoSuccessfulBuildViolation),
        (?violation sgr:hasOffendingResource ?violationCommit),
        (?violation sgr:involves ?commitAuthor),
        (?violation ?predicate ?object),
        (?violationCommit seonh:constituesVersion ?violationVersion),
        (?violationVersion seon: hasIdentifier ?violationRevision),
        (?successfulBuild rdf:type sgr:Build),
        (?successfulBuild sgr:hasStatus sgrs:Successful),
        (?successfulBuild sgr:includes ?goodCommit),
        (?goodCommit seonh:constituesVersion ?goodCommitVersion),
        (?goodCommitVersion seon:hasIdentifier ?goodCommitRevision),
        le(?violationRevision,?goodCommitRevision),
        -> drop(0,1,2,3)]
```

El primer término detecta las violaciones del tipo sgr:NoSuccessfulBuildViolation, los siguientes tres términos agregan los datos de la violación, el quinto término obtiene el commit involucrado en la violación, el sexto término obtiene el número de revisión del commit, los términos séptimo, octavo y noveno buscan build exitoso, y los últimos tres términos verifican que la versión del commit fallido sea menor que la del build exitoso

encontrado.

Luego, se ejecuta el término de encabezado que elimina todas las tripletas relacionadas con la violación del commit, debido a que ahora hay un build exitoso posterior.

8.1.3. Reglas de inferencia para detectar la edición simultánea del código fuente

Se deben implementar reglas para detectar las siguientes situaciones, que incluyen las no deseadas al realizar la edición del código fuente (mencinadas previamente en la sección 7.2.1) y detectar cuándo se dejan de cumplir las violaciones, para luego ejecutar la acción correspondiente:

- Una entidad de código está siendo modificada por más de un desarrollador al mismo tiempo. Un ejemplo de esta situación se ve en la figura 7.2.2, donde ClaseDos está siendo modificada por Juan y por Carlos.

  En éste caso se da una situación particular debido a que si hay varias personas modificando una misma entidad de código no son distintas violaciones sino la misma pero con varias personas involucradas, por lo tanto la siguiente regla sólo detecta que una entidad de código está siendo modificada por más de una persona e instancia la violación de tipo ConflictoEnEdiciónDeArchivo con la fecha actual, uno de los desarrolladores involucrados (al azar) y la entidad de código que está siendo modificada por varios desarrolladores:

```
[addFileEditionConflictViolation:
        (?codeEntity rdf:type seonc:CodeEntity),
        (?developer seonch:modifiesCodeEntity ?codeEntity),
        (?anotherDeveloper seonch:modifiesCodeEntity ?codeEntity),
        notEqual(?developer,?anotherDeveloper),
        noValue(?violation sgr:hasOffendingResource ?codeEntity),
        noValue(?violation sgr:involves ?developer),
        now(?now),
        -> addUUID(sgr:violation/$0 rdf:type sgr:FileEditionConflict),
        addUUID(sgr:violation/$0 sgr:hasOffendingResource ?codeEntity),
        addUUID(sgr:violation/$0 sgr:involves ?developer),
        addUUID(sgr:violation/$0 sgr:detected ?now)]
```

El primer término detecta las entidades de código, el segundo y el tercero a los desarrolladores que están modificando la misma entidad de código, el cuarto se asegura de que sean desarrolladores distintos, el quinto y el sexto validan que no haya una violación instanciada involucrando a la entidad de código ni al desarrollador, y el séptimo obtiene la fecha actual.

El primer término de encabezado agrega la nueva instancia de tipo sgr:FileEditionConflict, y los siguientes la relacionan con la fecha actual, el desarrollador encontrado y la entidad de código involucrada.

- Ésta regla sirve para agregar los desarrolladores involucrados que faltan a la violación detectada con la regla anterior.

```
[addInvolvedDeveloper:
        (?codeEntity rdf:type seonc:CodeEntity),
        (?developer seonch:modifiesCodeEntity ?codeEntity),
        (?anotherDeveloper seonch:modifiesCodeEntity ?codeEntity),
        notEqual(?developer,?anotherDeveloper),
        (?violation rdf:type sgr:FileEditionConflict),
        (?violation sgr:hasOffendingResource ?codeEntity),
        noValue(?violation sgr:involves ?developer),
        now(?now),
        -> (?violation sgr:involves ?developer)]
```

Los primeros cuatro términos son para detectar las entidades de código y los desarrolladores involucrados en una violación de la misma manera que en la regla anterior, en la quinta detecta la instancia de la violación, en la sexta relaciona la entidad de código encontrada previamente con la de la violación, en la séptima valida que el desarrollador detectado no esté relacionado todavía con la violación y el último obtiene la fecha actual.

El término de encabezado agrega la relación entre la violación y el desarrollador involucrado.

- También se debe detectar cuando deja de haber más de un desarrollador modificando la misma entidad de código, para eliminar la vio-

lación instanciada previamente. La idea de ésta regla es detectar cuando sólo queda un desarrollador modificando una entidad de código que está involucrada en un violación del tipo ConflictoEnEdiciónDeArchivo y eliminarla.

```
[removeFileEditionConflictViolation:
        (?violation rdf:type sgr:FileEditionConflict),
        (?violation sgr:hasOffendingResource ?codeEntity),
        (?violation ?predicate ?object),
        (?developer seonch:modifiesCodeEntity ?codeEntity),
        noValue(?anotherDeveloper seonch:modifiesCodeEntity ?
            codeEntity),
        notEqual(?developer,?anotherDeveloper),
        -> drop(0,2)]
```

El primer término detecta las violaciones del tipo ConflictoEnEdiciónDeArchivo, el segundo obtiene las entidades de código asociadas, el tercero obtiene cualquier relación que tengan con otro objeto, el cuarto obtiene las relaciones con desarrolladores, el quinto valida que no haya otro desarrollador modificando la misma entidad de código y el sexto valida que no sean los mismos desarrolladores.

El término de encabezado elimina las violaciones detectadas junto con todas sus relaciones.

8.1.4. Reglas de inferencia para detectar violaciones en la cobertura de tests

Se deben implementar reglas para detectar las siguientes situaciones y ejecutar la acción correspondiente:

- Una entidad de código no tiene información sobre su cobertura de tests. En la figura 7.2.3 se puede observar un ejemplo de ésta situación en ClaseUno, que no tiene realizada ninguna medición de su cobertura de tests.

  En este caso se debe instanciar una violación del tipo NoHayMediciónDeCoberturaDeTests con la fecha actual y la entidad de código relacionada. Por simplicidad no se tiene en cuenta que haya personas

directamente involucradas, aunque se podría incluir a todos los desarrolladores o a el que realizó la última modificación. Tampoco se tiene en cuenta la actualización de la medición.

```
[noTestCoverageInfo:
        (?codeEntity rdf:type seonc:CodeEntity),
        noValue(?measurement seonm:measuresArtifact ?codeEntity),
        now(?now),
        -> addUUID(sgr:violation/$0 rdf:type sgr:
            NoTestCoverageMeasurement),
        addUUID(sgr:violation/$0 sgr:hasOffendingResource ?codeEntity),
        addUUID(sgr:violation/$0 sgr:detected ?now)]
```

Los dos primeros términos de la regla obtienen las entidades de código que no tienen ninguna medición de cobertura de tests realizada y el tercero obtiene la fecha actual.

Los términos de encabezado instancian la violación de tipo NoHayMediciónDeCoberturaDeTests con la fecha actual y la entidad de código involucrada.

- Una entidad de código tiene una cobertura de tests menor al 100 %. En la figura 7.2.3 se puede observar un ejemplo de ésta situación en ClaseDos, que tiene una cobertura de tests del 72 %.

Se debe instanciar una violación del tipo NoHaySuficienteCoberturaDeTests con la fecha actual y la entidad de código relacionada.

```
[notEnoughTestCoverage:
        (?codeEntity rdf:type seonc:CodeEntity),
        (?measurement seonm:measuresArtifact ?codeEntity),
        (?measurement seonm:usesMeasure sgr:TestCoverage),
        (?measurement sgr:hasValue ?value),
        lessThan(?value, 100),
        now(?now),
        -> addUUID(sgr:violation/$0 rdf:type sgr:
            NotEnoughTestCoverage),
        addUUID(sgr:violation/$0 sgr:hasOffendingResource ?codeEntity),
        addUUID(sgr:violation/$0 sgr:detected ?now)]
```

Los primeros cuatro términos obtienen las entidades de código con sus medidas de cobertura de tests, el quinto valida que sean menores a 100 %, y el último obtiene la fecha actual.

Los términos de encabezado instancian la violación del tipo NoHaySuficienteCoberturaDeTests con la fecha actual y la entidad de código relacionada.

- Se realiza una medición de cobertura de tests sobre una entidad de código que no tenía ninguna realizada previamente.

  En este caso de debe eliminar la violación de tipo NoHayMediciónDeCoberturaDeTests instanciada anteriormente y todas sus relaciones.

```
[removeNoTestCoverageMeasurement:
        (?codeEntity rdf:type seonc:CodeEntity),
        (?measurement seonm:measuresArtifact ?codeEntity),
        (?violation rdf:type sgr:NoTestCoverageMeasurement),
        (?violation sgr:hasOffendingResource ?codeEntity),
        (?violation ?predicate ?object),
        -> drop(2,3,4)]
```

  La idea de ésta regla es detectar las entidades de código que tienen una medición realizada y están asociadas a una violación del tipo NoHayMediciónDeCoberturaDeTests, para eliminar la violación y todas sus relaciones.

- Se realiza una medición de cobertura de tests que tiene un valor de 100 % sobre una entidad de código que tenía una medición realizada con un valor menor.

  La siguiente regla detecta las entidades de código que tienen una medición de cobertura de tests realizada con un valor de 100 % y están asociadas a una violación del tipo NoHaySuficienteCoberturaDeTests, para luego eliminar la violación y todas sus relaciones.

```
[removeTestCoverageViolation:
        (?codeEntity rdf:type seonc:CodeEntity),
        (?measurement seonm:measuresArtifact ?codeEntity),
        (?measurement seonm:usesMeasure sgr:TestCoverage),
```

```
(?measurement sgr:hasValue ?value),
equal(?value, 100),
(?violation rdf:type sgr:NotEnoughTestCoverage),
(?violation sgr:hasOffendingResource ?codeEntity),
(?violation ?predicate ?object),
-> drop(5,6,7)]
```

8.2. Reporte de violaciones a las mejores prácticas

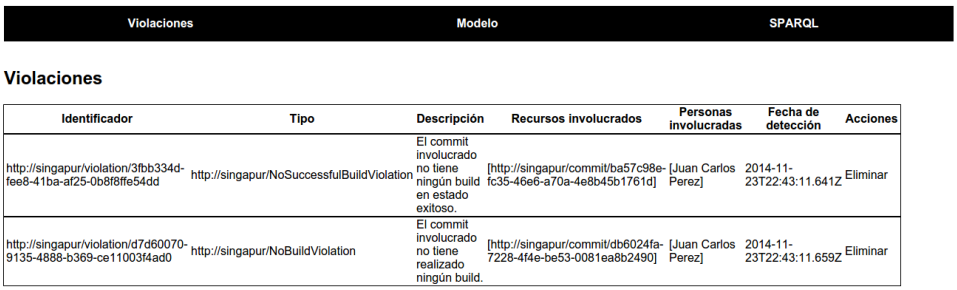
El reporte de las violaciones detectadas a las mejores prácticas se realiza de dos maneras complementarias. Por un lado, la aplicación web Singapur provee una interfaz gráfica donde se pueden ver las violaciones detectadas junto con sus datos y por otro lado se encuentran los agentes consumidores de datos que pueden reportar las violaciones en el contexto de trabajo.

8.2.1. Reporte genérico de violaciones detectadas por Singapur

En la figura 8.2.1 se puede observar la página web de reporte de las violaciones detectadas, donde se visualizan las mismas junto con sus datos. Las violaciones visualizadas son las que se encuentran instanciadas en el modelo de la aplicación que, como se vio en las reglas explicadas previamente, se van borrando a medida que se resuelven. En el caso de las violaciones detectadas que no se pueden corregir, se pueden eliminar desde la misma página web de reporte clickeando el boton eliminar al costado de cada violación o manualmente mediante una petición al endpoint SPARQL incluído en la aplicación web de Singapur. Por ejemplo, si hubiera una mejor práctica que enunciara que no se puede realizar ningún commit en una franja horaria determinada, una vez ocurrida una violación a la misma no se puede revertir la situación sino que debe ser corregida manualmente mediante otro commit.

8.2.2. Agente consumidor de datos

La otra manera de reportar las violaciones es mediante un agente consumidor de datos que realiza consultas SPARQL al servidor de Singapur. Dichas consultas pueden ser customizadas de cualquier manera para proveer



Violaciones | Modelo | SPARQL

Violaciones

Identificador	Tipo	Descripción	Recursos involucrados	Personas involucradas	Fecha de detección	Acciones
http://singapur/violation/3fbb334d-fee8-41ba-af25-0b8f8ffe54dd	http://singapur/NoSuccessfulBuildViolation	El commit involucrado no tiene ningún build en estado exitoso.	[http://singapur/commit/ba57c98e-fc35-46e6-a70a-4e8b45b1761d]	[Juan Carlos Perez]	2014-11-23T22:43:11.641Z	Eliminar
http://singapur/violation/d7d60070-9135-4888-b369-ce11003f4ad0	http://singapur/NoBuildViolation	El commit involucrado no tiene realizado ningún build.	[http://singapur/commit/db6024fa-7228-4f4e-be53-0081ea8b2490]	[Juan Carlos Perez]	2014-11-23T22:43:11.659Z	Eliminar

Figura 8.2: Página web de reporte de violaciones de Singapur.

sólo la información relevante para la función que cumple dicho agente. La misma página web de reporte de violaciones de Singapur es una forma de agente consumidor de datos que pide todas las violaciones existentes y las reporta de una manera en particular. Otra posibilidad es que el agente reporte un determinado tipo de violaciones para una persona en particular y se muestren en una herramienta de trabajo. Las posibilidades de reportar las violaciones son muy amplias ya que pueden realizarse con cualquier criterio, ya sea por tipo de violación, para un grupo específico de personas, relacionadas a un tipo de recursos, ocurridas en un determinado rango de fechas o con cualquier otro criterio que sea necesario.

Agente consumidor de datos de violaciones por conflicto en edición de un archivo

Una posibilidad interesante que surge con la arquitectura planteada en este trabajo, es la de poder reportar los problemas detectados en el contexto de trabajo. En particular, en el caso de que haya una violación a la buena práctica de evitar conflictos en la edición de archivos, se puede informar dentro del mismo IDE en que trabajan los desarrolladores (Eclipse).

Para realizar ésto es necesario desarrollar una extensión que al comenzar a modificar un archivo (o cuando se considere necesario) realice una consulta a Singapur para verificar que no hay ninguna violación del tipo ConflictoEnEdiciónDeArchivo asociada al recurso que se comenzó a modificar. Se puede utilizar la siguiente consulta para ello:

```
PREFIX sgr: <http://singapur/>

select ?usuarioInvolucrado
where {?violation a sgr:FileEditionConflict .
       FILTER NOT EXISTS {?violation sgr:resolved ?date } .
       ?violation sgr:hasOffendingResource <identificadorArchivo> .
       ?violation sgr:involves ?usuarioInvolucrado .
       FILTER (?usuarioInvolucrado != <identificadorUsuario>)}
```

La consulta verifica si hay alguna violación del tipo ConflictoEnEdiciónDeArchivo asociada al archivo, y si la tiene devuelve los nombres de los usuarios involucrados en la misma, exceptuando al del usuario que realizó la consulta. Para ésto es necesario contar con el identificador del usuario y el del recurso que se comenzó a modificar.

En caso de que la consulta devuelva resultados, se puede mostrar dentro del Eclipse una notificación de la situación junto con los usuarios que se encuentran involucrados para que pueda comunicarse con ellos y resolver el problema. En la figura 8.2.2 se muestra un ejemplo de cómo se puede visualizar ésta situación.

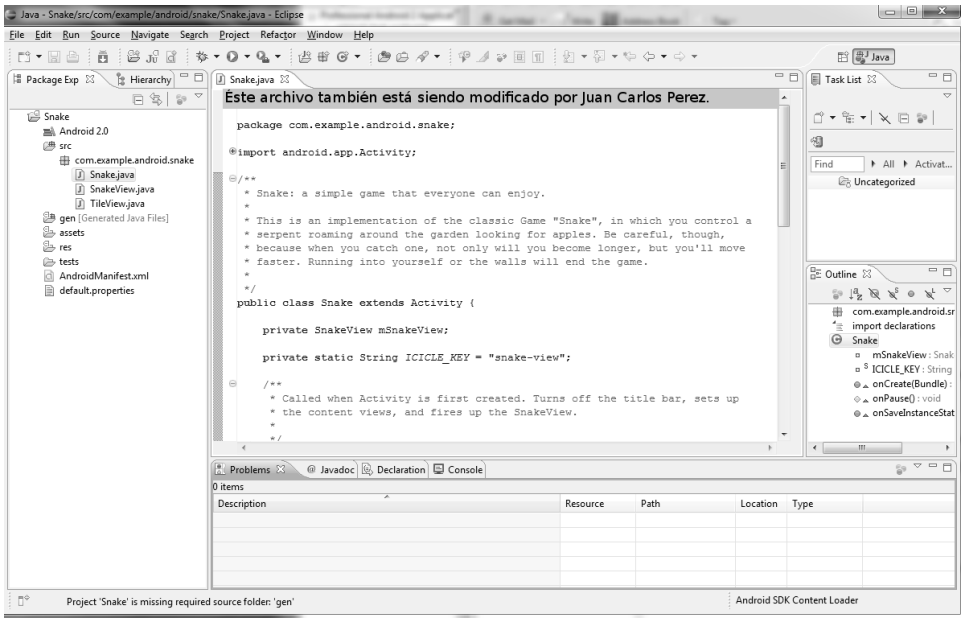


Figura 8.3: IDE Eclipse con un agente consumidor de datos de Singapur integrado para mostrar de forma contextualizada la información de las violaciones detectadas.

Capítulo 9

Evaluación

9.1. Modelado y detección de las mejores prácticas

Para explorar el modelado de las mejores prácticas y la detección de sus violaciones, se eligieron tres casos con características diferentes que son comunes a una gran parte de los problemas que se quieran modelar.

Las buenas prácticas que se tomaron son:

- Integracion exitosa del codigo fuente
- Evitar conflictos en las modificaciones al codigo fuente
- Cobertura de tests completa

El modelado de las mejores prácticas posee dos aspectos. Por un lado se encuentra la descripción, que no tiene grandes limitaciones debido a que las descripciones de las clases en el modelo semántico no limitan los posibles atributos sino que describe algunos conceptos que pueden utilizarse. Es decir, que además de los atributos que se describen en el capítulo referente al modelo 7, se puede utilizar cualquier tipo de recursos para la misma, desde texto a videos o cualquier tipo de representación de datos. Por otro lado se encuentra la descripción de las violaciones a la práctica, cuya limitación principal es que se deben poder expresar en función del lenguaje de reglas de Apache Jena utilizado por Singapur, por lo tanto, es en éste aspecto donde es importante evaluar las disintas posibilidades que puedan surgir.

Básicamente, las violaciones son situaciones que no se ajustan a las buenas prácticas definidas. Las posibilidades y diferencias conceptuales más importantes que puede haber entre las distintas violaciones (que son plasmadas

en la estructura de los datos del modelo semántico) son si dicha situación puede ser revertida para ajustarse o no a la buena práctica que violan, y si aparecen inevitablemente debido a que son parte de un procedimiento que tiene estados intermedios que son inaceptables si no se finaliza. Dichos tipos de situaciones se encuentran modeladas respectivamente en las subclases de Violación llamadas ViolaciónResoluble y ViolaciónInevitable. Éstas diferencias fundamentales son las que deben poder ser expresadas mediante las reglas de Singapur con el fin de detectar cuando ocurren.

Como se puede observar en la figura 7.1.3, las subclases de violaciones que surgen a partir de las mejores prácticas elegidas cubren los distintos tipos fundamentales de violaciones. En los capítulos 7 y 8 se ve que tanto las mejores prácticas elegidas como las violaciones que pueden ocurrir se pueden modelar y detectar utilizando Singpur, lo que evidencia que el mecanismo de modelado y detección utilizado es suficientemente potente y expresivo para ser utilizado en una gran cantidad de escenarios reales.

9.2. Pruebas automatizadas del funcionamiento de Singapur

Para validar el correcto funcionamiento de Singapur, se implementó una batería de tests automatizados utilizando JUnit[1] que cubre los posibles casos donde se deberían detectar violaciones a las mejores prácticas que se tomaron como ejemplo. Dichos tests son parte del código fuente de la aplicación, y se fueron desarrollando y manteniendo a la par de la misma para verificar que los cambios introducidos no produzcan errores en su funcionalidad, lo que resulta fundamental para asegurar la calidad y mantenibilidad del software. Por lo tanto, el resultado de la ejecución de los tests es exitoso tanto en su última versión como lo fue durante su desarrollo, luego de realizar cada cambio al código fuente.

Para realizar las pruebas se ejecuta la aplicación Singapur con las reglas correspondientes a cada escenario, y se utiliza un agente generador de datos y otro consumidor de datos desde las clases que implementan las pruebas unitarias para ir creando los datos necesarios y realizar las validaciones correspondientes. Al tratarse de pruebas unitarias, antes de cada prueba se vuelve la base de datos semántica al estado original (sin datos de instancias,

[1] JUnit es un framework que permite realizar pruebas unitarias de aplicaciones en Java. Forma parte de la familia de frameworks de tests de unidad que conforman xUnit.

```
select ?violation {?violation a <http://singapur/Violation>}
```

Figura 9.1: Consulta SPARQL que se utiliza para obtener la cantidad total de instancias de violaciones detectadas.

sólo con las reglas de detección de violaciones).

9.2.1. Pruebas de integración exitosa del código fuente

Se implementaron una serie de métodos en Java para generar y consultar datos en la base de datos semántica. Para crear un commit se implementó el método createCommit que retorna el identificador del recurso RDF que representa al nuevo commit creado. También se implementaron métodos para saber la cantidad de instancias de violaciones en un determinado momento. Algunos ejemplos de esto son el método getViolationsCount que obtiene la cantidad total de violaciones sin importar su tipo particular, para obtener dicha cantidad cuenta los resultados obtenidos por la consulta SPARQL que se muestra en la figura 9.2.1, y el método existsNoBuildViolationForCommit(commitId) que valida la existencia de una violación del tipo NoHayUnBuild que se corresponda con el commit cuyo identificador es pasado como parámetro.

Nuevo commit sin build

El primer escenario que se probó es que al crearse una nueva instancia de commit se detecte que el mismo no tiene un build realizado y se instancie una violación NoHayUnBuild correspondiente al nuevo commit. En la figura 9.2.1 se muestra el código de esta prueba.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación (comprobando que el método getViolationsCount devuelve cero).

2. Crear un nuevo commit.

```
/**
 * Cuando se crea un commit debe detectarse la violacion de que el
     commit creado no tiene un build.
 */
@Test
public void commitWithNoBuild() {
    // Se valida que no hay violaciones a las buenas practicas
    assertEquals(0, getViolationsCount());

    String commitId = createCommit();

    // Se valida que haya una violacion del tipo NoHayUnBuild que
        corresponde al nuevo commit
    assertEquals(1, getViolationsCount());
    assertTrue(existsNoBuildViolationForCommit(commitId));
}
```

Figura 9.2: Código del caso de prueba unitario donde se valida que al crear un nuevo commit y no realizar ningún build, se detecte la violación NoHayUnBuild.

3. Validar que hay sólo una violación, que la misma sea del tipo NoHayUnBuild, y que se corresponda al commit creado previamente.

Nuevo commit con build fallido

Cuando se crea un commit y luego se hace un build sobre ese commit y falla, debe detectarse la violacion de que ese commit no tiene un build exitoso.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear un nuevo commit.
3. Crear un build con estado fallido que incluya al commit creado en el paso anterior.

4. Validar que hay sólo una violación, que la misma sea del tipo NoHayUnBuildExitoso, y que se corresponda al commit creado previamente.

Nuevo commit con build exitoso

Luego de crear un commit y hacer un build que termina en estado exitoso, no debe haber ninguna violacion.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear un nuevo commit.
3. Crear un build con estado exitoso que incluya al commit creado en el paso anterior.
4. Validar que no existe ninguna instancia de violación.

Último commit sin build

Se crean varios commits donde el ultimo no tiene build, por lo tanto debe detectarse una violacion de que el ultimo no tiene build.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear cinco nuevos commits.
3. Crear cuatro builds con estado exitoso que incluyan a los primeros cuatro commits creados en el paso anterior.
4. Validar que hay sólo una violación, que la misma sea del tipo NoHayUnBuild, y que se corresponda al último commit creado.

Último commit con build fallido

Se crean varios commits donde el ultimo tiene un build fallido, por lo tanto debe detectarse una violacion de que el ultimo tiene build fallido.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear cinco nuevos commits.
3. Crear cuatro builds con estado exitoso que incluyan a los primeros cuatro commits creados en el paso anterior.
4. Crear un build con estado fallido que incluya al último commit creado.
5. Validar que hay sólo una violación, que la misma sea del tipo NoHayUnBuildExitoso, y que se corresponda al último commit creado.

Último commit con build exitoso

Se crean varios commits donde el ultimo tiene un build exitoso, por lo tanto no debe haber ninguna violación.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear cuatro nuevos commits.
3. Crear cuatro builds con estado exitoso que incluyan a los commits creados en el paso anterior.
4. Validar que no existe ninguna instancia de violación.

Último commit con build exitoso que incluye varios commits previos

Se crean varios commits y un único build exitoso que los cubre a todos, por lo tanto no debe haber ninguna violación.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear cinco nuevos commits.
3. Crear un build con estado exitoso que incluya a los commits creados en el paso anterior.
4. Validar que no existe ninguna instancia de violación.

9.2.2. Pruebas de evitar conflicto en edición de archivo

Detectar conflicto en edición de archivo y su resolución

Se crea una entidad de código y dos usuarios lo editan al mismo tiempo, generando una violación de conflicto en edición de archivo, luego uno deja de modificarlo y se elimina la violación detectada previamente.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear una entidad de código.
3. Crear dos desarrolladores (Juan y Carlos).
4. Validar que no existe ninguna instancia de violación.
5. Crear la relación de que Juan está modificando la entidad de código.
6. Validar que no existe ninguna instancia de violación.
7. Crear la relación de que Carlos está modificando la entidad de código.
8. Validar que existe una violación del tipo ConflictoEnEdicionDeArchivo que involucra a la entidad de código creada previamente y a los desarrolladores Juan y Carlos.
9. Eliminar la relación de que Carlos está modificando la entidad de código.
10. Validar que no existe ninguna instancia de violación.

Detectar conflicto en edición de archivo y su resolución con tres usuarios involucrados

Se crea una entidad de código y tres usuarios lo editan al mismo tiempo, generando una violación de conflicto en edición de archivo, luego dos de ellos dejan de modificarlo y se elimina la violación detectada previamente.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear una entidad de código.

3. Crear tres desarrolladores (Juan, Carlos y Roberto).
4. Validar que no existe ninguna instancia de violación.
5. Crear la relación de que Juan está modificando la entidad de código.
6. Validar que no existe ninguna instancia de violación.
7. Crear la relación de que Carlos está modificando la entidad de código.
8. Validar que no existe ninguna instancia de violación.
9. Crear la relación de que Roberto está modificando la entidad de código.
10. Validar que existe una violación del tipo ConflictoEnEdicionDeArchivo que involucra a la entidad de código creada previamente y a los desarrolladores Juan, Carlos y Roberto.
11. Eliminar la relación de que Carlos está modificando la entidad de código.
12. Validar que existe una violación del tipo ConflictoEnEdicionDeArchivo que involucra a la entidad de código creada previamente y a los desarrolladores Juan y Roberto.
13. Eliminar la relación de que Juan está modificando la entidad de código.
14. Validar que no existe ninguna instancia de violación.

9.2.3. Pruebas de cobertura completa de tests

Código sin medición de cobertura de tests

Al crear una entidad de código sin realizar una medición de su cobertura de tests, se debe detectar una violación del tipo NoHayMediciónDeCoberturaDeTests.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear una entidad de código.
3. Validar que existe una violación del tipo NoHayMediciónDeCoberturaDeTests correspondiente a la entidad de código creada.

Código con cobertura incompleta (99 %)

Al crear una entidad de código con una cobertura de tests del 99 % se debe detectar una violación del tipo NoHaySuficienteCoberturaDeTests.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear una entidad de código con una medición de cobertura de tests del 99 %.
3. Validar que hay una violación del tipo NoHaySuficienteCoberturaDeTests correspondiente a la entidad de código recién creada.

Código con cobertura incompleta (50 %)

Al crear una entidad de código con una cobertura de tests del 50 % se debe detectar una violación del tipo NoHaySuficienteCoberturaDeTests.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear una entidad de código con una medición de cobertura de tests del 50 %.
3. Validar que hay una violación del tipo NoHaySuficienteCoberturaDeTests correspondiente a la entidad de código recién creada.

Código con cobertura incompleta (0 %)

Al crear una entidad de código con una cobertura de tests del 0 % se debe detectar una violación del tipo NoHaySuficienteCoberturaDeTests.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear una entidad de código con una medición de cobertura de tests del 0 %.
3. Validar que hay una violación del tipo NoHaySuficienteCoberturaDeTests correspondiente a la entidad de código recién creada.

Código con cobertura de tests completa

Al crear una entidad de código con una cobertura de tests del 100 % no se debe detectar ninguna violación.

El procedimiento para realizar esta prueba es el siguiente:

1. Validar que no existe ninguna instancia de violación.
2. Crear una entidad de código con una medición de cobertura de tests del 100 %.
3. Validar que no existe ninguna instancia de violación.

9.3. Escenario real

Además de las pruebas unitarias automatizadas con JUnit, se ejecutó una prueba con datos tomados de un proyecto de desarrollo de software real de una aplicación móvil para Android. Se recolectó del servidor de integración contínua la información de los commits y los builds realizados, y mediante un script se convirtió la información en datos semánticos que fueron enviados al servidor de Singapur para que los procese y detecte las violaciones que surgieron a la buena práctica de integrar correctamente los cambios al código fuente.

El proyecto consta de 310 commits integrados en 442 builds. Para esta prueba se generó un nuevo conjunto de reglas, ya que en vez de tener en tiempo real las instancias de violaciones activas, es importante que las mismas queden registradas con la fecha en que fueron detectadas y cuando se resuelven se registra la fecha de resolución.

La máquina que se utilizó para realizar la prueba tenía las siguientes características:

- Procesador AMD Phenom II Triple-Core 2,8GHz
- Memoria Ram 2GB
- Sistema Operativo Ubuntu Desktop 14.04

La ejecución de la prueba duró aproximadamente 4 segundos, en los cuales se ingresó a Singapur toda la información correspondiente a los commits y los builds realizados, y se detectaron 366 violaciones (todas resueltas).

Luego de realizar la prueba se corrieron una serie de consultas SPARQL para evaluar los resultados obtenidos. De las 366 violaciones detectadas 310 son del tipo NoHayUnBuild, lo que se corresponde con los 310 commits realizados, y todas están resueltas porque tienen un build que las contiene. Las 56 violaciones restantes son del tipo NoHayUnBuildExitoso y se encuentran resueltas. Se verificó que éste resultado es correcto porque hay 56 commits realizados cuyo primer build finalizó con estado fallido, generando las violaciones mencionadas previamente, y el último build tiene estado exitoso y cubre al último commit, por lo tanto todas las violaciones del tipo NoHayUnBuildExitoso tienen que estar resueltas.

Además, se inspeccionaron algunos casos particulares para verificar que sean correctos. En la siguiente figura se muestra una violación del tipo NoHayUnBuild tomada al azar junto con los datos del commit y el build involucrados:

```
<http://singapur/commit/b9cf1047-0951-4082-81aa-c658bce72e57>
        a                           seonh:Commit ;
        seonh:constitutesVersion    <http://singapur/version/9ec47735-f855-445
            e-870c-adb37f47c099> ;
        seon:isCarriedOutBy         "fcarmine"^^xsd:string ;
        sgr:hasTimestamp            "2014-11-26T19:14:45"^^xsd:dateTime .

<http://singapur/violation/ec302846-59bc-4d56-9485-f894aba83156>
        a                           dcm:Event , sgr:NoBuildViolation , sgr:
            ResolvableViolation , sgr:InevitableViolation , sgr:Violation ;
        dce:relation                <http://singapur/commit/b9cf1047
            -0951-4082-81aa-c658bce72e57> ;
        dct:audience                "fcarmine"^^xsd:string ;
        dct:created                 "2014-11-26T19:14:45"^^xsd:dateTime ;
        dct:modified                "2014-11-26T16:16:16"^^xsd:dateTime ;
        sgr:detected                "2014-11-26T19:14:45"^^xsd:dateTime ;
        sgr:hasOffendingResource <http://singapur/commit/b9cf1047
            -0951-4082-81aa-c658bce72e57> ;
        sgr:involves                "fcarmine"^^xsd:string ;
        sgr:resolved                "2014-11-26T16:16:16"^^xsd:dateTime .

<http://singapur/build/mobile-red-social/363>
        a                   <http://se-on.org/ontologies/general/2012/2/main.
            owl#Artifact> , sgr:Build ;
        dce:relation        <http://singapur/commit/b9cf1047-0951-4082-81aa
            -c658bce72e57> ;
        sgr:hasStatus       sgrs:Successful ;
```

```
        sgr:hasTimestamp "2014-11-26T16:16:16"^^xsd:dateTime ;
        sgr:includes     <http://singapur/commit/b9cf1047-0951-4082-81aa
            -c658bce72e57> .
```

Como se puede ver la fecha de creación del commit (el atributo sgr:hasTimestamp) coincide con la fecha de detección de la violación, y la fecha de cración del build coincide con la fecha de resolución de la violación. Por lo tanto, se corrobora que la violación se creó cuando se realizó el commit y se resolvió cuando se creó el build.

También se verificó un caso de violación del tipo NoHayUnBuildExitoso. En la siguiente figura se puede ver un commit realizado, junto con el build que lo incluye con estado fallido, cuatro builds posteriores también con estado fallido y un build con estado exitoso:

```
<http://singapur/version/bfb264c1-b9d3-496e-b782-31ac9e610bc7>
        a                   seonh:Version ;
        seon: hasIdentifier   13306 .

<http://singapur/commit/db2f99df-7964-496d-b8fe-345448907b29>
        a                           seonh:Commit ;
        seonh:constitutesVersion    <http://singapur/version/bfb264c1-b9d3
            -496e-b782-31ac9e610bc7> ;
        seon:isCarriedOutBy         " jrisso"^^xsd:string  ;
        sgr:hasTimestamp            "2015-01-29T14:01:05"^^xsd:dateTime .

<http://singapur/build/mobile-red-social/458>
        a                   <http://se-on.org/ontologies/general/2012/2/main.
            owl#Artifact> , sgr:Build ;
        dce: relation       <http://singapur/commit/db2f99df-7964-496d-b8fe
            -345448907b29> ;
        sgr:hasStatus       sgrs:Failure  ;
        sgr:hasTimestamp "2015-01-29T11:02:57"^^xsd:dateTime ;
        sgr:includes        <http://singapur/commit/db2f99df-7964-496d-b8fe
            -345448907b29> .

<http://singapur/build/mobile-red-social/459>
        a                   <http://se-on.org/ontologies/general/2012/2/main.
            owl#Artifact> , sgr:Build ;
        sgr:hasStatus       sgrs:Failure  ;
        sgr:hasTimestamp "2015-01-29T11:04:18"^^xsd:dateTime .
```

```
<http://singapur/build/mobile-red-social/460>
        a                   <http://se-on.org/ontologies/general/2012/2/main.
            owl#Artifact> , sgr:Build ;
        dce:relation        <http://singapur/commit/d49425d2-bb61-4844-83
            bf-322805954c16> ;
        sgr:hasStatus       sgrs:Failure  ;
        sgr:hasTimestamp "2015-01-29T11:33:10"^^xsd:dateTime ;
        sgr:includes        <http://singapur/commit/d49425d2-bb61-4844-83
            bf-322805954c16> .

<http://singapur/build/mobile-red-social/461>
        a                   <http://se-on.org/ontologies/general/2012/2/main.
            owl#Artifact> , sgr:Build ;
        dce:relation        <http://singapur/commit/0da07a68-b9be-4f74-94b2
            -569ee89e530e> ;
        sgr:hasStatus       sgrs:Failure  ;
        sgr:hasTimestamp "2015-01-29T12:03:10"^^xsd:dateTime ;
        sgr:includes        <http://singapur/commit/0da07a68-b9be-4f74-94b2
            -569ee89e530e> .

<http://singapur/build/mobile-red-social/462>
        a                   <http://se-on.org/ontologies/general/2012/2/main.
            owl#Artifact> , sgr:Build ;
        dce:relation        <http://singapur/commit/27cf278d-ff49-40a3-9ac3
            -2165e8d328b1> ;
        sgr:hasStatus       sgrs:Failure  ;
        sgr:hasTimestamp "2015-01-29T15:48:11"^^xsd:dateTime ;
        sgr:includes        <http://singapur/commit/27cf278d-ff49-40a3-9ac3
            -2165e8d328b1> .

<http://singapur/build/mobile-red-social/463>
        a                   <http://se-on.org/ontologies/general/2012/2/main.
            owl#Artifact> , sgr:Build ;
        dce:relation        <http://singapur/commit/aacd287e-1e1f-4d7b-9d92
            -83837d06c2b5> ;
        sgr:hasStatus       sgrs:Successful  ;
        sgr:hasTimestamp "2015-01-29T16:23:43"^^xsd:dateTime ;
        sgr:includes        <http://singapur/commit/aacd287e-1e1f-4d7b-9d92
            -83837d06c2b5> .

<http://singapur/violation/8f7585f8-74f3-49f1-856f-bf244389a874>
        a                           dcm:Event , sgr:Violation  , sgr:
            NoSuccessfulBuildViolation , sgr:ResolvableViolation ;
        dce:relation                <http://singapur/commit/db2f99df
```

```
        -7964-496d-b8fe-345448907b29> ;
    dct:audience                "jrisso"^^xsd:string  ;
    dct:created                 "2015-01-29T11:02:57"^^xsd:dateTime ;
    dct:modified                "2015-01-29T16:23:43"^^xsd:dateTime ;
    sgr:detected                "2015-01-29T11:02:57"^^xsd:dateTime ;
    sgr:hasOffendingResource <http://singapur/commit/db2f99df
        -7964-496d-b8fe-345448907b29> ;
    sgr:involves                "jrisso"^^xsd:string  ;
    sgr:resolved                "2015-01-29T16:23:43"^^xsd:dateTime .
```

La violación fue detectada correctamente ya que su fecha de detección coincide con la fecha de creación del commit, y su fecha de resolución coincide con la fecha de creación del último build.

Capítulo 10

Conclusiones y trabajo futuro

El objetivo de este trabajo es proveer una plataforma que permita modelar y gestionar las buenas prácticas, con una definición semántica detallada de las mismas y de los procedimientos que involucran. A su vez, mediante la recolección de información de las actividades del proyecto, debe detectar la ocurrencia de violaciones a dichas prácticas y facilitar su corrección mediante la distribución del conocimiento de la organización en el momento en que es relevante a las personas involucradas, dando como resultado un proceso de desarrollo más confiable y eficiente.

El aporte de Singapur a la gestión de las mejores prácticas consiste en la generación automática de nueva información y su difusión, a partir de los datos de trabajo que se registran.

Para ello, es fundamental el rol de las tecnologías de la Web Semántica al momento de realizar el modelado de la información, ya que permiten una fácil integración de los datos provenientes de distintas fuentes[10] y realizar inferencia sobre los mismos, posibilitando la generación de nueva información.

Otro de los aspectos importantes de Singapur es su arquitectura, que brinda una flexibilidad que resulta fundamental para poder resolver los problemas que surjen en los proyectos reales. Su diseño permite integrar fácilmente los agentes consumidores y generadores de datos a las herramientas que ya se utilizan. Ésto tiene un gran impacto debido a que en la actualidad existe una gran cantidad de herramientas que realizan las mediciones o las tareas necesarias. En la mayoría de los casos sólo es necesario re-

alizar pequeñas extensiones para mostrar la información de las violaciones detectadas en el contexto de trabajo o convertir los datos generados a una representación semántica y enviarlos al servidor de Singapur.

A modo de verificación se tomaron los datos de un proyecto de desarrollo de software real con el fin de demostrar que Singapur efectivamente puede detectar las violaciones modeladas y reportarlas de manera prácticamente instantánea. Los resultados obtenidos fueron satisfactorios ya que se detectaron y registraron de manera correcta todas las violaciones ocurridas.

10.1. Trabajo futuro

Enfoque de información a distribuir

Singapur detecta las violaciones luego de que se produzcan, en el futuro se puede trabajar sobre un enfoque que distribuya la información necesaria para evitar que se produzcan las violaciones.

Analisis de informacion para generar nuevo conocimiento

El nuevo conocimiento que se genera en Singapur está limitado a las reglas que se creen para detectar las violaciones a las mejores prácticas. Una posibilidad que surge al disponer de los datos semánticos del trabajo realizado en varios proyectos pertenecientes a distintas organizaciones, es analizarlos para generar nuevo conocimiento[14] sobre las cosas que funcionaron bien o mal dentro del proyecto. Para ésto se pueden crear nuevas ontologías que describan los conceptos necesarios para poder evaluar automáticamente los datos recopilados y generar nuevas mejores prácticas.

Validación de reglas de detección de violaciones

Se puede realizar una validacion automática de las nuevas reglas de detección de violaciones en base al juego de datos que se tiene en el modelo semántico, o un análisis semántico teniendo en cuenta si pueden o no producir resultados.

Bibliografía

[1] Grigoris Antoniou and Frank van Harmelen. A Semantic Web Primer, 2009.

[2] Aybuke Aurum, Hå kan Petersson, and Claes Wohlin. State-of-the-art: Software inspections after 25 years. *Software Testing Verification and Reliability*, 12(3):133–154, 2002.

[3] Max Boisot. *Knowledge Assets: Securing Competitive Advantage in the Information Economy*, volume 1. 1998.

[4] John M. Carroll, Hao Jiang, Mary Beth Rosson, Shin-I Shih, Jing Wang, Lu Xiao, and Dejin Zhao. Supporting activity awareness in computer-mediated collaboration. *2011 International Conference on Collaboration Technologies and Systems (CTS)*, pages 1–12, 2011.

[5] Kimiz Dalkir. *Knowledge Management in Theory and Practice*, volume 4. 2011.

[6] T H Davenport, R G Eccles, and L Prusak. Information Politics. *Sloan Management Review*, 34(1):52–65, 1992.

[7] A.J. Davies and A.K. Kochhar. A framework for the selection of best practices, 2000.

[8] Stefan Decker and M.R. Frank. The networked semantic desktop. In *WWW Workshop on Application Design, Development and Implementation Issues in the Semantic Web*, volume 105, pages 1613–0073, 2004.

[9] Marc Demarest. Understanding knowledge management, 1997.

[10] Laura Drăgan, Renaud Delbru, Tudor Groza, Siegfried Handschuh, and Stefan Decker. Linking Semantic Desktop Data to the Web of Data. *The Semantic Web – ISWC 2011 SE - 3*, 7032:33–48, 2011.

[11] Bob DuCharme. *Learning SPARQL*. O'Reilly Media, 2011.

[12] Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware: some issues and experiences, 1991.

[13] Martin Fowler. Continuous Integration, 2006.

[14] Tom Gruber. Collective knowledge systems: Where the Social Web meets the Semantic Web, 2008.

[15] Carl Gutwin and Saul Greenberg. Workspace awareness for groupware. In *Conference companion on Human factors in computing systems common ground - CHI '96*, pages 208–209, 1996.

[16] Tom Heath and Christian Bizer. Linked Data: Evolving the Web into a Global Data Space, 2011.

[17] John Hebeler, Matthew Fisher, Ryan Blace, and Andrew Perez-Lopez. *Semantic Web Programming*, volume 20. 2009.

[18] Y.H. Lee and S.I. Yoo. A Rete-based integration of forward and backward chaining inferences. *Proceedings of Tenth International Symposium on Intelligent Control*, 1995.

[19] E L Lesser, M A Fontaine, and J A Slusher. *Knowledge and Communities*. 2000.

[20] Leo Sauermann, Ansgar Bernardi, and Andreas Dengel. Overview and Outlook on the Semantic Desktop. *Proceedings of the 1st Workshop on The Semantic Desktop at the ISWC 2005 Conference*, 175:1–19, 2005.

[21] Michael Sintek, Siegfried Handschuh, Simon Scerri, and Ludger van Elst. Technologies for the Social Semantic Desktop. In *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689, pages 222–254. 2009.

[22] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H D Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best Practices for Scientific Computing. *PLoS Biology*, 12(1), 2014.

[23] Liyang Yu. *A Developer's Guide to the Semantic Web*. 2011.