ARTÍCULO SELECCIONADO DEL CONAIISI

TracEDaaS: captura y trazabilidad de artefactos del proceso de diseño.

Ing. Federico Hernández[1]
Dra. Ing. Luciana Roldán[2]
Dra. Ing. Marcela Vegetti[3]
Dr. Ing. Silvio Gonnet[4]
Dr. Ing. Horacio Leone[5]

Facultad Regional Santa Fe,
Universidad Tecnológica Nacional.

[1] Ingeniero en Sistemas de Información.
INGAR (UTN-CONICET).
E-mail: fhernandez@santafe-conicet.gov.ar

[2] Doctora en Ingeniería. INGAR(UTN-CONICET)
E-mail:lroldan@santafe-conicet.gov.ar

[3] Doctora en Ingeniería. INGAR(UTN-CONICET)
E-mail: mvegetti@santafe-conicet.gov.ar

[4] Doctor en Ingeniería. INGAR(UTN-CONICET)
E-mail: sgonnet@santafe-conicet.gov.ar

[5] Doctor en Ingeniería. INGAR(UTN-CONICET).
E-mail: hleone@santafe-conicet.gov.ar

RESUMEN

Se presenta una herramienta, TracEDaaS, que permite la captura y la trazabilidad de la evolución de los productos del proceso de diseño de software, soportando el refinamiento y la elaboración iterativa del mismo. TracEDaaS ha sido implementada como un servicio en la nube bajo el modelo "Software as a Service" (SaaS) y permite definir dominios de diseño de software particulares y capturar la evolución de los objetos de diseño de esos dominios. La información capturada está estructurada en término de las operaciones aplicadas a los objetos de diseño y posibilita la trazabilidad de los artefactos del proceso de desarrollo.

ABSTRACT

In this proposal, TracEDaaS, a computational environment to support the capture and tracing of software design process is presented. TracEDaas has been implemented as a services in the cloud (SaaS) and it allows defining a particular software design domain and supporting the capture of how products under development are transformed along a design process. This captured information is structured in terms of operations applied on design objects and it enabling traceability of artifacts of development process.

1. INTRODUCCIÓN

Aportes recientes consideran trazabilidad como un método para seguir el proceso de desarrollo de software [1] [2]. En la última década se realizaron numerosos aportes con este objetivo, la mayoría de las cuales representan las relaciones de traza entre objetos del proceso de desarrollo [3][4][5]. Otros trabajos se concentran en el modelado y captura del razonamiento [6][7], aspecto relevante para poder representar y reconstruir un proceso de desarrollo [8]. Sin embargo, es necesario contar con herramientas que aborden la trazabilidad de forma integrada, permitiendo la representación del proceso de desarrollo junto al conocimiento empleado. Diferentes autores concuerdan en que tales decisiones constituyen conocimiento tácito, el cual a pesar de ser esencial para la solución alcanzada, no es explícitamente documentado, ni forma

parte del artefacto resultante [2][8][9][10]. Esto conduce a que en futuras revisiones o modificaciones del artefacto diseñado sea difícil y costoso establecer correspondencias entre decisiones de diseño y las razones que las respaldaron. En este sentido, Roldán y colab. [11] [12] proponen un modelo conceptual para soportar los diferentes modelos que son generados cuando un producto ingenieril es diseñado. La propuesta soporta la representación explícita de los estados a través de los cuales el modelo del artefacto que está siendo diseñado evoluciona durante su diseño.

En virtud de lo antes expresado, el desafío actual consiste en la definición de una herramienta que facilite la administración de la evolución de los artefactos generados en un proceso de desarrollo de software. La herramienta propuesta deberá brindar soporte en la captura de la evolución de los distintos artefactos, considerando distintos niveles de abstracción, soportando el refinamiento y la elaboración iterativa del mismo. Es vital la captura y representación de la información que permita la trazabilidad de los artefactos del proceso de desarrollo, manteniendo la navegabilidad entre los distintos modelos y las diferentes versiones de los mismos. Por estos motivos, se trabajó en la propuesta de una herramienta, TracEDaaS, que permita la captura y la trazabilidad de la evolución de los productos del proceso de diseño. La herramienta se basa en un modelo genérico para la captura del diseño en término de las operaciones aplicadas a los objetos de diseño [12]. El objetivo principal de la herramienta es capturar las versiones del modelo desarrollado (estados del diseño) durante un proceso de diseño y cómo fueron obtenidos (razonamiento, decisiones, actores, etc.). Este ambiente fue diseñado con el requerimiento de brindar soporte a distintos dominios de diseño. Este enfoque permite su extensión a nuevos problemas de diseño en término de los conceptos particulares del dominio y las posibles operaciones que pueden ser aplicadas a las instancias de tales conceptos.

A continuación, en la Sección 2, se presenta el modelo conceptual de TracEDaaS. Luego, en la Sección 3, se define la arquitectura de TracEDaaS y su implementación en la nuble. Por último, en la Sección 4 se presentan las conclusiones del trabajo.

2. MODELO CONCEPTUAL DE TRACEDAAS

La herramienta propuesta, TracEDaaS, se sustenta en un esquema de administración de versiones que permite capturar y representar la evolución de los distintos productos del diseño en forma conjunta con las operaciones aplicadas a los mismos [11][12]. Para realizar tal captura, TracEDaaS considera el proceso de diseño como una secuencia de actividades que opera sobre los productos del proceso de diseño, denominados objetos de diseño. Los objetos de diseño representan modelos del artefacto que está siendo diseñado, requerimientos a cumplir, restricciones impuestas al modelo. Naturalmente, estos objetos evolucionan durante un proceso de diseño, dando lugar a múltiples versiones de los mismos. Estas versiones pueden ser consideradas como fotos de los objetos de diseño en un determinado instante, y un conjunto de estas versiones constituyen una versión del modelo. Una versión de modelo describe el estado del proceso de diseño en un determinado instante. En este esquema, cada versión de modelo es generada mediante la aplicación de una secuencia de operaciones en una versión de modelo predecesora. La secuencia de operaciones puede incluir la eliminación, creación, y modificación de versiones que forman la versión de modelo predecesora. En consecuencia, se definió en TracEDaaS un conjunto de operaciones básicas (agregar, eliminar, modificar) a través de las cuales se representan las operaciones más complejas (refinar, abstraer, aplicarMVC, etc.) que se ejecutan en las distintas etapas del proceso de desarrollo de los modelos. La definición de estas operaciones facilita el seguimiento de las operaciones realizadas en la generación de las distintas versiones, permitiendo construir el ciclo de vida de cada modelo en particular [12].

TracEDaaS puede ser adaptado a nuevos problemas de diseño. Esta adaptación se realiza en término de los conceptos del método de diseño y del lenguaje de modelado empleado. Además, es posible definir las operaciones que pueden ser aplicadas en un proceso de diseño. En la Fig. 1 se presenta una vista global del ambiente TracEDaaS. Los dos componentes principales de TracEDaaS son el editor de dominios y el administrador de versiones. Mediante el empleo del Editor de Dominios, un experto del Dominio puede definir uno o más dominios de diseños. Esta definición del dominio se realiza en término de los conceptos o tipos de objetos de diseño y las operaciones que son aplicables a los mismos.

Empleando el Administrador de Versiones los diseñadores pueden elegir un dominio existente y ejecutar/capturar las operaciones definidas junto a las versiones de los objetos de diseño generadas.

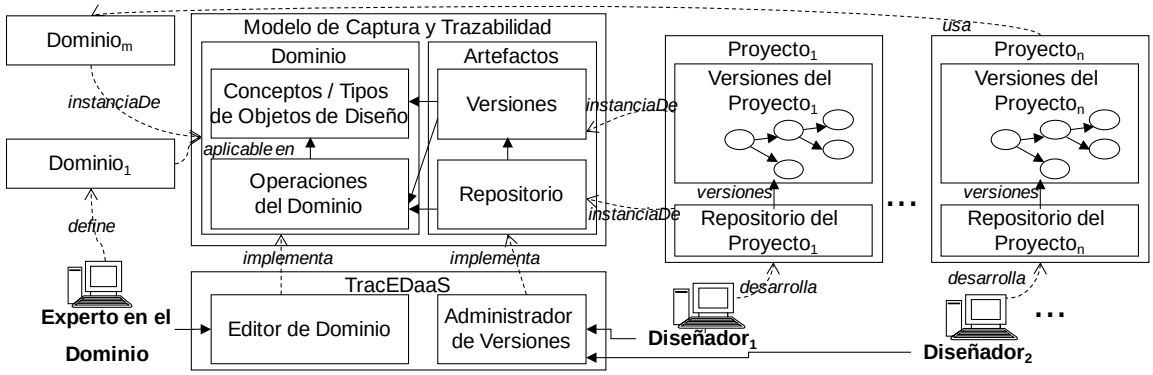

Figura 1. Vista Global de TracEDaaS.

El esquema de representación de versiones divide la representación de los objetos de diseño en dos niveles: i) nivel repositorio (Repositorio en Artefactos, Fig. 1), donde se representa cada objeto de diseño por medio de un *objeto versionable*; ii) nivel versiones (Versiones en Artefactos, Fig. 1), donde se representan los distintos estados (*versión de objeto*) alcanzados por cada objeto de diseño durante un proyecto. En la Fig. 2 se presenta la estructura de árbol que posee el esquema de representación de versiones de modelo, donde a partir de la versión de modelo inicial (m_0 en Fig. 2), pueden obtenerse sucesivas versiones de la misma (m_i y m_k). En la parte inferior de la Fig. 2 se ejemplifica la descripción visual de la arquitectura de software correspondiente a las versiones de modelo m_k y m_q. En este caso, se trata de dos versiones de modelo tomadas de una porción del caso de estudio que se abordará posteriormente, la versión m_q se obtiene aplicando la secuencia de operaciones ϕ_g sobre la versión de modelo m_k. La secuencia ϕ_q incluye una operación de refinamiento (*applyMVC*) que permite transformar al componente *WebApplication* en un conjunto de componentes (*Model*, *View* y *Controller*) y conectores.

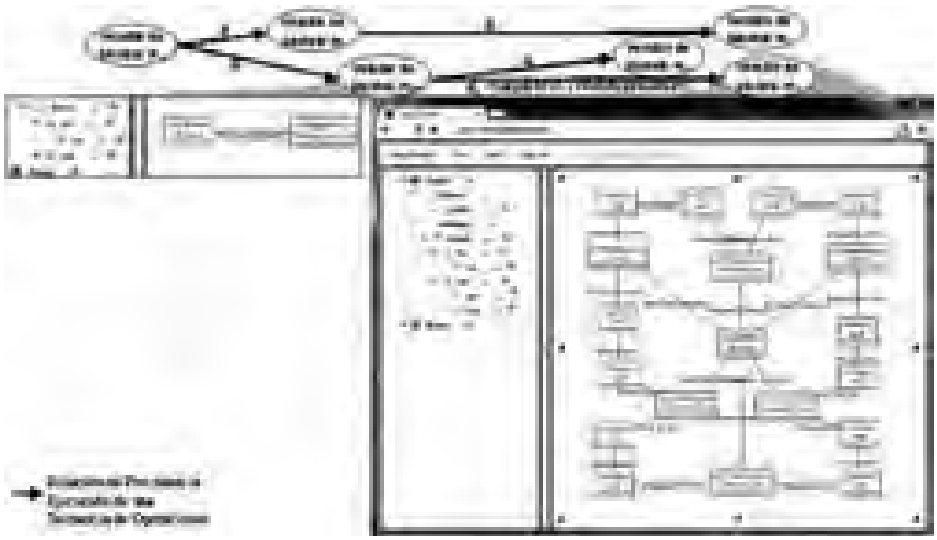

Figura 2. Ejemplo de una Evolución de Versiones de Modelo de una Arquitectura.

3. EDITOR DE DOMINIO

El Editor de Dominio (Fig. 1) permite la definición de un dominio de diseño. El dominio se especifica en términos de los *conceptos* o *tipos de objetos de diseño* que se desean modelar y las *operaciones* que son aplicables a los mismos. Los conceptos están organizados en una estructura de árbol (Concept Tree en Fig. 3). Cada concepto posee un conjunto de *propiedades* o atributos. Cada atributo es un par compuesto por un nombre y un tipo de dato primitivo. En la versión implementada se definieron los tipos de datos primitivos String, Integer, Boolean, y Float. En el ejemplo de la Fig. 3, el concepto *Component* posee dos atributos de tipo String, *description* y *type*.

El Editor de Dominio permite especificar operaciones en los conceptos definidos, las cuales se ejecutarán luego desde el Administrador de Versiones. La Fig. 4 ilustra los argumentos y la definición de la operación *addComponentwPorts*. En este caso, los argumentos de *addComponentwPorts* (Fig. 4 (a)) son: un nombre de componente (*name*), una lista de nombres de puertos (*lPorts*), una lista de nombres de responsabilidades (*lResps*) y una versión de *System* (*vSystem*). Como se ilustra en la Fig. 4 (b), la operación *addComponentwPorts* es definida en término de operaciones auxiliares (*AddRelationship*, *Loop*), y operaciones definidas en el dominio (*addComponent*, *addResponsibility*, *addPort*). En primer lugar se genera una instancia de *Component* (*vComponent*), cuyo nombre es *name*, mediante la operación *addComponent*. Luego, para cada nombre (*portName*) de la lista de puertos (*lPorts[]*) se instancia y agrega un puerto al componente recientemente creado (*addPort*(*portName*,*vComponent*)). En forma similar, se agregan las responsabilidades (*addResponsibility*(*respName*, *vComponent*)). La versión *vComponent* es agregada mediante la ejecución de *addComponent*. La Fig. 4(c) muestra la especificación de *addComponent*, la cual incluye la operación primitiva *add* y la operación auxiliar *AddRelationship*. A partir de simples operaciones es posible definir operaciones de más alto nivel que representan decisiones del proceso de diseño. Por ejemplo, la operación *applyMVC* (Fig. 5) especifica la aplicación del patrón Modelo-Vista-Controlador (MVC).

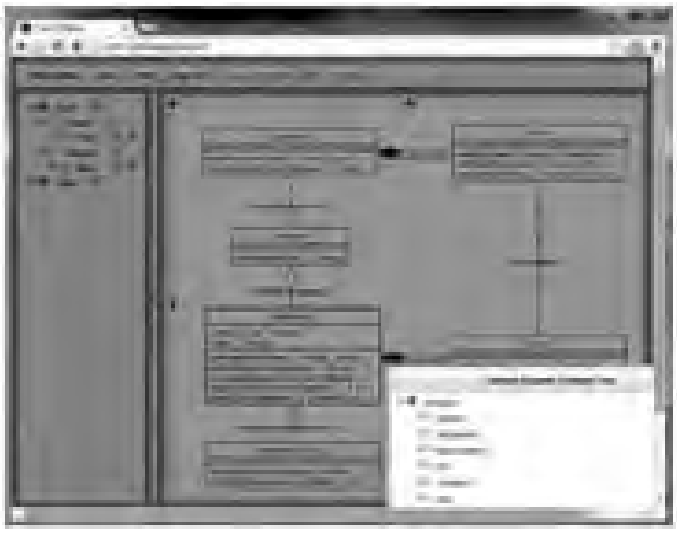

Figura 3. Editor de Dominios, conceptos definidos.

Figura 4. Editor de Dominios, edición de una operación. (a) argumentos. (b) especificación del cuerpo. (c) especificación de addComponent.

En la Fig. 5(a) se incluyen los argumentos de *applyMVC*, como ser la versión del componente a ser refinado (*vComponent*), el nombre de la vista (*view*), el modelo (*model*) y el controlador (*controller*), y el conjunto de responsabilidades que serán delegadas a los distintos elementos (*respsToView*, *respsToModel*, *respsToController* en Fig. 5(a)). En la Fig. 5(b) se muestra el cuerpo de *applyMVC*. La operación es especificada en términos de operaciones previamente definidas en el dominio, tales como *addComponent*, *addPort*, *addConnector*, *delegateResponsibilities*, y de la operación primitiva *delete*.

4. ADMINISTRADOR DE VERSIONES

El Administrador de Versiones permite la ejecución de un proyecto de diseño, capturando las operaciones realizadas y las versiones generadas. Cuando se inicia un nuevo proyecto de diseño, se selecciona un dominio existente. De esta manera, la evolución de un proyecto se basa en la ejecución de las operaciones definidas en el dominio seleccionado y la instanciación de los conceptos de tal dominio. Además, TracEDaaS, mediante su Administrador de Versiones, permite mantener información sobre: i) versiones de modelo predecesora y sucesora de cada versión de modelo; ii) relaciones de historia que trazan las secuencias de operaciones aplicadas y permiten generar una nueva versión de modelo; iii) referencias al conjunto de versiones de objetos resultante de ejecutar las distintas operaciones

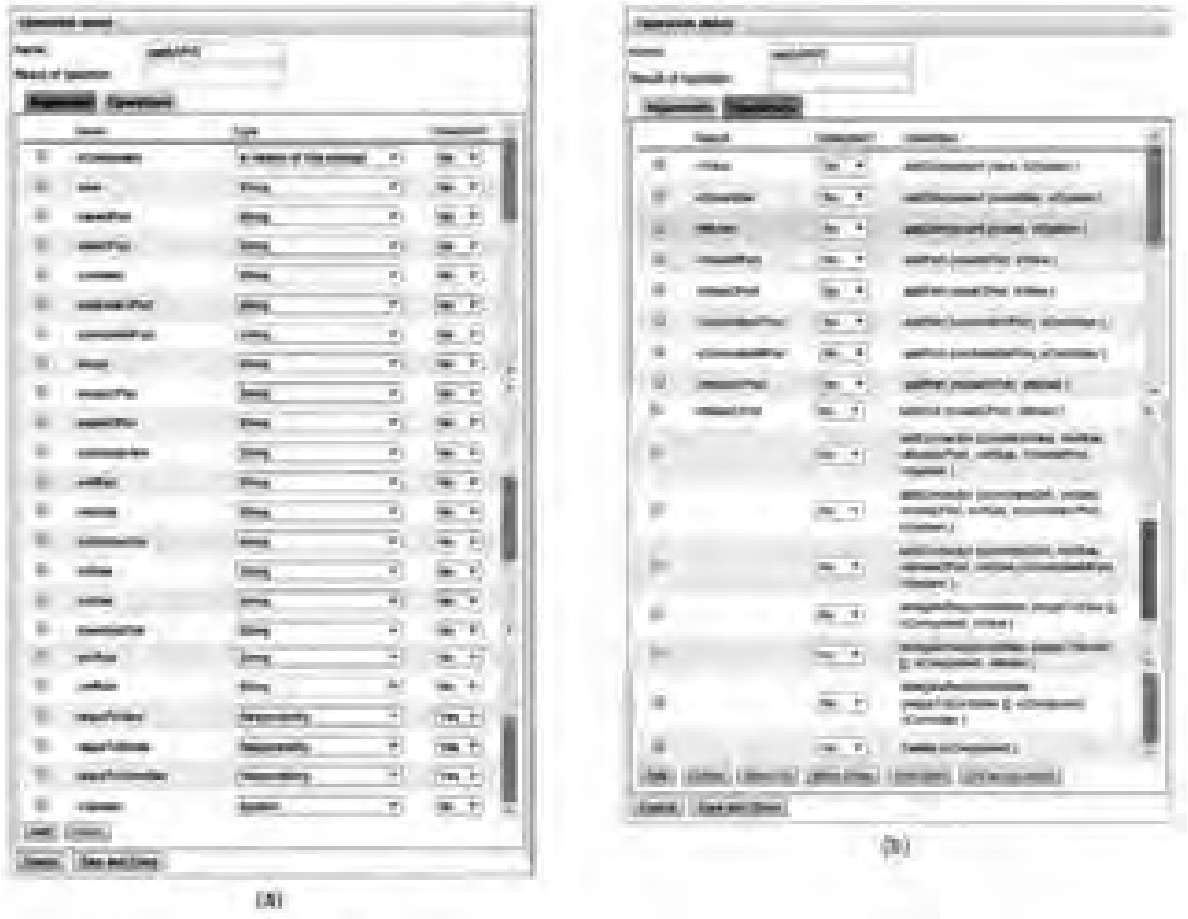

Figura 5. Editor de Dominios, edición de applyMVC. (a) argumentos. (b) cuerpo.

La ejecución de una operación consiste en la ejecución de cada una de las operaciones que la especifican, además, se genera un objeto que mantiene información de la operación ejecutada, enlazando las versiones de objeto predecesoras (argumentos de la operación), con las versiones de sucesoras (resultante de la operación). Esto permite reconstruir la historia de una versión de modelo en término de las operaciones aplicadas.

A continuación, se muestra el funcionamiento del administrador de versiones empleando como ejemplo una porción del desarrollo del proyecto de diseño Struts. Una vez iniciado el proceso de diseño, éste se va desarrollando de la siguiente manera: cuando el diseñador desea crear una nueva versión de modelo, debe seleccionar en el árbol de versiones de modelo la versión con la que trabajará (versión de modelo actual), es decir, aquella versión de modelo sobre la que aplicará una secuencia de operaciones. Realizando un click derecho sobre el diagrama, la herramienta muestra las operaciones disponibles para el dominio actual (Fig. 6), disponiéndolas en sub-menúes que clasifican a las operaciones según los distintos tipos de objeto de diseño del dominio. Por ejemplo, en la Fig. 6 se ilustra la ejecución de la operación *addSystem*(*Struts*) (seleccionando el ítem de menú *addSystem*), la cual agrega la primera versión de

objeto de tipo *System* a la versión de modelo actual. Luego, el actor (arquitecto) genera la siguiente versión de modelo definiendo la estructura de la arquitectura de software. De esta manera, agrega el primer componente a través de la ejecución de una operación *addComponentwPorts*, la cual genera la versión de objeto de tipo *Component* denominada *WebApplication* y las responsabilidades (*Responsibility*) asociadas al mismo. Los objetos resultantes de esta operación son incorporados en la versión de modelo *Struts Version 2* y se visualizan en la parte inferior de la Fig. 6.

Luego, el diseñador considera que es conveniente aplicar el patrón MVC, dado que posibilita la satisfacción de un requerimiento de modificabilidad que fuera inicialmente planteado. El arquitecto lleva a cabo esto mediante la ejecución de la operación *applyMVC* sobre la versión *WebApplication*, especificando los distintos argumentos para la operación (Fig. 7). Como resultado, se obtiene la versión de modelo *Struts Version 3* (Fig. 8), en la cual están presentes tres nuevas versiones de objeto de tipo *Component*: *Model*, *View* y *Controller* y las versiones de los conectores entre tales componentes (*ConnViewCtrlr*, *ConnModCtrlr*, *ConnModView*). Además, la ejecución de la operación *applyMVC* delegó las responsabilidades del componente original (*WebApplication* en Fig. 6) en cada uno de los nuevos componentes y eliminó la versión *WebApplication*. Adicionalmente, la ejecución de la operación almacena la traza entre los argumentos (en este caso *WebApplication*) y los resultados de la operación. Esto permite navegar por las trazas desde las versiones de objeto origen a las versiones de objeto destino. Esta información puede luego ser recuperada mediante la consulta de la historia del proceso realizado. La Fig. 9 muestra la historia de cómo fue alcanzada la versión de modelo *Struts Version 3*. La figura presenta parcialmente tres porciones de la ventana de Historia, la cual informa todas las operaciones que han sido aplicadas para evolucionar desde la versión de modelo inicial a la versión de modelo seleccionada (*Struts Version 3*).

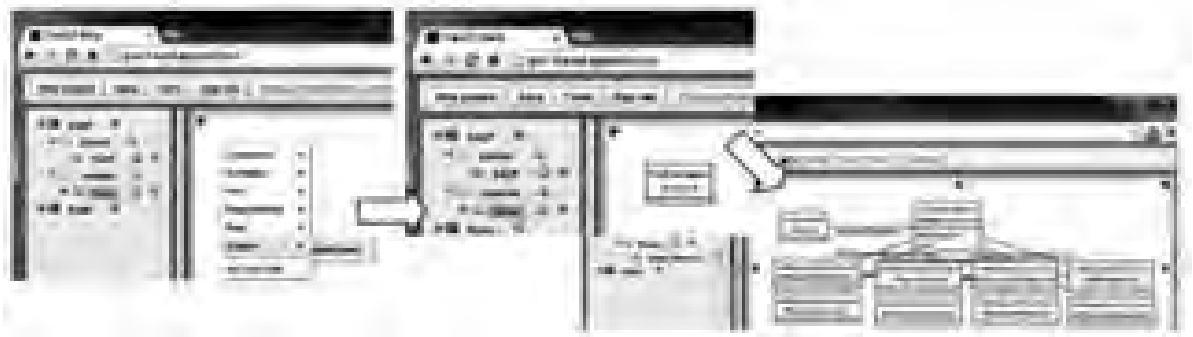

Figura 6. Versión de Modelo Inicial del proyecto Struts, ejecución de addSystem. Versión de Modelo Struts Version 2.

Figura 7. Ejecución de applyMVC.

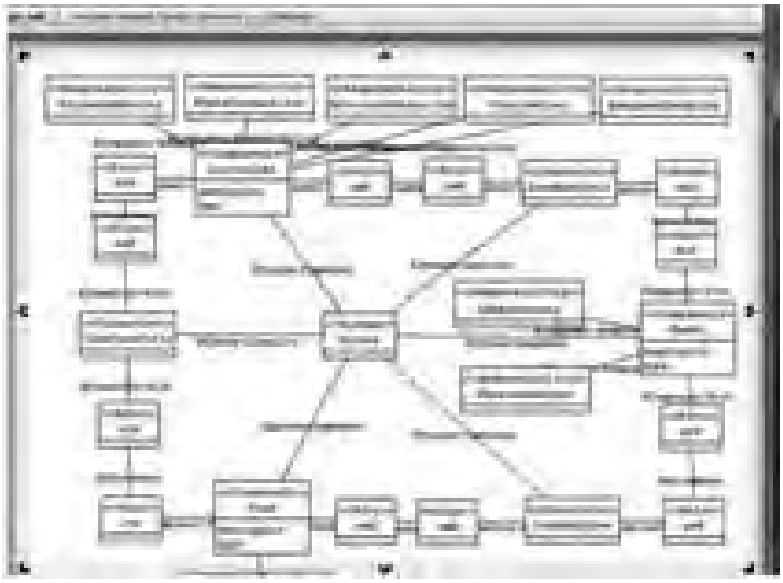

Figura 8. Versión de Modelo Struts Version 3.

5. TRACEDAAS. IMPLEMENTACIÓN EN LA NUBE

Para la implementación de TracEDaaS como un servicio en la web se eligió el lenguaje Java, y particularmente el framework GWT de Google. Actualmente es posible acceder a la aplicación desde http://gwt-traced.appspot.com/. Se seleccionó este framework de desarrollo por las siguientes características que facilitaron la implementación y posterior empleo de TracEDaaS: clientes ricos en el navegador; soporte nativo HTML5; cloud runtime environment; API de seguridad y cloud datastore.

Los clientes ricos en el navegador facilitaron la implementación de las interfaces de usuario del editor de dominios y el administrador de versiones, ya que estas requieren de la edición de diagramas similares a diagramas UML. Además, brindan compatibilidad multi navegador, interfaz optimizada, no se requieren instalaciones de la aplicación, ni bibliotecas adicionales.

Figura 9. Recuperación de la historia de una versión de modelo.

El soporte nativo HTML5 permite realizar gráficos 2D y 3D en el browser de forma nativa, sin necesidad de instalar complementos adicionales.

Cloud runtime environment posibilita la ejecución de la aplicación en el AppEngine de Google y no requiere de la instalación de un servidor propio. Además, esto permitió emplear como usuarios las cuentas de Google con relativa facilidad, lo cual ahorra el desarrollo de una capa de seguridad y gestión de usuarios. La aplicación usa la tecnología de DataStore de Google, esto permite desacoplar la programación de la lógica de la aplicación con el almacenamiento.

La Fig. 10 ilustra la arquitectura de TracEDaaS implementada en la nube empleando el framework GWT de Google. Como se puede observar en la figura, la arquitectura de la aplicación está definida por las capas *UI*, *Services*, y *Data Access*.

La capa de presentación (*UI*) brinda soporte a la realización de los diagramas del editor de dominio y el administrador de versiones. Se diseñó la interfaz de la aplicación con el objetivo de maximizar el área de trabajo y minimizar el área de información y elementos auxiliares. Como resultado de esto, se dispone de 3 áreas bien diferenciadas: menú superior, barra de navegación, y área gráfica de trabajo (ver Fig. 2, versión de modelo m_q).

El menú superior muestra un menú con las acciones que permiten: crear nuevo proyecto (New Project); guardar versión de modelo actual (Save);- mostrar conceptos de un dominio en forma de árbol (Tools -> Show domains concepts tree, como el ilustrado en la Fig. 3); validar las operaciones de un dominio para establecer su consistencia (Tools -> Validate domain operations); clonar un dominio (Tools -> Clone domain); crear una nueva versión de modelo a partir de la versión de modelo actual (Tools -> New model version); ver el historial de cómo se obtuvo una versión de modelo en particular, a partir de la aplicación de una secuencia de operaciones (Tools -> View model history); exportar la versión de modelo actual a un formato imprimible (Tools -> Export to SVG); salir de la aplicación (Sign out). Además se visualiza el estado de la aplicación.

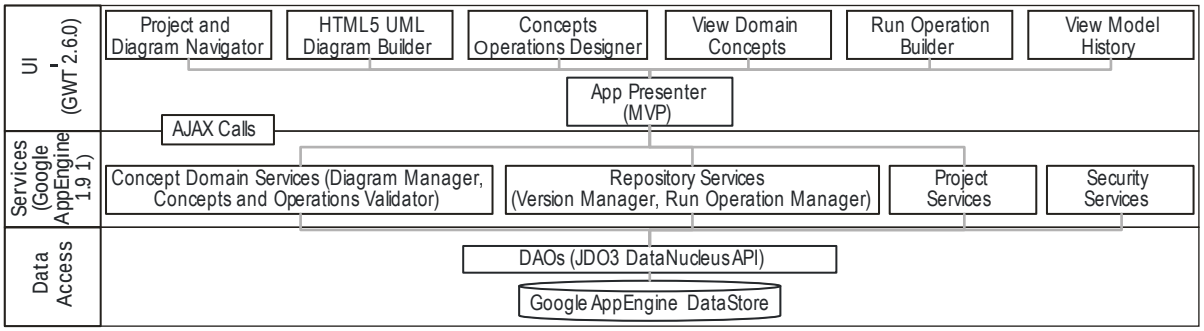

Figura 10. Arquitectura de TracED empleando el framework GWT de Google.

La barra de navegación muestra los proyectos, con su correspondientes Dominios y Versiones de modelos existentes. Los proyectos pueden ser públicos o privados (no compartidos). El área gráfica de trabajo es el área de trabajo principal, donde se editan los diagramas del editor de dominios y el administrador de versiones. Tiene un menú contextual dependiente del lugar donde esté posicionado.

Esta capa fue programada en Java, mediante el framework GWT 2.6.0. Se hace un uso intensivo de la biblioteca de widgets provista por el mismo. Asimismo, para la realización de gráficos, se ha utilizado como base el framework Tatami (https://code.google.com/p/tatami/), que es una implementación en GWT del framework DOJO (http://dojotoolkit.org/), que se utiliza para hacer gráficas nativas HTML5 en la web.

UI está conformada por el componente principal *App Presenter* y seis componentes con responsabilidades específicas: *Project and Diagram Navigator*, *HTML 5 UML Diagram Builder*, *Concepts Operation Designer*, *View Domain Concepts Tree*, *Run Operation Builder*, y *View Model History*. Toda esta capa se compila con el compilador GWT, que transforma el código en javascript optimizado para cada navegador y corre totalmente en la máquina del usuario.

Los componentes de *App Presenter* están programados mediante el patrón Model-View-Presenter, el cual permite distribuir el código entre la vista y la lógica, y ayuda a crear módulos bien

estructurados y testeables. De esta forma, la vista puede ser diferente para cada tipo de dispositivo deseado, pero las llamadas a los servicios se hacen a través del Presenter con llamadas RPC. Se hace uso de GWT EventBus para el manejo de eventos en el front-end. *UI* se comunica con la capa de servicios (*Services*) a través de llamadas asincrónicas (*AJAX*), lo cual permite que la página no se tenga que recargar y genera una mejor experiencia de usuario. Por esta razón, se utilizó el patrón Data Transfer Objects (DTO), minimizando el paso de objetos complejos entre el cliente y el server y simplificando el proceso de serialización de los mismos.

Project and Diagram Navigator permite organizar y gestionar los diagramas en proyectos, permitiendo crear, copiar, mover, borrar y hacerlos públicos y privados. Las versiones de modelos se presentan en una estructura de árbol respetando el enfoque conceptual introducido en la Fig. 2.

HTML5 UML Diagram Builder se encarga de generar y manipular diagramas en forma ágil con tecnología Drag & Drop. Su uso difiere si se utiliza en el Editor de Dominios o en el Administrador de Versiones. En el Editor de Dominio permite definir un dominio, con sus conceptos y relaciones. Los conceptos están representados por un nombre, propiedades y operaciones. Para la definición de las propiedades y el nombre se utiliza un analizador léxico que permite identificar el nombre, el tipo y la presencia de estereotipos.

Concepts Operation Designer es un componente principal del Editor de Dominio, permite la especificación de operaciones. Para la definición del cuerpo de una operación se utiliza una combinación de comandos básicos que se pueden combinar, creando operaciones complejas. En la definición de las operaciones se establece el binding de los argumentos de las mismas, es decir, el elemento del cual va a tomar el valor al momento de correr la operación.

View Domain Concepts Tree permite ver el árbol jerárquico de los conceptos. Útil para ver todos los conceptos estructurados según las relaciones de generalización-especialización.

Run Operation Builder es un componente principal del Administrador de Versiones. Nos permite ir construyendo un modelo de versiones de un dominio en particular, ejecutando las operaciones asociadas al dominio. Se encarga de mostrar las operaciones disponibles y de devolver los resultados para su visualización gráfica en el diagrama.

View Model History permite ver el historial de una versión de modelo en particular.

La capa de servicios (*Services*, Fig. 10) corre en el server y contiene la mayoría de la lógica de la aplicación. *Services* está compuesta de 4 componentes: *Concept Domain Services*, *Repository Services*, *Project Services*, y *Securitiy Services*. Los dos primeros implementan la lógica del editor de dominios y el administrador de versiones, respectivamente.

Concept Domain Services posee dos servicios: *Diagram Manager* y *Concepts and Operations Validator*. El *Diagram Manager* se encarga de la gestión de los diferentes objetos de un dominio, identificando y validando *Conceptos* y *Relaciones*. *Concepts and Operations Validator* posee la responsabilidad de validar las operaciones definidas en los conceptos. La validación se realiza en término de los bindings establecidos y los argumentos requeridos por las operaciones.

Repository Services está compuesto por los servicios *Version Manager* y *Run Operation Manager*. *Version Manager* se encarga de la gestión de versiones de un modelo: crear versión de modelo inicial, crear nuevas versiones de modelos a partir de versiones existentes, mantener el historial de operaciones ejecutadas y las diferentes versiones de modelo. *Run Operation Manager* es el encargado de ejecutar las operaciones de una versión del modelo en particular, generando nuevas versiones de objetos. Este servicio funciona como un intérprete de comandos, el cual interpreta la operación a ejecutar, que siempre es una sucesión de comandos que se ejecutan en forma secuencial. La ejecución de la operación modifica la versión del modelo y va guardando las trazas que permiten seguir la historia del diseño. Como resultado de la ejecución de una operación, devuelve un resultado formado por un log (historia) y un conjunto de componentes para ser ubicados correctamente en el diagrama.

Project Services se encarga de gestionar los proyectos en carpetas agrupadas por dominio y versiones de modelo.

Security Services se encarga de la autenticación de usuario, utilizando el API de seguridad de google, permitiendo autenticar contra una credencial existente de Gmail.

La capa de persistencia está programada siguiendo la especificación Java Data Object (JDO). El API

utilizado para la implementación es DataNucleus JDO3. En tiempo de ejecución el API crea una PersistenceManagerFactory (PMF) que proporciona la conectividad con el DataStore. Como DataStore se utiliza Google AppEngine DataStore, que es una base de datos cloud o en la nube, de tipo NoSQL, que provee acceso robusto y escalable, que permite almacenar los gráficos en forma serializada como BLOBs.

6. CONCLUSIONES

En el trabajo se propone una herramienta en la nube, TracEDaaS, que permite implementar un enfoque operacional para la captura y trazabilidad del proceso de diseño de software. La herramienta incluye un editor de dominio que permite la definición del lenguaje de modelado y las operaciones aplicables a los conceptos identificados. Además, TracEDaaS provee los mecanismos necesarios para llevar a cabo un proceso de diseño mediante la ejecución de una secuencia de operaciones definida en el dominio de trabajo. Esta secuencia de operaciones permite representar las decisiones de diseño realizadas en la evolución de una versión de modelo predecesora a una sucesora.

Cabe destacar que TracEDaaS fue desarrollada con el objetivo de verificar que es posible aplicar el modelo operacional propuesto para la captura y trazabilidad del proceso de diseño, y no intenta remplazar herramientas tradicionales de diseño. TracEDaaS debería ser integrado a herramientas existentes y funcionar en modo background. De esta manera, TracEDaaS capturaría las operaciones realizadas en ambientes de diseño tradicionales.

AGRADECIMIENTOS

Se agradece el apoyo brindado por las instituciones que financiaron este trabajo: CONICET (PIP 112 20110100906), y Universidad Tecnológica Nacional (PID 25/O118 y PID 25/O156).

REFERENCIAS

[1] GOTEL, O., CLELAND-HUANG, J., HAYES, J., ZISMAN, A., EGYED, A., GRÜNBACHER, P., DEKHTYAR, A., ANTONIOL, G., MALETIC, J., MÄDER, P., (2012). *Traceability Fundamentals*. Software and Systems Traceability, J. Cleland-Huang, O. Gotel, A. Zisman (Eds.): 3-22. Springer.

[2] WINKLER, S., VON PILGRIM, J. (2010). *A survey of traceability in requirements engineering and model-driven development*. Software System Model 9: 529-565.

[3] MÄDER, P., GOTEL, O. (2012). *Towards automated traceability maintenance*. Journal of Systems and Software 85(10): 2205-2227.

[4] SCHWARZ, H., EBERT, J., WINTER, A. (2010). *Graph-based traceability: a comprehensive approach*. Software System Model 9: 473-492.

[5] OBJECT MANAGEMENT GROUP (2011). *OMG Unified Modeling Language (OMG UML), Superstructure*, Version 2.4.

[6] CARIGNANO, M.C., GONNET, S., LEONE, H. (2009). *A model to represent architectural design rationale*. Proceedings of Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture. (WICSA/ECSA): 301-304.

[7] DUTOIT, A., MCCALL, R., MISTRIK, I., PAECH, B. (eds.) (2006). *Rationale Management in Software Engineering*. Springer.

[8] RAMESH, B., JARKE, M. (2001). *Toward reference models for requirements traceability*. IEEE Transactions on Software Engineering 27(1): 58-93.

[9] KRUCHTEN, P. (2004). *An Ontology of Architectural Design Decisions*. Proceedings of 2nd Groningen Workshop on Software Variability Management.

[10] CARIGNANO, M.C., GONNET, S., LEONE, H. (2013). *Reasoning and Reuse in Software Architecture Design: Practices in the Argentine Industry*. Electronic Journal of SADIO 12(1).

[11] ROLDÁN, M.L., GONNET, S., LEONE, H. (2010). *TracED: A Tool for Capturing and Tracing Engineering Design Processes*. Advances in Engineering Software 41(9): 1087-1109.

[12] ROLDÁN, M.L., GONNET, S., LEONE, H. (2013). *Knowledge Representation of the Software Architecture Design Process based on Situation Calculus*. Expert Systems 30 (1): 34-53.