ARAT

Architectural Rationale Annotations Tool

Manual de usuario

Escrito por S. Hyun.
Revisado por J. A. Hurtado
25/06/2018

TABLA DE CONTENIDO

INTRODUCCIÓN Y PROPÓSITO

La tecnología hoy en día toma un papel muy importante en la sociedad, la demanda de software cada día es más alta y los sistemas son cada vez más complejos causando que su evolución sea inevitable. Sin embargo, el desarrollo de software es un proceso difícil que involucra a nivel general habilidades técnicas para analizar las reglas de negocio de un dominio en particular, transformar esas reglas en requerimientos para luego implementarlos, probarlas y posteriormente realizarles correcciones a futuro.

El proceso de desarrollo de software se constituye de diferentes actividades que van desde el análisis de requisitos hasta el mantenimiento y la evolución de un sistema [1], generalmente este proceso está acompañado de un conjunto de decisiones que se toman para definir cómo se deben realizar las actividades en cada etapa del ciclo de vida del software. Una de las fases más importantes en el ciclo de vida es el diseño de software, en donde se definen los componentes, las relaciones, las interfaces, los puertos, los nodos y cualquier otra abstracción que representa una parte de la arquitectura del software [2]. En esta fase se toman decisiones importantes sobre el diseño de un sistema, generalmente esas decisiones se definen en diferentes niveles de abstracción, como por ejemplo la elección de ciertos patrones de diseño en el nivel de diseño detallado, o la selección de ciertas tácticas y/o patrones de arquitectura a nivel de diseño arquitectónico. Las decisiones de diseño arquitectónicas generalmente están enfocadas a atender ciertas preocupaciones de calidad [3], frecuentemente a estas necesidades de calidad se les denomina atributos de calidad del producto software, y entre ellos se encuentran el rendimiento, la seguridad, la disponibilidad, la usabilidad, entre otros [4]. Estos atributos de calidad generalmente dirigen el diseño de la arquitectura de cualquier sistema y son los motivos por los cuales se toman ciertas decisiones de diseño de un conjunto de posibles opciones de diseño, con el objetivo de conseguir la mejor combinación posible de decisiones que cumplan con las necesidades de calidad de los stakeholders[1] [5].

El Rationale arquitectónico son las razones del porqué una arquitectura está planteada de una forma o de otra, este Rationale arquitectónico define la justificación o el porqué de una acción o un pensamiento dirigido a la toma de decisiones en la construcción de la arquitectura de un sistema [6]. Generalmente este Rationale no se encuentra de manera explícita en los diferentes artefactos de la documentación de un sistema, causando que sea mucho más lenta la fase de mantenimiento del mismo, debido a que se requiere de más tiempo para tratar de capturar y analizar las decisiones tomadas en el diseño del software.

[1] Stakeholder: persona, grupo u organización que está activamente involucrada en el desarrollo de un proyecto, el cual se ve afectado en forma negativa o positivamente de acuerdo a los resultados del mismo

Este documento pretende servir como un manual de uso sugerido para la utilización del plugin ARAT (Architectural Rationale Annotations Tool); una herramienta complemento basada en anotaciones de código fuente en Java que permite documentar el Rationale arquitectónico desde el código fuente. Este documento se constituye de 6 secciones posteriores a esta introducción, las cuales se mencionan a continuación: Sección de fundamentos, donde se menciona la importancia, las ventajas, las desventajas los límites y los casos de uso. Sección de instalación, en la cual se muestra como descargar, compilar y ejecutar el plugin. En la sección de especificación se define el modelo del Rationale que se define, los atributos y la documentación de la herramienta relacionada. En la sección de uso sugerido se encuentran todas las sugerencias de uso respecto a la convención y nomenclatura en algunos de los atributos, la ubicación de la librería y las posibles actividades. Finalmente, en las últimas dos secciones se habla sobre el proyecto, se brinda la información de contacto y se muestran las referencias consultadas.

FUNDAMENTOS

Con respecto a las anotaciones de código, son metadatos que complementan el código fuente enriqueciendo el software con funcionalidades encapsuladas en componentes aislados de la lógica y las reglas del negocio de un sistema [7]. Este lenguaje de anotaciones se crea en la solicitud de especificación java 175 (JSR-175) [8] y buscaba reemplazar las diversas formas de asociación de información a componentes de software como clases, métodos, atributos, etc. En la actualidad, el uso de las anotaciones de código se ha extendido de una manera creciente, permitiendo encapsular procedimientos de creación y validación, relacionados con la información de los metadatos en los componentes de software marcados con anotaciones. Después de la JSR-175 surgen dos JSR (305 y 308 [9], [10]) las cuales buscan que las anotaciones se puedan aplicar a los programas de Java en más objetivos de anotación, con el propósito de servir de apoyo a las herramientas que detectan los defectos del software.

Por otra parte, frecuentemente en las metodologías de desarrollo tienden a documentar cómo un sistema va a cumplir o cumple con los requerimientos funcionales y/o no funcionales. Sin embargo, no es muy usual que se realicen esfuerzos adecuados para documentar las razones por las cuales el sistema debe comportarse de una forma o de otra. Estas razones de diseño arquitecturales se convierten en metadatos que complementan el desarrollo de un sistema, estos frecuentemente se encuentran como conocimiento tácito o implícito en el desarrollo y gestionarlo es un proceso que requiere de mucho tiempo y esfuerzo.

IMPORTANCIA, VENTAJAS Y DESVENTAJAS

Las anotaciones de código fuente sirven como un puente entre el desarrollador y las diferentes herramientas y librerías que son capaces de reconocerlas, gestionarlas y brindar una mayor cantidad de servicios a través de los metadatos marcados con información relevante para el desarrollo. El uso de estas anotaciones es importante ya que tienen como objetivo construir sistemas más complejos, que permitan encapsular lógica de programación repetitiva en componentes aparte de las reglas de negocio de un sistema en particular, precisamente una de las mayores ventajas de las anotaciones de código fuente es permitir al desarrollador crear diferentes elementos lógicos encargados de capturar y procesar dichas anotaciones con los metadatos de los componentes marcados, con el objetivo de delegarle a las anotaciones de código la responsabilidad de realizar procesos comunes y repetitivos en toda la aplicación [11]. Otra ventaja de las anotaciones de código es la capacidad informativa a nivel declarativo, ya que permiten una mejor comunicación entre el desarrollador, el cual crea sus propias anotaciones definiendo el comportamiento esperado; y el compilador, el cual es el encargado de entender esas anotaciones y procesarlas. Además, la declaración de anotaciones de código fuente mejora la legibilidad del código escrito, facilitando de alguna forma el entendimiento del código a personas ajenas a su construcción y/o mantenimiento. Una de las desventajas más notables es el cambio de paradigma requerido para programar este tipo de lenguaje de marcado ya que las anotaciones de código fuente son una especificación del paradigma orientado a atributos [12]. Este paradigma no es muy común y requiere de capacitación adicional para comprenderlo e implementarlo, además la documentación sobre este paradigma es escasa y el soporte en la comunidad de desarrollo es menor en relación con otras especificaciones del lenguaje de programación Java. Otra desventaja relacionada al cambio de paradigma es la falta de documentación relacionada al modelado de los diferentes tipos de anotaciones creadas por los desarrolladores,

LÍMITES Y CASOS DE USO

Las anotaciones de código se pueden definir mediante meta anotaciones, las cuales especifican el comportamiento de la anotación en el momento de su declaración. Si se requiere que el compilador realice algún tipo de validación en tiempo de compilación sobre un componente marcado con una anotación, es necesario crear un procesador que se encargue de capturar las acciones relacionadas con la declaración de dicha anotación. Las anotaciones por si solas representan únicamente información declarativa sobre el código fuente, si se requiere de funcionalidad adicional de creación, validación de archivos o clases en tiempo de compilación y/o ejecución, es necesario establecer mecanismos de reflexión

que permitan capturar y gestionar un determinado tipo de anotación, con el objetivo de realizar diferentes acciones con los metadatos marcados en las anotaciones de los diferentes componentes.
De los casos de uso más frecuentes podemos observar el framework de pruebas unitarias JUnit [13], el cual permite establecer un conjunto de pruebas a métodos marcados con la anotación @Test, este framework captura por reflexión todos los componentes marcados con la anotación, carga los valores esperados mediante atributos de la anotación y los compara con los resultados de la ejecución de los métodos marcados. Todo este proceso es transparente para el desarrollador, ya que se encuentra encapsulado en módulos independientes que se basan en la información proporcionada en las anotaciones, Otro caso de uso muy popular es el framework de mapeo objeto relacional Hibernate [14], el cual permite declarar anotaciones que realizan definiciones y validaciones de persistencia como @Id, @Entity, @Column, etc; estas anotaciones se encargan respectivamente de definir un determinado atributo como un identificador en la base de datos, convertir una determinada clase en una entidad perteneciente a una tabla en una base de datos o de validar si un atributo puede ser nulo, único, o estar dentro de un rango específico, entre otras muchas funcionalidades de persistencia. El framework de desarrollo Java Server Faces (JSF) utiliza un tipo de anotaciones de código fuente para JavaEE, con el objetivo de especificar el comportamiento de los Beans empresariales mediante anotaciones como @ManageBean, @SessionScoped, @RequestScoped, etc. Otro framework de desarrollo que hace uso de anotaciones de código fuente para desarrollo web es Spring, el cual por medio de anotaciones como @Component, @Controller, @Service, etc; que definen los componentes, controladores y servicios Spring necesarios para el funcionamiento de una aplicación web con Spring.

INSTALACIÓN

En esta sección se indican los pasos para descargar el código fuente y el .jar con la librería, la cual viene acompañada de este manual, con el objetivo de brindar toda la información posible relacionada con esta herramienta. Seguido de esto se explica cómo agregar la librería a un proyecto en Java y finalmente se muestra cómo realizar la generación del reporte con los componentes marcados con información del Rationale.

DESCARGA DE LA HERRAMIENTA

En el siguiente link encontrará información relacionada con el proyecto, el código fuente, algunas características de la herramienta, el manual de usuario en cuestión y la herramienta en formato .jar.

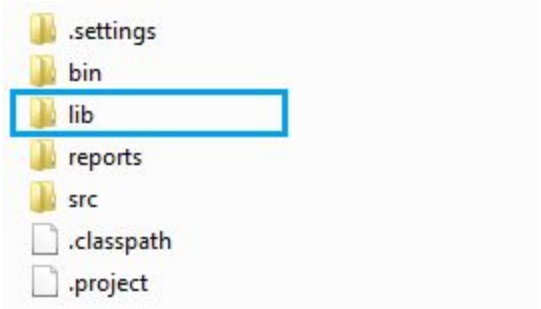
Descargue el plugin .jar y guárdelo en un lugar en el que recuerde su ubicación.

AGREGACIÓN DE LA LIBRERÍA

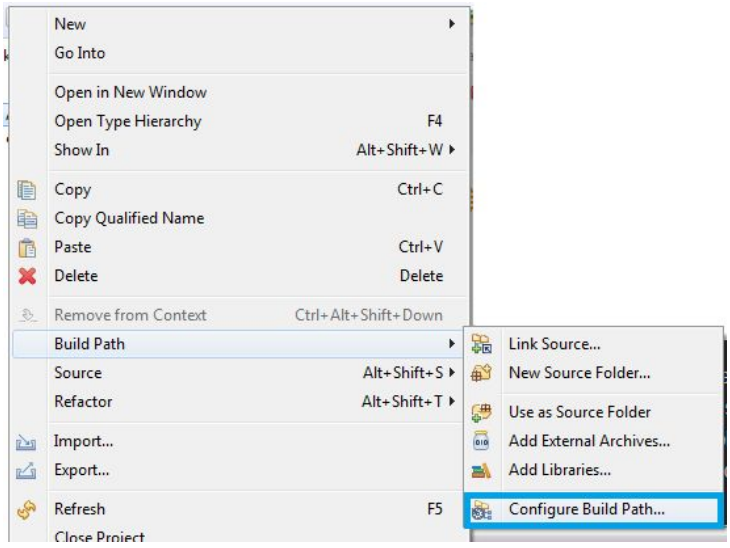
Para agregar la librería a un proyecto es necesario crear una carpeta interna en un proyecto, en la cual se contienen todos los .jar que necesita la aplicación. A continuación, se muestra el proceso para agregar la librería en los IDE más conocidos, Eclipse y NetBeans:

Proyecto Java en Eclipse:

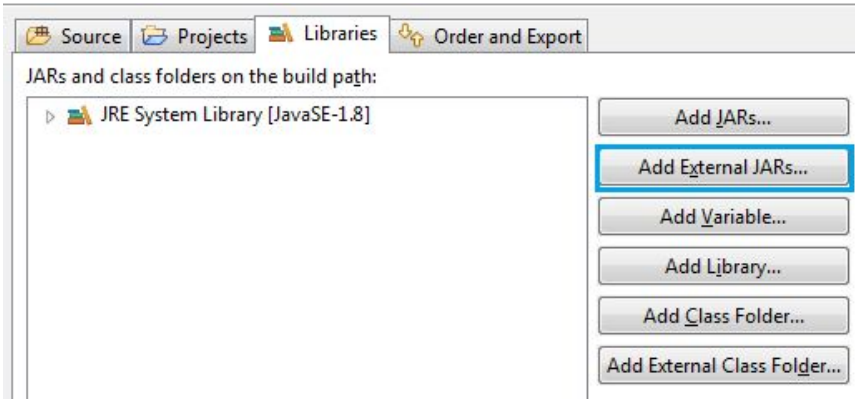
- Paso 1: Dentro del directorio del proyecto, crear la carpeta 'lib' en donde se guarda la librería .jar descargada.



- Paso 2: Seleccionar el proyecto al que se quiere agregar la librería, presionar click derecho y configurar el Build Path.

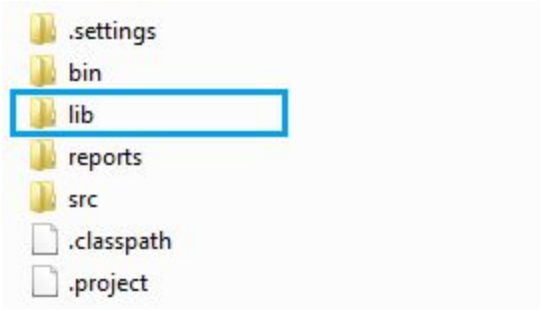


- Paso 3: Presionar añadir .jar externas, seleccionar el .jar guardado en la carpeta lib, aplicar los cambios y cerrar.

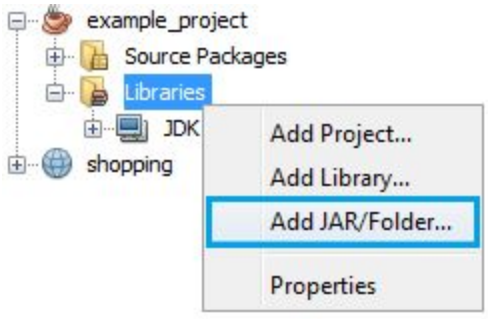


Proyecto Java en NetBeans:

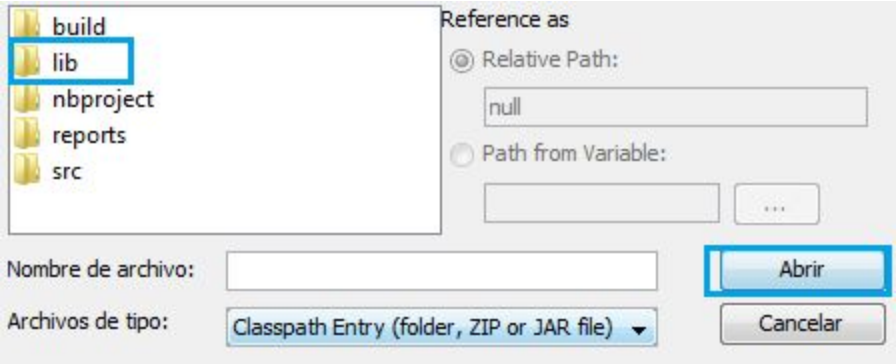
- Paso 1: Dentro del directorio del proyecto, crear la carpeta 'lib' en donde se guarda la librería .jar descargada.



- Paso 2: En la sección 'Libraries' presione click derecho y seleccione 'Add JAR/Folder'



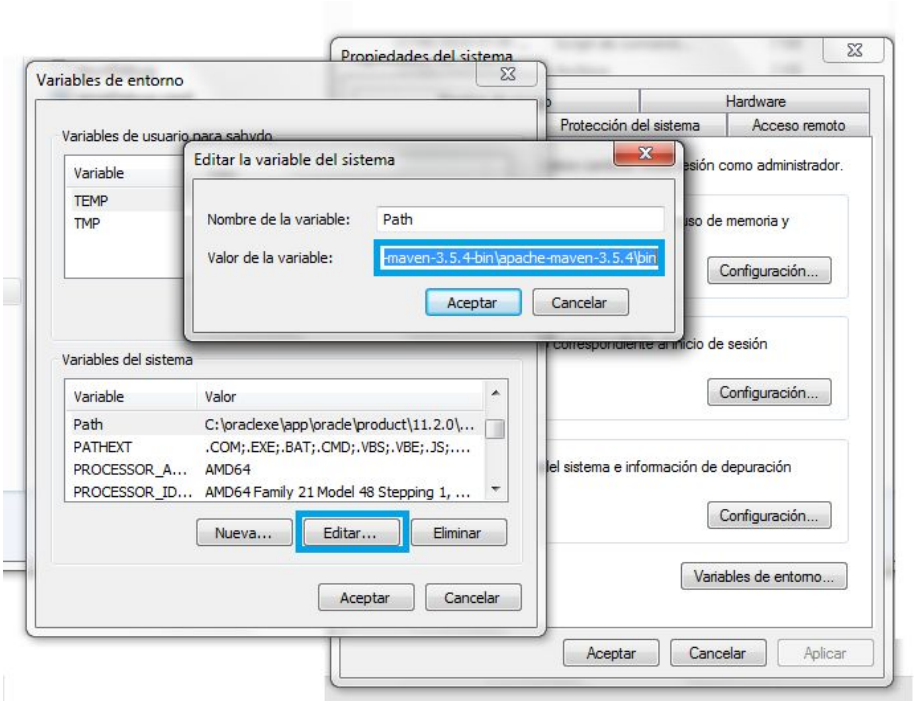
- Paso 3: Busque la carpeta 'lib' donde guardó la librería, seleccione la librería y presione abrir.



Proyecto Maven:

En Eclipse IDE se realiza de igual forma para un proyecto Java normal como para un proyecto Maven, a diferencia de NetBeans IDE, en el cual se hace necesario instalar Apache Maven para después agregar la librería por medio del archivo POM. A continuación, se explica el proceso de instalación para la herramienta en un proyecto maven en NetBeans:

- Paso 1: Descargar Apache Maven
- Paso 2: Extraer los archivos e Instalar Apache Maven en el directorio de preferencia
- Paso 3: Agregar el PATH de la carpeta bin a las variables de entorno del sistema



- Paso 4: Instalar la herramienta a través de Apache Maven de acuerdo a la definición de la dependencia en el siguiente apartado

- Paso 5: Agregar la dependencia en el POM del proyecto:

```
<dependency>
        <groupId>com.arat</groupId>
        <artifactId>arat</artifactId>
        <version>1</version>
</dependency>
```

EJECUCIÓN Y GENERACIÓN DEL REPORTE

La herramienta contiene una clase principal denominada *RationaleFacade* encargada de proveer dos métodos estáticos para generar los reportes:

1. *generateReportByAll* - Genera el reporte de las anotaciones en un solo archivo denominado "*Architectural Rationale Report.pdf*" en una carpeta dentro del proyecto denominada *"reports"*
2. *generateReportsByOne* - Genera un reporte por cada anotación marcada en el código fuente dentro de una carpeta con el nombre de la fecha actual. Cada reporte se nombra de acuerdo al id que se anota en el código, el tipo y el nombre del componente marcado.

Ambos métodos reciben como parámetro el nombre del paquete raíz del proyecto en el cual se quiere realizar la búsqueda de las anotaciones marcadas en el código fuente. En la Figura 1 se puede ver un ejemplo de cómo se hace el llamado a los métodos.

```
public static void main(String[] args) {
    // Para generar el reporte de todas las anotaciones
    // en un solo archivo .pdf
    RationaleFacade.generateReportByAll("com");
    // Para generar el reporte .pdf por cada una de
    // las anotaciones.
    RationaleFacade.generateReportsByOne("com");
}
```

Figura 1

ESPECIFICACIÓN

En esta sección se especifican los atributos del modelo que se define en este trabajo como Rationale arquitectónico. Para la definición de este modelo se tuvieron en cuenta artículos y libros de investigación relacionados a la gestión del Rationale. De igual forma se consultaron artículos y herramientas similares, las cuales brindan una idea general de cómo se utilizan las anotaciones de código fuente para encapsular funcionamiento adicional en

un sistema software.

MODELO DE RATIONALE ARQUITECTÓNICO

En la Figura 2 se puede observar el modelo de Rationale Arquitectónico definido en este trabajo:

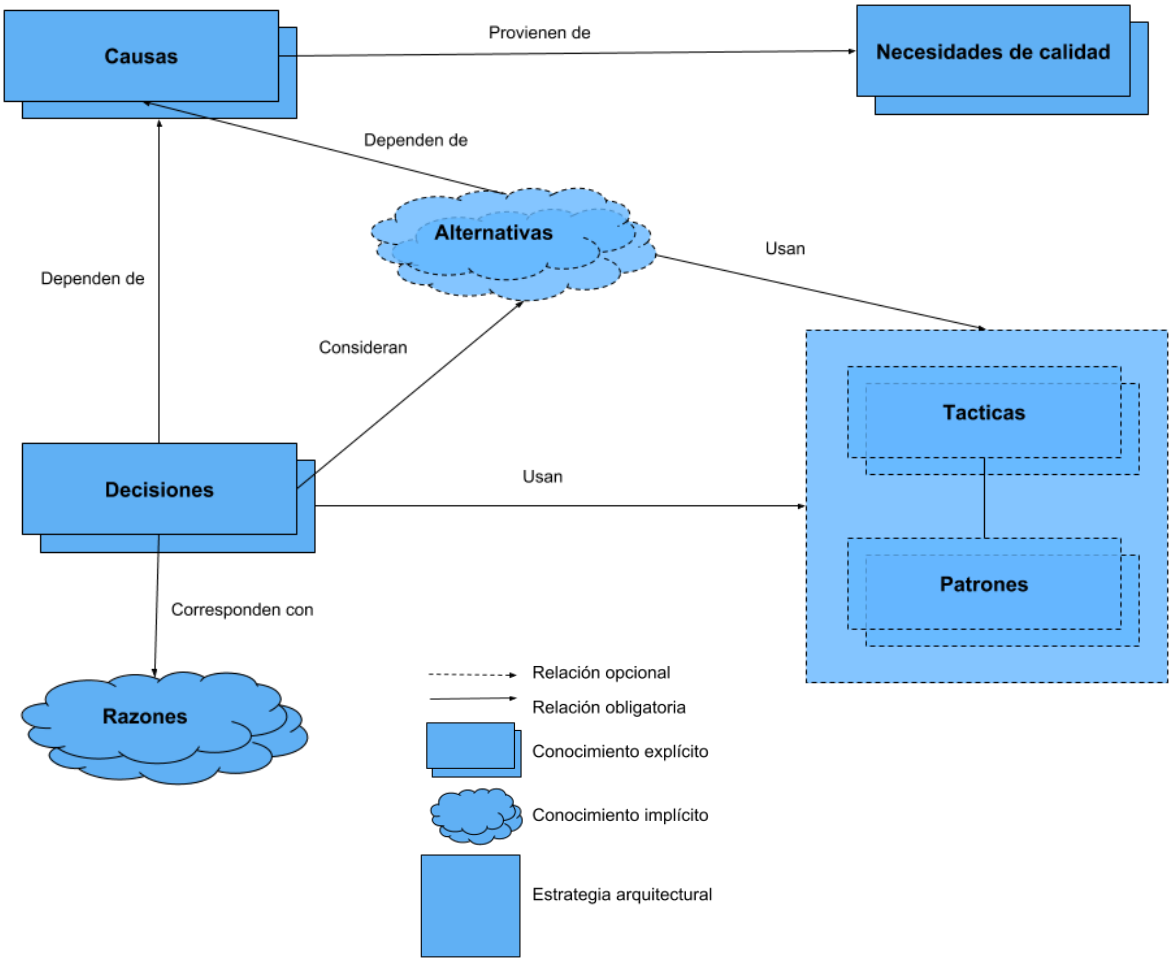


Figura 2

DEFINICIÓN DE LOS ATRIBUTOS MIEMBRO

De acuerdo al modelo presentado anteriormente se plantea una anotación denominada @Rationale, la cual encapsula la información presentada en la imagen anterior. Este modelo de anotaciones consiste en un conjunto de atributos clasificados según su funcionalidad: Configuración e información. A continuación, se especifica cada uno de los atributos de la

anotación. Nótese que la información se compone de un grupo de listas, a diferencia de los atributos de configuración.

1. Atributos de configuración **(opcionales)**:
    - **id** : String que permite identificar una anotación.
    - **hiden**: Boolean que permite ocultar la información de una anotación en la generación del reporte. Por defecto viene con valor *false*
    - **links**: Lista de enlaces a otros documentos
2. Atributos de información:
    - **quality_atributes**: Lista de atributos de calidad que quieren considerar.
    - **causes**: Lista de causas por las cuales es necesario cumplir con los atributos de calidad.
    - **tactics (opcional/Recomendado)**: Lista de tácticas arquitecturales planteadas para lograr la consecución de los atributos de calidad.
    - **patterns (opcional/Recomendado)**: Lista de patrones que complementan las tácticas arquitecturales.
    - **alternatives (opcional/Recomendado)**: Lista de alternativas que se consideran en el momento de tomar decisiones.
    - **decisions_record**: Lista de decisiones que se toman en un determinado momento, para cumplir con un grupo de atributos de calidad específicos.
    - **reasons**: Lista de razones que justifican o expresan el porqué de un grupo de decisiones.

En la Figura 3 se puede observar la definición del Rationale Arquitectónico a través de los atributos en una anotación de código fuente;

```
@Documented
@Retention(RUNTIME)
@Target({METHOD, PACKAGE, TYPE})
public @interface Rationale {
    enum QualityAtribute {
        FUNCTIONAL_ADECUATION, PERFORMANCE, COMPATIBILITY, USABILITY,
        RELIABILITY, SECURITY, MAINTENANCE, PORTABILITY
    }
    String id() default "";
    boolean hiden() default false;
    String[] links() default {};
    QualityAtribute[] quality_attributes();
    String[] causes();
    String[] tactics() default {};
    String[] patterns() default {};
    String[] alternatives() default {};
    String[] decisions_record();
    String[] reasons();
}
```

Figura 3

La anotación @Documented permite agregar el tipo de anotación a la documentación de javadoc, la anotación @Retention indica cómo se va a gestionar el archivo de clases, para este caso se necesita leerlo en tiempo de ejecución a través de reflexión. Finalmente mediante @Target le indicamos al procesador que componentes se pueden marcar con esta anotación, para este caso se pueden marcar componentes a nivel de métodos, clases y paquetes.

CASOS DE USO DE LA HERRAMIENTA

Las funcionalidades y los roles se definen en la Figura 4

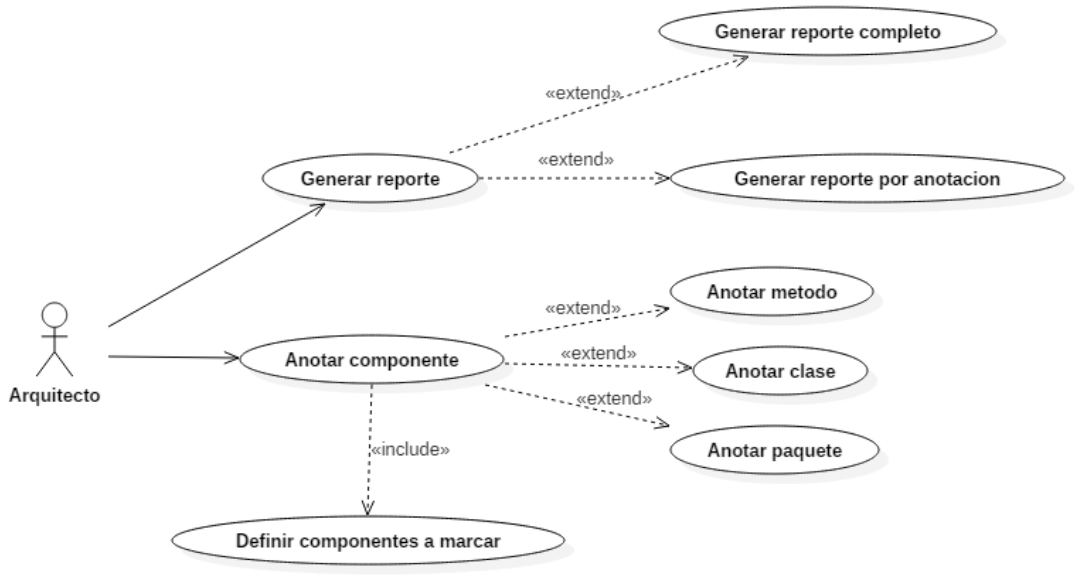


Figura 4

USO SUGERIDO

En esta sección se brindan sugerencias básicas respecto al uso de la anotación @Rationale, con el fin de maximizar la utilidad de la herramienta *ARAT*. Estas recomendaciones comienzan desde las características que debe tener un proyecto en el que se quiera integrar la herramienta hasta el uso detallado y sistemático de algunos atributos miembro de la anotación.

CARACTERÍSTICAS DEL PROYECTO

Esta herramienta se diseñó para ser utilizada en los siguientes tipos de aplicaciones:

1. Java SE
2. Java EE

ACTIVIDADES SUGERIDAS

El proceso de documentación del Rationale Arquitectónico requiere de actividades que fortalezcan el registro de conocimiento implícito en las decisiones de diseño. A continuación, se muestra una lista de actividades sugeridas para documentar el Rationale Arquitectónico.

1. **Definir los atributos de calidad:** En esta actividad se encuentran y se definen las necesidades de calidad que se desean cumplir y que son inherentes a las decisiones de arquitectura que se toman dentro de un proyecto. Para esta actividad es útil hacer uso de escenarios que clarifiquen los requerimientos no funcionales en un sistema.

   Ej: Es necesario que el sistema brinde tiempos de respuesta y procesamiento menores a 1 seg, cuando se realizan peticiones con un gran volumen de usuarios.

2. **Establecer las causas**: Además de saber las necesidades puntuales de calidad, también es necesario establecer por qué es importante cumplir con estas necesidades, cuáles son las ventajas que puede brindar atender estas preocupaciones y cuáles son las desventajas.

   Ej: El sistema es un aplicativo web de comercio electrónico muy concurrido en el cual se registran más de 1000 peticiones por minuto. Cada petición al servidor representa una posible compra, es por esto que el sistema debe responder de la manera adecuada y permitir al usuario terminar con su transacción de manera satisfactoria y en el menor tiempo posible.

3. **Buscar alternativas**: Cuando existen diferentes maneras de llegar a un mismo resultado, se deben esclarecer cuáles son los pro y los contra de tomar alguna decisión que nos lleve por un determinado camino. Generalmente una decisión está acompañada de alternativas que se consideran en un proceso de selección de opciones, estas alternativas se deben registrar en caso de que las características de un sistema varíen y las decisiones actuales ya no sean una buena opción.

   Ej: Existen diferentes alternativas para dar solución a esta necesidad de calidad, entre ellas están:

   - Poner un servidor espejo el cual me responda a las peticiones en caso de que

el servidor principal colapse por la cantidad de solicitudes de usuarios.
- Sacar los componentes que se ven más afectados por el volumen de petición de los usuarios y definirlos en un servicio aparte.
- Introducir métodos que permitan la concurrencia a través de balanceo de carga
- Aumentar la capacidad de recursos físicos

4. **Definir una estrategia arquitectural**: En muchas ocasiones un problema software específico se soluciona con base en tácticas y patrones de arquitectura, los cuales definen entidades precisas que se encargan de resolver un problema recurrente. Esta selección de tácticas y patrones se les considera una estrategia arquitectural.

   Ej: Para separar las responsabilidades y hacer un despliegue independiente de componentes se hace uso del patrón arquitectural multinivel.

5. **Documentar las decisiones**: Las decisiones que se toman sobre el estado del diseño de un sistema generalmente se evaporan con el tiempo, por esto es importante registrarlas y actualizarlas, con el objetivo de mantener actualizado el conocimiento implícito involucrado en el proyecto.

   Ej: Se realizan las consideraciones teniendo en cuenta las ventajas y desventajas de cada alternativa, a lo cual se llega que es mejor separar los componentes más afectados en un componente aparte.

6. **Registrar las razones de las decisiones**: Todas las decisiones de diseño con un impacto arquitectural que se establecen sobre el proceso de desarrollo están fundamentadas en argumentos que justifican y soportan las decisiones tomadas. Estas razones son importantes para entender porqué se decide tomar una decisión y no cualquier otra alternativa dentro de un conjunto de opciones posibles.

   Ej: Poner un servidor espejo no sería de utilidad ya que en caso de que el servidor espejo falle se perderían las peticiones del usuario al sistema. Respecto a los métodos de balanceo de carga y aumento de los recursos físicos no son posibles debido a que no se cuenta con más recursos físicos disponibles.

Finalmente. una de las actividades más relevantes en este proceso es la selección de los componentes que se requieren marcar, estos componentes deben tener relevancia arquitectónica y congruencia con la información que se pretende agregar. La anotación debe estar de manera similar a la Figura 5, una vez marcado el código fuente con la información correspondiente al Rationale arquitectónico.

```
@Rationale(
        id = "1",
        hiden = false,
        quality_attributes = Rationale.QualityAtribute.PERFORMANCE,
        causes = {"El sistema es un aplicativo web de comercio electrónico muy concurrido "
                + "en el cual se registran más de 1000 peticiones por minuto.",
                  "Cada petición al servidor representa una posible compra, es por esto que "
                + "el sistema debe responder de la manera adecuada y permitir al usuario "
                + "terminar con su transacción de manera satisfactoria y en el menor tiempo posible"},
        alternatives = {"Poner un servidor espejo el cual me responda a las peticiones en caso de "
                        + "que el servidor principal colapse por la cantidad de solicitudes de usuarios.",
                        "Sacar los componentes que se ven más afectados por el volumen de petición de "
                        + "los usuarios y definirlos en un servicio aparte",
                        "Introducir métodos que permitan la concurrencia a través de balanceo de carga.",
                        "Aumentar la capacidad de recursos físicos"},
        patterns = {"Ptrón arquitectural Multinivel"},
        tactics = {"Separación de responsabilidades","abstracción de servicios comunes"},
        decisions_record = {"Se realizan las consideraciones teniendo en cuenta las ventajas y "
                        + "desventajas de cada alternativa, a lo cual se llega que es mejor separar "
                        + "los componentes más afectados en un componente aparte."},
        reasons = {"Poner un servidor espejo no sería de utilidad ya que en caso de que el servidor "
                + "espejo falle se perderían las peticiones del usuario al sistema.",
                    "Respecto a los métodos de balanceo de carga y aumento de los recursos físicos "
                + "no son posibles debido a que no se cuenta con más recursos físicos disponibles."}
)
```

Figura 5

NOMENCLATURA Y JERARQUÍA

En muchas ocasiones se pueden presentar casos en los que una decisión arquitectural tenga impacto en diferentes niveles de abstracción, por ejemplo, cuando se establecen restricciones arquitecturales a nivel de métodos, clases, paquetes, componentes, nodos, entre otros. La jerarquía de las anotaciones de código @Rationale se permite mediante uno de los atributos denominado ***id***, el cual es una cadena de texto que establece un identificador único para cada anotación marcada en alguno de los componentes permitidos.

Se sugiere que los identificadores sigan una enumeración en forma de listas concatenadas, como se muestra a continuación en la Figura 6:

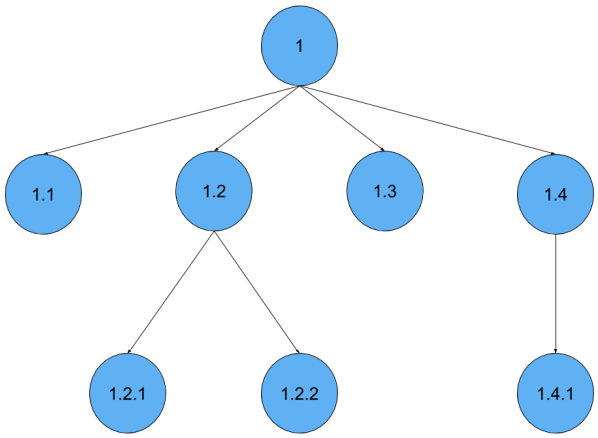


Figura 6

Cada número 'x.y.z' representa información del Rationale arquitectónico marcado a través de una anotación directamente en el código fuente. El número 'x' es el de mayor abstracción ya que permite agrupar un conjunto de decisiones arquitecturales junto con su Rationale. 'y' nos indica cómo los componentes marcados se relacionan con las decisiones de arquitecturales, finalmente 'z' nos permite llegar a un nivel de detalle en el cual se requiere documentar funcionalidad específica con un impacto arquitectural dentro de un componente marcado.

Ej:

1. Se decide utilizar Modelo-Vista-Controlador para separar responsabilidades en la interacción de la GUI con el usuario...
   1.1. El Modelo se diseña al estilo de API para aumentar la escalabilidad del sistema...
      1.1.1. Esta interfaz se encarga de brindar los servicios de la API, para ...
   1.2. El controlador realiza llamados mediante los métodos HTTP para ...

SOBRE EL PROYECTO

Este documento es un artefacto generado en la investigación del uso de las anotaciones de código como una herramienta para gestionar conocimiento tácito directamente en el código fuente, estableciendo un modelo de Rationale Arquitectónico que permita el registro y la gestión de decisiones y razones arquitectónicas, importantes para la evolución y el correcto mantenimiento de cualquier sistema.

La documentación completa de este proyecto y la investigación se encuentra en el siguiente repositorio de github. En este repositorio también se encuentra el link a un sitio web en el cual se puede descargar la herramienta.

Para las personas interesadas en continuar con el proyecto o realizar contribuciones pueden solicitar acceso al repositorio contactando al autor en el siguiente link.

LICENCIA DE USO

Uso permitido:

- Uso comercial
- Modificación
- Distribución
- Uso de patente
- Uso privado

Limitaciones:

- Uso de marca registrada
- Responsabilidad
- Garantía

Condiciones:

- Licencia y aviso de copyright
- Cambios de estado

INFORMACIÓN DE CONTACTO

Autor: Santiago Hyun Dorado

Correo electrónico: santiagodorado@unicauca.edu.co

Sitio web: http://artemisa.unicauca.edu.co/~santiagodorado/

Teléfono (celular): +57 3176349898

REFERENCIAS

[1] J. Highsmith and A. Cockburn, "Agile software development: The business of innovation," Computer (Long. Beach. Calif)., vol. 34, no. 9, pp. 120–122, 2001.

[2] P. Kruchten, H. Obbink, and J. Stafford, "The Past, Present, and Future for Software Architecture," IEEE Softw., vol. 23, no. 2, pp. 22–30, 2006.

[3] D. Garlan and M. Shaw, "An Introduction to Software Architecture," Knowl. Creat. Diffus. Util., vol. 1, no. January, pp. 1–40, 1994.

[4] I. Sommerville, Software Engineering, 9th ed. Boston, Massachusetts: Addison Wesley, 2010.

[5] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, Second. Addison Wesley, 2003.

[6] M. L. Roldan, S. Gonnet, and H. Leone, "Operation-based approach for documenting software architecture knowledge," Expert Syst., vol. 33, no. 4, pp. 313–348, 2016.

[7] C. Lin, "An Annotation Language for Optimizing Software Libraries *," pp. 39–52.

[8] A. Buckley, "JSR 175," JSR 175: A Metadata Facility for the JavaTM Programming Language, 2004. [Online]. Available: https://jcp.org/en/jsr/detail?id=175.

[9] M. Pugh, William Ernst, "JSR 305," 2006. [Online]. Available: https://jcp.org/en/jsr/detail?id=305.

[10] A. Buckley and M. Ernst, "JSR 308," 2014. [Online]. Available: https://jcp.org/en/jsr/detail?id=308.

[11] M. Nosáľ, M. Sulír, and J. Juhár, "Source Code Annotations as Formal Languages," Comput. Sci. Inf. Syst. (FedCSIS), 2015 Fed. Conf., vol. 5, no. 1, pp. 953–964, 2015.

[12] C. Noguera and R. Pawlak, "AVal: An extensible attribute-oriented programming validator for Java," J. Softw. Maint. Evol., vol. 19, no. 4, pp. 253–275, 2007.

[13] T. Ju. Team, "JUnit," The new major version of the programmer-friendly testing framework for Java 8 and beyond. [Online]. Available: https://junit.org/junit5/.

[14] Redhat, "Hibernate," MORE THAN AN ORM, DISCOVER THE HIBERNATE GALAXY. [Online]. Available: http://hibernate.org/.