E.T.S. Ingeniería Geodésica, Cartográfica y Topográfica

Máster en Ingeniería en Geomática y Geoinformación

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA GEODÉSICA CARTOGRÁFICA Y TOPOGRÁFICA

Trabajo Fin de Máster

Nuevas reglas para el marco topológico de gvSIG Desktop.

Alumno: **José Olivas Carriquí**

Tutor: Don Ángel Marqués Mateu

Titulación: Máster en Ingeniería en Geomática y Geoinformación

Centro: E.T.S.I. Geodésica, Cartográfica y Topográfica
Universitat Politècnica de València

Diciembre, 2020

TRABAJO FIN DE MÁSTER

Nuevas reglas para el marco topológico de gvSIG Desktop

ALUMNO

José Olivas Carriquí

Graduado en Ingeniería en Geomática y Topografía

Colegiado nº 8022

Agradecimientos

Con estas páginas quiero agradecer la confianza depositada en mí por parte de Ángel y todo el equipo de la Asociación gvSIG.

También me gustaría dedicar este trabajo a mi familia y a un servidor.

Compromiso

"El presente documento ha sido realizado completamente por el firmante, no ha sido entregado como otro trabajo académico previo y todo el material tomado de otras fuentes ha sido convenientemente entrecomillado y citado su origen en el texto, así como referenciado en la bibliografía"

Índice

Índice de Figuras

Índice de Tablas

1. Introducción

En pleno siglo XXI, y particularmente en este año 2020, ha quedado demostrada la necesidad de sistemas de gestión de información digital. La cantidad y variedad de dicha información se ha acrecentado con el devenir de los años influyendo en todos los aspectos de nuestra sociedad. Un claro ejemplo de esa imparable evolución hacia el "byte" se lleva trabajando y explotando en el ámbito de la ingeniería geomática desde hace ya algún tiempo, siendo el presente proyecto un ejemplo de las capacidades que poseen los técnicos en este campo en el tratamiento y gestión de la geoinformación.

Entrando más de lleno en el trabajo realizado, éste se centra en la topología, una rama de las matemáticas relacionada o ligada con la geometría y las propiedades presentes en determinados elementos situados o localizados sobre un lugar o espacio. Esta es sin duda una de las temáticas más interesantes para realizar un trabajo final para este Máster en Ingeniería Geomática y Geoinformación.

2. Objetivo

El objetivo específico del presente Trabajo Fin de Máster en Ingeniería en Geomática y Geoinformación es la mejora del marco topológico del software libre gvSIG Desktop dentro del proyecto Google Summer of Code 2020 (GSOC 2020). La descripción del proyecto se encuentra en el Apartado ***4. GSOC***.

La mejora consiste en la definición, implementación y testeo de cinco reglas topológicas, así como la elaboración de documentación, plan de pruebas y acciones correctoras para dichas reglas. Las reglas topológicas tratadas en el documento se listan a continuación:

- *Debe estar correctamente dentro de polígonos (Must be properly inside polygons)*, regla topológica para geometrías de tipo punto.
- *Contiene puntos (Contains point)*, una regla para geometrías de tipo polígono.
- *Contiene un punto (Contains one point)*, regla topológica para geometrías poligonales.
- *Deben cubrirse unos polígonos a otros (Must cover each other),* una regla destinada a geometrías de tipo polígono.
- *Regla definida por el usuario (UDR o User defined rule).* Esta es una regla especial o comodín que permite la definición de cualquier regla ya sea para tratar geometrías o datas alfanuméricos siempre y cuando el usuario conozca la expresión de avaluación.

Una circunstancia importante de este proyecto es que las dos primeras reglas citadas ya se encontraban desarrolladas pero debido a múltiples errores en su implementación se optó por su desarrollo desde cero como el resto. El proceso de creación de estas y las demás se cita con mayor grado de detalle en el Apartado ***5. Reglas***.

3. Topología

Tal y como se comenta en el Apartado **1. Introducción** la topología es una rama de las matemáticas relacionada o ligada con la geometría y las propiedades geométricas presentes en determinados elementos situados o localizados sobre un lugar o espacio. De lo anterior parece más que evidente su marcado carácter espacial por lo cual los sistemas de información geográfica son la herramienta perfecta para su implantación y desarrollo, pero antes de la existencia de éstos, la topología ya se explotaba en la resolución de problemas desde hace varios siglos.

3.1. Origen

Uno de los problemas y posterior resolución que da origen según multitud de estudios tanto a la teoría de grafos como a la topología, con permiso de los estudios griegos, data del año 1735, y se denomina el problema de los puentes de Königsberg, actual Kaliningrado. Dicho problema fue resuelto por el matemático y físico Leonhard Paul Euler y a modo de breve resumen busca calcular una ruta por los puentes que comunican un par de islas de dicha ciudad, intentando pasar por todos, pero sin transcurrir por el mismo 2 veces.

La Figura 1 muestra la distribución espacial de las islas y los puentes en aquella fecha y la discretización que realizó Euler para su resolución.

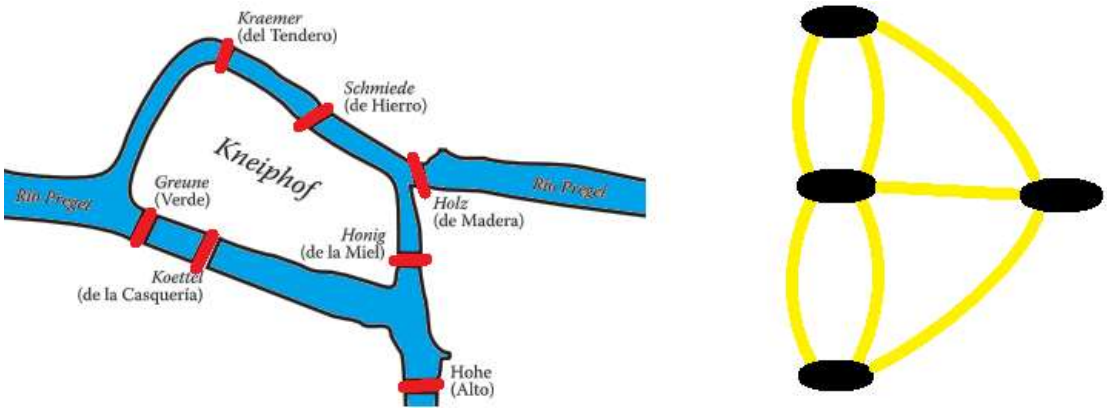


Figura 1. Izquierda; Distribución de las islas y los puentes de Königsberg. Derecha; Discretización del problema ofrecida por L.P.Euler.

La Figura 1 izquierda muestra el rio, polígonos de color azul, puentes polígonos de color rojo y ciudad resto de polígonos, mientras que la imagen de la derecha muestra la ciudad como vértices o nodos, elipses de color negro, y los puentes como aristas o arcos, segmentos de color amarillo.

Para la resolución del problema L.P. Euler se centró en las aristas que entran y salen de cada vértice de modo que se podrá trazar una ruta por las aristas sin repetir estas siempre que a todos los vértices llegue un número par de aristas o que en la distribución sea cual sea solo tenga dos vértices con un número de aristas impar y la ruta inicie en uno de esos dos puntos. De lo anterior y tras observar el ejemplo con el que se realizó en el estudio no era posible pasar por los siete puentes sin repetir uno.

Más recientemente, y como otro ejemplo, la Oficina del Censo de los Estados Unidos, mientras se preparaba para el censo de 1970, fue pionera en la aplicación de la topología matemática a los mapas para reducir los errores al tabular cantidades masivas de datos del censo.

Estos dos ejemplos históricos permiten afirmar la importancia en la resolución de determinados problemas espaciales de la topología, rama que se centra en la posición de los objetos en detrimento de otras características geométricas como sus medidas o dimensiones.

3.2. Topología en gvSIG Desktop

En el caso particular del uso en los sistemas de información geográfica, la topología se centra en definir relaciones espaciales entre objetos, presenten dichos objetos una estructura de datos topológica o no topológica, en cuyo caso se denominan estructuras de datos cartográficas. De modo que cuanto mejor sea la definición de esas relaciones espaciales más riguroso será el software en procesos de filtrado y limpieza de datos de entrada, que junto a tareas de análisis son el principal objetivo de la topología o geometría de la posición en los GIS.

El sistema de información geográfica gvSIG Desktop desarrollado inicialmente por la Generalitat Valenciana y mantenido actualmente por la Asociación gvSIG presenta una implementación propia en cuestiones de tratamiento de geometrías y topología, usando la librería Java Topology Suite (JTS) para determinadas funcionalidades.

A continuación, y desde un punto de vista usuario se detalla el proceso de aplicación de reglas topológicas en dicho software. La definición y esquema de datos de estas se verán con más detalle en el Apartado ***5. Reglas***.

Antes, hay que destacar que la topología en gvSIG Desktop no se encuentra en la instalación base del programa, sino que es un plugin, un paquete o complemento. Para obtenerla hay que descargarla en el *Administrador de complementos* de la pestaña *Herramientas* de dicho software.

La estructura o flujo de trabajo de gvSIG desktop se basa en el siguiente esquema de la Figura 2.

Figura 2. Esquema de funcionamiento de la topología en gvSIG Desktop.

A partir de una fuente de datos (shapefiles, ficheros *CSV*, ficheros procedentes de una base de datos, etcétera) y con una colección de reglas topológicas se crea un Plan topológico mediante la configuración de las siguientes ventanas (ver Figuras 3, 4 y 5):

- Ventana *'Crear plan'*
- Ventana *'Añadir fuente de datos'*
- Ventana *'Añadir regla'*

Tras la ejecución del plan se genera una *Ventana de resultados y reportes de errores* la cual proporciona los resultados o errores y las geometrías que no cumplen la regla. En esta ventana además se aplican las acciones correctoras si el usuario lo desea, siendo estas acciones algoritmos que crean, eliminan o modifican elementos para solucionar los problemas detectados en el proceso de evaluación de la regla.

Como anotación hay que decir que el software presenta el plugin de topología en la pestaña *Herramientas*, y en este complemento solo hay un conjunto de reglas preestablecidas, si se busca aumentar esa colección, las nuevas reglas se tiene que descargar al igual que el plugin gracias al *Administrador de complementos* de la pestaña *Herramientas*.

3.2.1. Ventana *'Crear plan'*

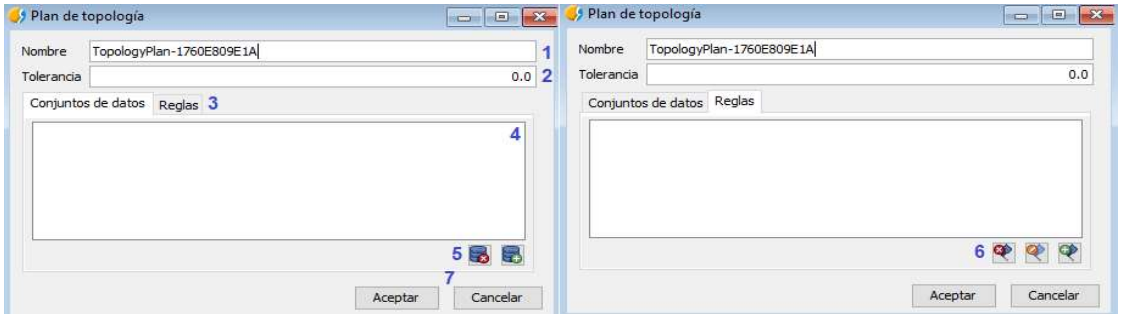


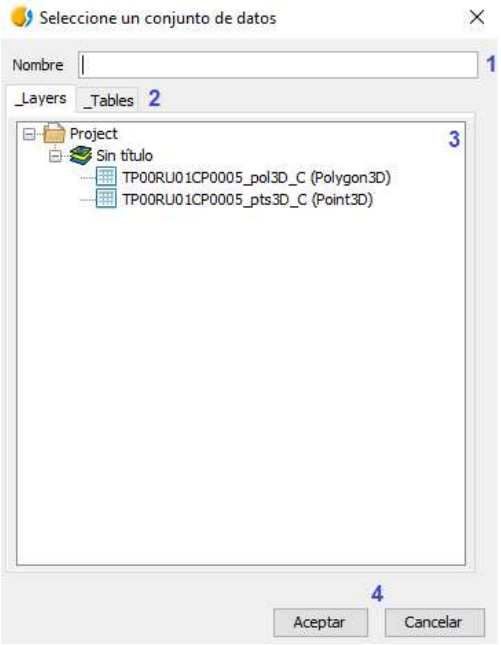
Figura 3. Izquierda, Ventana 'Crear plan' versión Conjunto de datos; Derecha, ventana 'Crear plan' versión Regla topológica.

Esta es la principal ventana en la creación del plan como su propio nombre indica y da inicio a éste (Figura 3). Tiene dos opciones a intercambiar por el componente número 3, en la opción de *Conjunto de datos* se introducen los datasets o fuentes de datos a evaluar, mientras que en la opción *Reglas* se configura la regla topológica a utilizar en el plan.

Los componentes de la ventana que se muestra en la Figura 3 se listan a continuación;

1. Campo Nombre. Renombra el Plan topológico.
2. Campo Tolerancia. Permite especificar una tolerancia en el plan topológico.
3. Pestaña Conjunto de datos/Reglas. Zona de intercambio entre la opción *Conjunto de datos* y la opción *Reglas*.
4. Área donde se muestran los datos introducidos en las diferentes opciones de la ventana.
5. Botones de gestión de la opción *Conjunto de datos*, el botón derecho permite la inclusión de información y su funcionamiento se detallará en el siguiente apartado del documento. El botón izquierdo elimina del plan la información introducida por el botón adyacente.
6. Botones de gestión de la opción *Reglas*, el botón derecho permite la inclusión de reglas topológicas y su funcionamiento se detalla en el Apartado ***3.2.3 Ventana 'Añadir regla'***. El botón central permite la edición y eliminación de las reglas configuradas en el plan respectivamente.
7. Botones de gestión de la ventana.

3.2.2. Ventana *'Añadir fuente de datos'*



Como se indica en el componente número 5 del apartado anterior esta ventana (Figura 4) se muestra tras pulsar el botón de añadir *Conjunto de datos* de la opción Conjunto de datos y al igual que la anterior tiene dos opciones según el tipo de dataset.

Figura 4. Ventana 'Añadir fuente de datos'

Sus componentes son los siguientes;

1. Campo Nombre. Campo que sirve para renombrar datos en plan de topología.
2. Pestaña Capas/Tablas. Zona de intercambio entre la opción *Layers* o *capas de datos* y la opción *Tablas*.
3. Área de visualización de Capas/Tablas. (Árbol del proyecto)
4. Botones de gestión de la ventana.

Como se comenta en la lista el componente número 1 es una entrada de texto que permite nombrar en el plan las fuentes de datos. Renombrar los datos es útil en topología si tenemos un plan con varias reglas el cual hay que pasar por diferentes capas. Gracias a esta herramienta solo hay que poner el mismo nombre a las diferentes capas y ejecutar el plan. De esta forma se optimiza el proceso ya que no es necesaria la creación de una regla por capa.

El componente numero 3 muestra todas las capas disponibles del proyecto o todas las tablas disponibles del proyecto y además estos elementos muestran el tipo de geometría que son.

3.2.3. Ventana *'Añadir regla'*

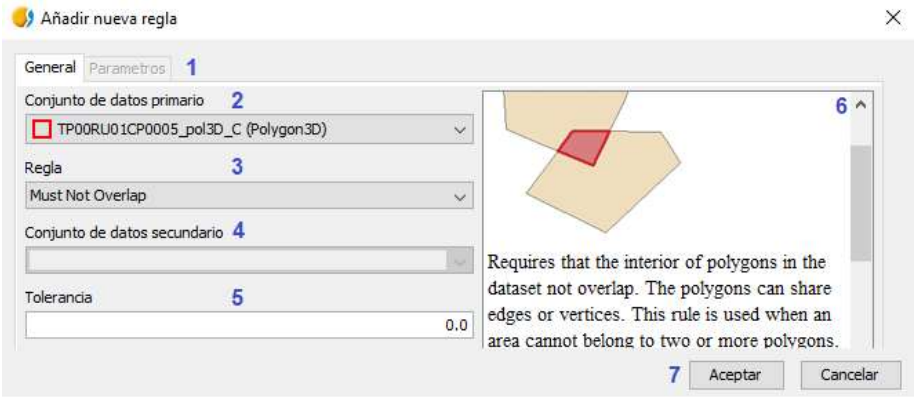


Figura 5. Ventana 'Añadir regla'

Esta última ventana en la creación de un Plan Topológico de gvSIG Desktop (Figura 5) es sin duda la más importante ya que en ella se identifica los datos a evaluar, la regla en cuestión, su tolerancia y los parámetros de esta si fueran necesarios dando fin al proceso. Dichos parámetros se introducirán si la regla seleccionada los necesita, habilitando la pestaña parámetros del componente número 1.

Los componentes de la ventana *'Añadir regla'* se listan a continuación:

1. Pestaña General/Parámetros. Como se indica en el párrafo anterior, esta pestaña se habilita si la regla seleccionada en el componente 3 necesita parámetros extra.
2. Desplegable del Conjunto de datos primario. En este desplegable se muestran los datos introducidos en la pestaña *'Añadir fuentes de datos'* y el conjunto seleccionado es el dataset principal sobre el que se evalúa la regla.
3. Desplegable Regla. En este componente se listan las reglas disponibles en el software. Es importante destacar que solo aparecerán en el desplegable las reglas que actúen sobre el tipo de dato del conjunto de datos primario.
4. Desplegable de Conjunto de datos secundario. Este desplegable se comporta igual que el componente 2 con la salvedad de que si la regla no necesita dataset secundario este componente aparece deshabilitado.
5. Campo Tolerancia. Ventana que permite la intrusión de un valor de tolerancia para determinadas reglas.
6. Área con descripción literal y gráfica de la regla seleccionada en el componente 3.
7. Botones de gestión e la ventana.

3.2.4. Ventana de resultados y reportes de errores

Tal y coma se detalla anteriormente una vez creado el Plan de Topología y tras su ejecución se genera una ventana con los resultados, dicha ventana se muestra en la *Figura 6. Ventana de resultados y reportes de errores*.

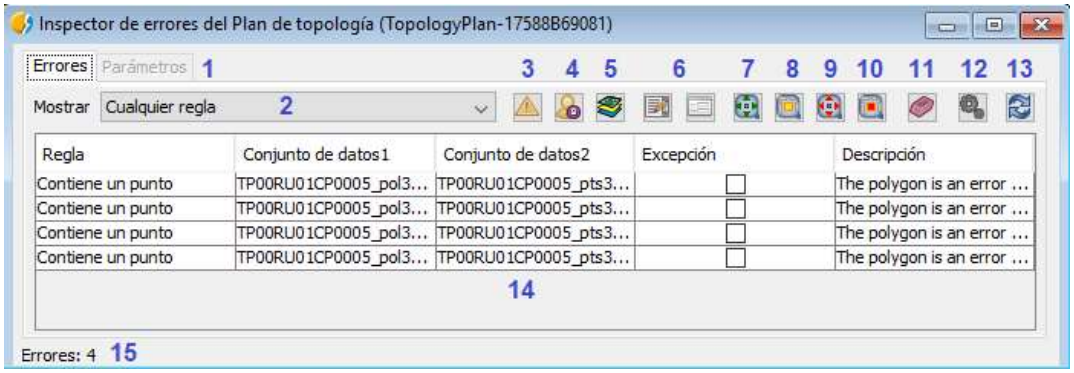


Figura 6. Ventana de resultados y reporte de errores llamada 'Inspector de errores del plan de topología'

Esta ventana no solo nos permite la visualización de los errores o casos que no cumplen la regla, sino que también integra las acciones correctoras. Dichos artificios ofrecen al usuario la posibilidad de rectificar esos errores, y dependiendo de cada regla estas van desde la creación, edición y eliminación de elementos.

Los componentes de esta interfaz se detallan en la siguiente lista:

1. Pestaña Errores/Parámetros. Esta pestaña habilitada por defecto en la opción *Errores* muestra los casos que no cumplen la regla mientras que la opción *Parámetros* solo se habilita si la acción correctora seleccionada para la resolución de errores la necesita.
2. Desplegable de filtro por reglas. Permite un filtrado de los errores según la regla topológica que incumplan.
3. Botón *Mostrar errores*. Funciona a modo de interruptor; mientras esté habilitado se muestran los errores, si no es así se muestran las excepciones, ver siguiente componente.
4. Botón *Mostrar excepciones*. Presenta el mismo funcionamiento con la salvedad que trata la excepciones, errores que aun siendo detectados como tal por la regla el usuario bajo su responsabilidad decide no tratarlos. Un error se marca como excepción marcando la casilla presente en la columna del área de visualización de errores, componente 14.
5. Botón *Mostrar solo área visible*. Muestra solo los errores presentes en la porción o sección visible en ese momento.
6. Botones acceso a formularios. Permite obtener formularios alfanuméricos con los datos del error.
7. *Zoom a elemento con error*. Tras seleccionar un error y pulsar este icono la vista hace zoom en el elemento que presenta el error.
8. Botón Centrar en geometría errónea. Tras seleccionar un error y pulsar este icono la vista se centra en el elemento que presenta el error y se selecciona dicho elemento (feature).
9. *Zoom a error*. Tras seleccionar un error y pulsar este icono la vista hace zoom al error como tal, es decir totalidad del elemento o porción que no satisface la regla topológica.

10. Botón *Centrar en error*. Tras seleccionar un error y pulsar este icono la vista se centra en el error como tal, es decir totalidad del elemento o porción que no satisface la regla topológica v lo selecciona
11. Botón *Eliminar error.*
12. Botón *Acceso acciones correctoras*. Tras pulsar este icono se despliegan las acciones correctoras propias de la regla con el fin de ejecutarse.
13. Botón actualizar área de visualización.
14. Área de visualización de errores.
15. Contador de errores detectados por la regla topológica.

Con esta última ventana se termina la descripción del funcionamiento del complemento que implementa la topología en gvSIG Desktop.

4. GSOC

GSOC es un programa internacional de Google destinado a impulsar el desarrollo de software libre mediante la colaboración de organizaciones y estudiantes universitarios de todo el mundo. Este programa pionero se inicia en el año 2005 y ha reunido a más de 16000 estudiantes de 118 países con 715 organizaciones de código abierto.

El funcionamiento es básico, los estudiantes previa comunicación y coordinación con las organizaciones presentan un proyecto de software libre y una planificación de éste. Dicho proyecto debe basarse en los principios de software libre y debe cumplir una serie de criterios registrados en las bases del programa. Además, la planificación del mismo, dividida en semanas, debe ser muy detalla y expresar los pasos a realizar en cada momento. Una vez presentado lo anterior solo un número determinado de proyectos serán seleccionados, dichos proyectos se realizan durante 3 meses, 10 semanas, en el periodo vacacional de julio, agosto y septiembre.

Un aspecto muy destacado de este proyecto es su marcado carácter divulgativo ya que no solo busca generar nuevos algoritmos que mejoren los diferentes softwares de las organizaciones participantes, que sin duda se consigue, o acercar a un ambiente laboral a estudiantes que realizan unas de las fases finales de su formación. El GSOC busca además de lo anterior acrecentar y hacer lo más operativa posible la comunidad de desarrollo de software libre, ya que entiende que es la base para que esta forma de desarrollar subsista.

4.1. GSOC2020 New rules for the Topology Framework in gvSIG Desktop

Dentro de este programa la Asociación gvSIG como miembro de la organización OSGeo pudo conseguir 2 estudiantes este año 2020, entre ellos el autor de este documento, que realizaron dos proyectos de mejora y ampliación del marco de topología *titulados New rules for the Topology Framework in gvSIG Desktop*. Cada proyecto presentaba diferentes reglas a implementar, siendo la información que se detalla a continuación la del proyecto realizado por José Olivas Carriquí, el cual es la base del Presente Trabajo fina de Máster.

Título: New rules for the Topology Framework in gvSIG Desktop

Organizaciones: OSGeo y Asociación gvSIG

Breve descripción: Definición, implementación y testeo de cinco reglas topológicas, así como la elaboración de documentación, plan de pruebas y acciones correctoras para dichas reglas. Las reglas son las siguientes;

- *Debe estar correctamente dentro de polígonos (Must be properly inside polygons).*
- *Contiene puntos (Contains point).*
- *Contiene un punto (Contains one point).*
- *Deben cubrirse polígonos a otros (Must cover each other).*
- *Regla definida por el usuario (UDR o User defined rule).*

Estado del Software antes del GSOC 2020: El software de escritorio gvSIG tiene algunas reglas de topología, pero tras estudio previo se detectan algunos errores en algunas de éstas. Además, se denota una importante ausencia de reglas topológicas útiles para los usuarios.

Planificación:

Vinculación comunitaria, 4 de mayo de 2020 - 31 de mayo de 2020

- Familiarizar con los mentores.
- Familiarizar con la comunidad gvSIG.
- Ayuda a corregir errores de codificación.
- Crea una página wiki para el proyecto.
- Presentarse y presentar el proyecto la lista de correo SOC.
- Investigación inicial del motor de topología de gvSIG Desktop.
- Creación y configuración de un repositorio.
- Creación de una estrategia para implementar las reglas.

Tiempo de codificación, 1 de junio de 2020-24 de agosto de 2020

Semana 1, 1 de junio de 2020 - 7 de junio de 2020

TopologyRuleMustBeProperlyInsidePolygonsPoint (1/2)

- Creación de la documentación de la regla y acciones.
- Creación un plan de prueba.
- Reporte semanal.

Semana 2, 8 de junio de 2020 - 14 de junio de 2020

TopologyRuleMustBeProperlyInsidePolygonsPoint (2/2)

- Implementación del código.
- Prueba y depuración del código.
- Reporte semanal.

Semana 3, 15 de junio de 2020 - 21 de junio de 2020

TopologyRuleContainsPointPolygon (1/2)

- Creación de la documentación de la regla y acciones.
- Creación un plan de prueba.
- Reporte semanal.

Semana 4, 22 de junio de 2020 - 28 de junio de 2020

TopologyRuleContainsPointPolygon (2/2)

- Implementación del código.
- Creación de la acción correctora *createPoint*.
- Prueba y depuración del código.
- Reporte semanal.

Semana 5, 29 de junio de 2020 - 5 de julio de 2020

TopologyRuleContainsOnePointPolygon (1/2)

- Creación de la documentación de la regla y acciones.

- Creación un plan de prueba.
- Reporte semanal.

Semana 6, 6 de julio de 2020 - 12 de julio de 2020

TopologyRuleContainsOnePointPolygon (2/2)

- Implementación del código.
- Creación de las acciones correctoras *createFeature* y *deleteFeature*.
- Prueba y depuración del código.
- Reporte semanal.

Semana 7, 13 de julio de 2020 - 19 de julio de 2020

UDR (TopologyRuleUserDefinedRule) (1/4)

- Creación de la documentación de la regla.
- Creación un plan de prueba.
- Reporte semanal.

Semana 8, 20 de julio de 2020 - 26 de julio de 2020

UDR (TopologyRuleUserDefinedRule) (2/4)

- Cambios en el núcleo/motor de topología de gvSIG Desktop. Agregar nuevos métodos; *CreateRuleParameters*, *SetParameters* y *GetParameters*. Modificar clases; *TopologyRuleFactory*, *TopologyRule* y *CreateRuleDialog*.
- Reporte semanal.

Semana 9, 27 de julio de 2020 - 2 de agosto de 2020

UDR (TopologyRuleUserDefinedRule) (3/4)

- Implementación del código.
- Reporte semanal.

Semana 10, 3 de agosto de 2020 - 9 de agosto de 2020

UDR (TopologyRuleUserDefinedRule) (4/4)

- Creación de la acción correctora *deleteFeature*.
- Prueba y depuración del código.
- Reporte semanal.

Semana 11, 10 de agosto de 2020 -16 de agosto de 2020

TopologyRuleMustCoverEachOtherPolygon (1/2)

- Creación de la documentación de la regla y acciones.
- Creación un plan de prueba.
- Reporte semanal.

Semana 12, 17 de agosto de 2020 - 23 de agosto de 2020

TopologyRuleMustCoverEachOtherPolygon (2/2)

- Implementación del código.
- Creación de las acciones correctoras *createFeature* y *deleteFeature*.

- Prueba y depuración del código.
- Reporte semanal.

Envío final, 24 de agosto de 2020 - 31 de agosto de 2020

- Reporte final.

Repositorio:

https://github.com/jolicar/GSoC2020/wiki/GSoC2020-New-rules-for-the-Topology-Framework-in-gvSIG-Desktop#basic-info

NOTA: El lenguaje con el que se ha realizado el proyecto New rules for the Topology Framework in gvSIG Desktop en el Google Summer of Code 2020 base para el presente Trabajo Fin de Máster es el inglés, pero para la realización del documento se ha optado por el castellano. Esta elección se debe a el mejor manejo de este a la hora de la realización de explicaciones debido a ser el lenguaje nativo del autor del trabajo.

5. Reglas

5.1. Consideraciones previas

Antes de proceder con la definición de las reglas se detallan una serie de consideraciones previas que facilitan la comprensión de conceptos utilizados en dicha implementación. Esas consideraciones hacen referencia a los siguientes aspectos:

- Tipo de geometrías
- Tipo de datos
- Multigeometrías
- Lenguajes de programación utilizados

5.1.1. Tipos de geometrías

Los tipos de geometrías o primitivas geométricas se derivan de la discretización de los diferentes elementos presentes en la realidad. Existen tres tipos bien diferenciados:

- **Puntos**. Describen la posición discreta de elemento a partir de coordenadas (X_1, Y_1, Z_1). Generalmente representan a las entidades que no son lo suficientemente grande como para mostrarse con una línea o polígono.
- **Líneas**. Representan la forma y situación de un elemento a partir de un conjunto ordenado de coordenadas [(X_1, Y_1, Z_1); (X_2, Y_2, Z_2); (X_3, Y_3, Z_3)].
- **Polígonos**. Representan superficies mediante áreas cerradas. Los polígonos están definidos por las líneas que conforman su contorno [(X_1, Y_1, Z_1); (X_2, Y_2, Z_2); (X_3, Y_3, Z_3); (X_4, Y_4, Z_4); (X_1, Y_1, Z_1)].

5.1.2. Tipos de datos

En el presente documento cuando se utiliza el termino tipo de dato se hace referencia al tipo de coordenada que presenta este, definiéndose los siguientes cuatro tipos en el presente proyecto.

- **2D**. Este tipo de coordenadas almacena los datos de la posición planimétrica del elemento (X, Y).
- **2DM**. Este tipo de coordenadas almacena los datos de la posición planimétrica del elemento más un tercer valor complemento (X, Y, M).
- **3D**. Este tipo de coordenadas almacena los datos de la posición planimétrica y altimétrica del elemento (X, Y, Z).
- **3DM**. Este tipo de coordenadas almacena los datos de la posición planimétrica y altimétrica del elemento más un cuarto valor complemento (X, Y, M).

5.1.3. Multigeometrías

De manera simple, una multigeometría es un conjunto de geometrías de un tipo determinado que se comportan como un único elemento.

5.1.4. Lenguajes de programación utilizados

Los lenguajes utilizados para la definición de las reglas del proyecto son los siguientes:

- **Markdown**. Este lenguaje permite dar formato HTML a un texto plano. Es sencillo de escribir y garantiza en todo momento que el contenido sea legible por el usuario. Se definió en 2004 por John Gruber y se distribuye bajo licencia BSD, licencia de software libre.
- **Jython**. Este lenguaje es una implementación del lenguaje de programación Python en el lenguaje de programación JAVA. La versión de Python implementada en el Jython de gvSIG Desktop es la 2.7 y se distribuye bajo licencia PSF v2, otro tipo de licencia de software libre.
- **Lenguaje Cosa**. Este es un lenguaje propio de gvSIG desarrollado para la ejecución de expresiones. En muchos casos es similar a la sintaxis SQL pero tiene funciones y características propias. La utilidad de este lenguaje es la sencillez de poder utilizarlo en cualquier parte de la aplicación para resolver operaciones. Además de permitir operaciones espaciales, puede registrar funciones en el programa propias del desarrollador y utilizarlas como parte de este.

5.2. Estructura de información

Tras definir en el apartado anterior algunos aspectos a tener en cuenta a la hora de implementar una regla topológica en general y en gvSIG Desktop, este apartado define el marco o como su propio nombre indica la estructura seguida para la completa definición de una regla topológica.

Esta estructura no engloba solo al código, sino todos los aspectos necesarios para la creación de esta, es decir documentación, archivos auxiliares, casos de prueba, plan de prueba, etcétera.

La estructura implementada para la definición de una regla topológica en el proyecto *GSOC2020 New rules for the Topology Framework in gvSIG Desktop* base de este Trabajo Fin de Máster se basa en el esquema de la Figura 7.

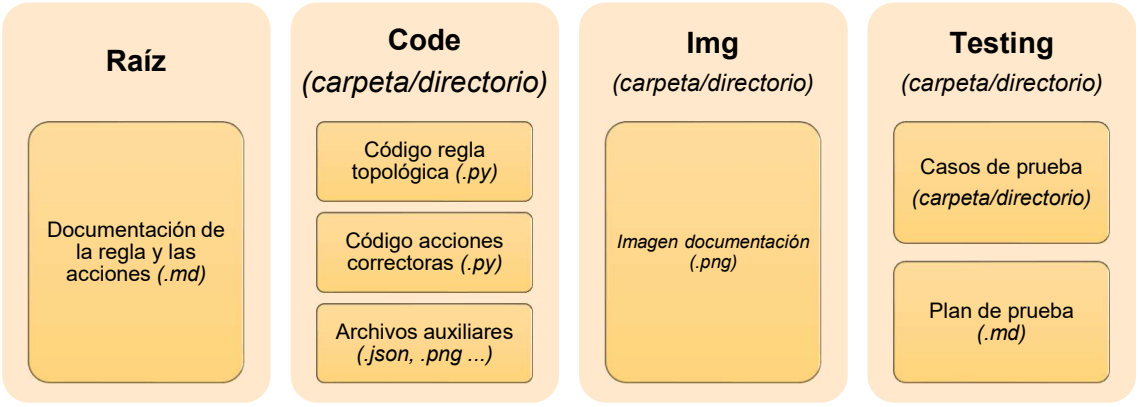


Figura 7. Carpeta o directorio Regla Topológica ejemplo TFM

Hay que destacar que el departamento de desarrollo de gvSIG Desktop de la Asociación gvSIG está considerando la futura adopción de otra distribución diferente de la anterior, ver Figura 8, pero como se menciona en el anterior párrafo la utilizada es la nombrada en primer lugar.

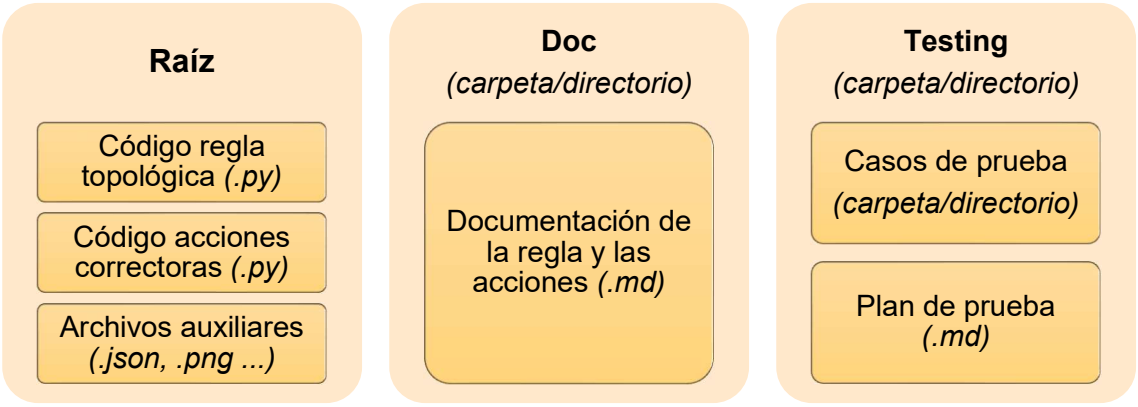


Figura 8. Nueva carpeta o directorio Regla Topológica ejemplo gvSIG

Aun siendo diferentes los esquemas se identifican tres grupos comunes de información, la **documentación**, el **código de programación** y el apartado de prueba o **testing**. Los siguientes apartados especifican la utilidad de éstos en la definición de las reglas, así como los diferentes elementos que almacenan.

5.2.1. Documentación

Tal y como se aprecia en la Figura 7, la documentación se encuentra en la raíz de los directorios de cada regla y tal como se comenta en el apartado ***5.1 Consideraciones previas*** está definida en formato Markdown (Cone, 2018).

El documento está conformado por una imagen que describe de manera gráfica el funcionamiento de la regla, así como varios apartados alfanuméricos que informan del funcionamiento de ésta de manera inequívoca.

Dichos apartados se listan a continuación, junto a una breve explicación de estos;

- **Título**.
- **Tipo de regla**. Indica el tipo de geometría al que evalúa.
- **Conjunto de datos primario**. Especifica el tipo de dato que soporta como dataset primario. En este apartado indica si soporta multigeometrías.
- **Conjunto de datos secundario**. Dicta el tipo de dato que soporta como dataset secundario. En este apartado indica si soporta multigeometrías.
- **Descripción**. Breve resumen del funcionamiento de la regla. Este hace referencias a la imagen de modo que junto a ella disipen todo tipo de dudas en cuanto al funcionamiento o evaluación de la regla.
- **Limitaciones**. Como su propio nombre describe casos que inhabilitan la evaluación de la regla.
- **Comportamientos a destacar de la regla**. Este apartado detalla comportamientos atípicos de la regla para aclarar aún más si cabe su funcionamiento.

- **Acciones correctoras**. Listado y breve descripción de las acciones correctoras que soporta la regla en cuestión.
- **Comportamientos a destacar de las acciones**. Este apartado detalla comportamientos atípicos de las acciones correctoras para aclarar aún más si cabe su funcionamiento.

El archivo de documentación se denomina *readme.md* para que pueda ser previsualizado sin necesidad de abrirlo como tal en la plataforma de control de versiones utilizada para almacenar los repositorios de cada regla.

Para finalizar este apartado, es necesario destacar la importancia de una buena documentación en cualquier desarrollo, pero más si cabe en topología. Este tipo de información en topología es crítica ya que describe con todo detalle el comportamiento final del script, de modo que el usuario pueda saber sin ninguna duda y antes de aplicarlo si esa regla es la correcta para su caso.

5.2.2. Código de programación

Todos los scripts, así como archivos auxiliares necesarios para la implementación de la regla topológica se almacenan en la carpeta *code*.

Cabría pensar que en la definición de una regla topológica solo es necesario un único fichero, nada más lejos de la realidad, ya que es necesaria una colección de éstos para su correcta definición en gvSIG Desktop. Los archivos necesarios para conformar reglas topológicas se dividen en cuatro grandes grupos;

- Ficheros propios de la regla.
- Ficheros de acciones correctoras.
- Ficheros auxiliares.
- Ficheros funcionales.

5.2.2.1. Ficheros propios de la regla

Sin lugar a duda son los ficheros principales y en ellos se aglutinan la mayoría de líneas de código necesarias para la definición de una regla. En ellos se almacenan los algoritmos tanto para la creación de estos procesos, como para realizar la evaluación topológica específica. Los ficheros que conforman este grupo son:

- Fichero *RuleFactory.py*
- Fichero *Rule.py*

Antes de hablar sobre el contenido de estos ficheros, así como las diferentes partes que los componen, es necesario decir que sus contenidos siguen la estructura de clases y métodos definidos en la interfaz JAVA del motor de Topología de gvSIG Desktop correspondiente a cada caso, *TopologyRule.java* y *TopologyRuleFactory.java*.

5.2.2.1.1. Fichero RuleFactory.py

Este fichero es el responsable de la carga de la regla en el software y la ejecución. Con otras palabras, permite añadir esa nueva regla al conjunto de topología de gvSIG. Es importante decir que solo la añade al programa, la lógica de la regla se encuentra en el fichero *Rule.py*.

A continuación, y a modo de ejemplo se muestra un breve esquema de los contenidos de este fichero sobre el archivo *Rulefactory.py* de la primera regla desarrollada, "*Debe estar correctamente dentro de polígonos" (Must be properly inside polygons).* La explicación del código como tal encuentra en el *Apartado* ***5.3.2 Código de programación.***

mustBeProperlyInsidePolygonsPointRuleFactory.py

```
import gvsig
…
from mustBeProperlyInsidePolygonsPointRule import MustBeProperlyInsidePolygonsPointRule

class MustBeProperlyInsidePolygonsPointRuleFactory(AbstractTopologyRuleFactory):

    def __init__(self):
        AbstractTopologyRuleFactory.__init__(
            …

        pathName = gvsig.getResource(__file__,'MustBeProperlyInsidePolygonsPoint.json')
        url = File(pathName).toURL()
        gvsig.logger(str(url))
        json = RuleResourceLoaderUtils.getRule(url)
        self.load_from_resource(url, json)

def createRule(self, plan, dataSet1, dataSet2, tolerance):
        rule = MustBeProperlyInsidePolygonsPointRule(plan, self, tolerance, dataSet1, dataSet2)
        return rule

def selfRegister():
    try:
        …
        manager.addRuleFactories(MustBeProperlyInsidePolygonsPointRuleFactory())
    except:
        …

def main(*args):
    pass
```

Bloque 'import' Bloque conformado por los *imports* necesarios para la carga de módulos y la ejecución del código restante.

Bloque clase En este bloque se define la clase que se crea, *MustBeProperlyInsidePolygonsPointRuleFactory,* que extiende o implementa de la clase abstracta JAVA *AbstractTopologyRuleFactory*.

NOTA: Las clases abstractas son un híbrido entre una clase 'normal' y la clase definida en un interfaz JAVA. Las interfaces indican la estructura o esqueleto que tiene que presentar una determinada clase, mientras que las clases abstractas cumplen todas las especificaciones de una interfaz de modo que ayudan al desarrollador a 'ahorrar' esfuerzos a la hora de realizar varios elementos que presentan una misma estructura. Por ultimo las clases 'normales' especifican aún más el algoritmo para casos concretos.

Bloque Constructor Aquí se conforma el constructor de la clase, código que inicializa esta con una serie de valores en este caso información de la regla, así como el proceso que realiza y los datos que necesita.

Bloque archivos auxiliares Este bloque está constituido por la líneas de código que nos permite introducir la información de varios archivos auxiliares (PNG y JSON) en la ventana de descripción de la regla. Esta funcionalidad se ve en mayor detalle en el grupo de ficheros auxiliares.

Bloque createRule Este bloque sobrescribe el método con el mismo nombre de la clase *AbstractTopologyRuleFactory* creando la regla *MustBeProperlyInsidePolygonsPointRule* en el motor de topología de gvSIG.

Bloque selfRegister Bloque que registra esta factoría en gvSIG.

Bloque main Bloque que almacena el programa principal del script. Con la definición de clases no tiene mucho sentido.

5.2.2.1.2. Fichero Rule.py

El fichero *Rule.py* implementa la lógica de la regla, es decir contiene el algoritmo que evalúa topológicamente los diferentes elementos. Tras dicha evaluación los elementos (*features)* que no cumplen la regla se envían al reporte de errores, una herramienta definida de igual manera que las reglas y factorías que permite representar dichos errores y gestionarlos.

Al igual que el fichero *RuleFactory.py* se detalla en el siguiente esquema el fichero *Rule.py* de la primera regla desarrollada, "*Debe estar correctamente dentro de polígonos" (Must be properly inside polygons)*. La explicación del código como tal se realiza en el Apartado ***5.3.2 Código de programación.***

mustBeProperlyInsidePolygonsPointRule.py

```
import gvsig
…
from deletePointAction import DeletePointAction

class MustBeProperlyInsidePolygonsPointRule(AbstractTopologyRule):

  def __init__(self, plan, factory, tolerance, dataSet1, dataSet2):
     …

  def contains(self, point1, dataSet2):
    if dataSet2.getSpatialIndex() != None:
    …
  def intersectsWithBuffer(self, point1, dataSet2):
    buffer1 = point1.buffer(self.getTolerance())
    …

  def check(self, taskStatus, report, feature1):
    try:
    …
    except:
    …

def main(*args):
    pass
```

Bloque 'import' Bloque conformado por los *imports* necesarios para la carga de módulos y la ejecución del código restante.

Bloque clase En este bloque se define la clase que se crea, *MustBeProperlyInsidePolygonsPointRule,* que extiende o implementa de la clase abstracta JAVA *AbstractTopologyRule*.

Bloque Constructor más inicializar variables Aquí se conforma el constructor de la clase, código que inicializa la clase con una serie de valores. En este bloque también se inicializan variables y añaden las acciones correctoras que presenta la regla topológica.

Bloque operaciones auxiliares Este bloque contiene operaciones lógicas que complementan a la función *check*, ayudando a ésta a evaluar los diferentes elementos que trata la regla

Bloque check Este bloque sobrescribe el método con el mismo nombre de la clase *AbstractTopologyRule.* Es el encargado de evaluar si una feature o elemento presenta las propiedades geométricas necesarias para cumplir la regla. Como se indica en el apartado anterior, puede estar complementado por otras funciones.

Bloque main Bloque que almacena el programa principal del script. Con la definición de clases no tiene mucho sentido.

5.2.2.2. Ficheros de acciones correctoras

Estos algoritmos caracterizan la acción correctora de la regla en cuestión, pudiendo llegar ésta desde la creación de nuevos elementes, modificación de los existentes hasta la eliminación si fuera el caso.

Para definir su estructura utilizaremos el fichero *deletePointAction.py*, el único fichero de acciones que presenta la regla topológica ejemplo, "*Debe estar correctamente dentro de polígonos" (Must be properly inside polygons).* La explicación del código con mayor profundidad ésta en el Apartado ***5.3.2 Código de programación.***

deletePointAction.py

```
import gvsig
…
from org.gvsig.topology.lib.spi import AbstractTopologyRuleAction
```

```
class DeletePointAction(AbstractTopologyRuleAction):
```

```
    def __init__(self):
        AbstractTopologyRuleAction.__init__(
         …
```

```
    def execute(self, rule, line, parameters):
        try:
            …
        except:
            …
```

```
def main(*args):
    pass
```

Bloque 'import' Bloque conformado por los *imports* necesarios para la ejecución del código restante.

Bloque clase En este bloque se define la clase que se crea, *DeletePointAction,* que extiende o implementa de la clase abstracta JAVA *AbstractTopologyRuleAction*.

Bloque Constructor Aquí se conforma el constructor de la clase, código que inicializa esta con una serie de valores, en este caso información de la acción, así como el proceso que realiza.

Bloque execute Este bloque sobrescribe el método con el mismo nombre de la clase *AbstractTopologyRuleAction.* Es el encargado de realizar la acción sobre el elemento que presenta error en la regla. En este caso concreto la acción elimina el elemento erróneo.

Bloque main Bloque que almacena el programa principal del script. Con la definición de clases no tiene mucho sentido.

5.2.2.3. Ficheros auxiliares

Los ficheros auxiliares son archivos opcionales que no obstante aportan un componente de mayor calidad a las reglas. Básicamente mejoran la presentación de estas en la ventana o área con la descripción literal y gráfica de la regla seleccionada, componente número 6 de la ventana *'Añadir regla'*. Puede verse dicha ventana y la zona en cuestión en la Figura, así como en el apartado ***3.2.3 Ventana 'Añadir regla'***.

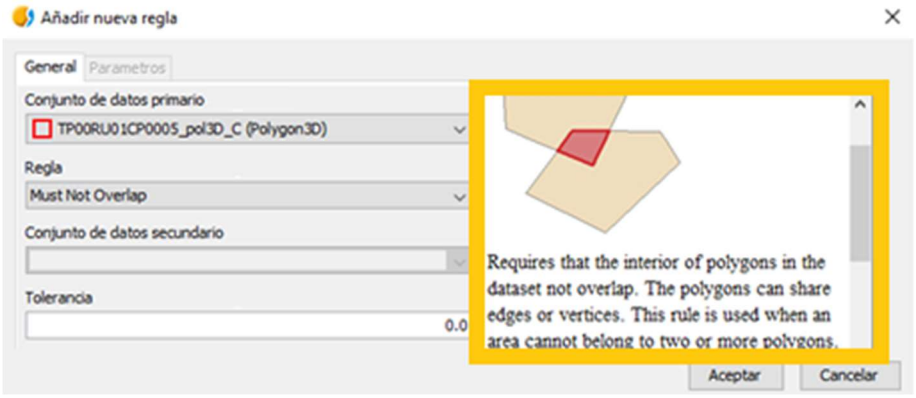


Figura 9. Ventana 'Añadir regla' con la zona de descripción gráfica y alfanumérica destacada

Los archivos auxiliares como se menciona antes mejoran la descripción de las reglas, pero no son necesarios, en el caso de no existir la ventana o área con las descripciones queda rellena con los datos de la regla especificados en el archivo *RuleFactory.py*.

Los ficheros auxiliares para definir esa mejor descripción son dos, un archivo JSON y otro PNG. Como parece lógico pensar el primer archivo contiene la descripción alfanumérica y el segundo almacena la descripción gráfica. El en ejemplo de regla tratado en el presente documentos son los siguientes:

- *MustBeProperlyInsidePolygonsPoint.json*
- *MustBeProperlyInsidePolygonsPoint.png*

5.2.2.3.1. Fichero JSON

El fichero JSON no es más que un conjunto básico de datos literales de la regla y en el caso del proyecto estos son iguales que los datos que describen la regla y las acciones en los ficheros *RuleFactory.py* y *RuleAction.py*. El siguiente esquema muestra este archivo ejemplo en la regla ejemplo.

MustBeProperlyInsidePolygonsPoint.json

```
{
    "id": "MustBeProperlyInsidePolygonsPoint",

    "name": "Must Be Properly Inside Polygons Point",

    "description": [
        "<img src=\"@@@.png\">\n",
        " This point's rule return True when the points falls … “],

    "actions" : {
        "DeletePointAction": {

            "name" : "Delete Point Action",

            "description": [
            "The delete action removes points features for cases when … ] “
        }
    }
}
```

5.2.2.3.2. Fichero PNG

Este fichero referido en el apartado del archivo anterior no es más que una versión reducida de la imagen que describe el funcionamiento de la regla de forma gráfica utilizado en la documentación. La Figura 10 muestra dicho archivo para la regla “*Debe estar correctamente dentro de polígonos” (Must be properly inside polygons)*.

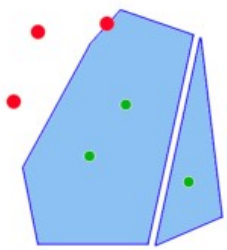


Figura 10. Imagen que almacena el fichero MustBeProperlyInsidePolygonsPoint.png

5.2.2.4. Ficheros Funcionales

Este es el último grupo de ficheros pueden parecer insignificantes, pero tiene una gran utilidad y de hecho la carencia de éstos en un desarrollo es un error determinante para un desarrollador, ya que sin ellos el código es inservible. Los ficheros funcionales son:

- *__init__.py*
- *autorun.py*

5.2.2.4.1. Fichero __init__.py

Este fichero sin código alguno, estándar de Python, tiene como único objetivo convertir la carpeta donde está contenido en un paquete. Los paquetes son archivos sobre los cuales permiten realizar imports, funcionalidad necesaria para la implementación.

5.2.2.4.2. Fichero autorun.com

Dicho fichero es el encargado de iniciar el proceso, es decir registrar la regla en el marco de topología del software. Al ser un fichero autorun el proceso se inicia de manera automática al arrancar gvSIG. Sin la ejecución de este script la regla nunca forma parte del conjunto de gvSIG Desktop. A continuación, se muestra el contenido de dicho fichero para la regla elegida como ejemplo, *"Debe estar correctamente dentro de polígonos" (Must be properly inside polygons).*

autorun.py

```
import gvsig
…
from mustBeProperlyInsidePolygonsPointRuleFactory import selfRegister

def main(*args):
    selfRegister()
```

5.2.3. Testing

Este es el último grupo de información que se deduce de la arquitectura de datos de una regla topológica y su función es testear, chequear o comprobar el correcto funcionamiento del software. Para llevar a cabo la comprobación utiliza principalmente dos figuras estrechamente relacionadas, los **casos de prueba** y el **plan de prueba**.

A ojos de un usuario común la carpeta *Testing* y la información que lleva implícita no tiene mucho sentido. Un usuario regular no modificará código, solo se limita a utilizar la regla. El apartado de *Testing* está íntimamente ligado al desarrollo y por tanto solo es útil para los desarrolladores.

5.2.3.1. Caso de prueba

"Un caso de prueba no es más que un breve documento que describe de manera detallada el comportamiento de una parte o herramienta del software al realizar una determinada acción concreta. Para que esta simulación sea de utilidad las condiciones con las que se realiza deben de ser fijas al igual que los datos utilizados, testándose únicamente el "motor" o algoritmo que trata dichos datos bajo dichas condiciones.

Por tanto, la función principal de un caso de prueba es poder detectar errores de manera fácil y sin apenas consumir tiempo y esfuerzo cada vez que se realizan cambios en el Código del software." [1]

Al igual que en el caso de la documentación y el próximo documento a definir se presenta en formato Markdown.

Las partes de dicho documento quedan listadas a continuación, junto con una breve descripción de su contenido;

- ***"Título.*** *Está compuesto por un identificador único más una breve frase que describe que lo se va a probar.*
- ***Enlace de petición existente con ese caso.*** *Enlace a la web de control de errores de la Asociación gvSIG con el identificador único de forma que se puede comprobar si ya hay algún reporte de error asociado al caso en cuestión.*
- ***Descripción.*** *Breve texto que resume la acción a realizar, así como el tipo de resultados obtenidos. La descripción debe identificar sin ningún tipo de duda el comportamiento a testear.*
- ***Prerrequisitos.*** *Lista con los datos y configuración necesarios para la correcta ejecución del caso práctico.*
- ***Pasos.*** *Secuencia de acciones a realizar que proporcionan una serie de resultados tanto intermedios como finales. Esta es la parte más importante del documento ya que en ella se detallan todos los aspectos a probar y la forma de hacerlo.*
- ***Resultados esperados.*** *Texto en el que se indican tanto los resultados intermedios si son necesarios, como los resultados finales del caso. Puede presentarse como un párrafo o en forma de lista si los resultados son varios o si así se mejora la comprensión.*
- ***Reportar fallo.*** *Parte del documento en el que se facilita la comunicación de errores detectados tras realizar el caso práctico. Se compone de dos enlaces, un primer enlace que te dirige a la página web de control de errores de la Asociación gvSIG y un segundo enlace, que previo a identificación te permite reportar en la página anterior una petición de corrección de error de dicho caso mediante su identificador único y título de manera automática."* [2]

5.2.3.2. Plan de prueba

Al igual que los casos de prueba, el plan de prueba es un documento MD o Markdown constituido por todos los casos de prueba utilizados para evaluar la regla topológica en cuestión. La finalidad de este documento es presentar todos los casos de prueba unificados en el módulo de desarrollo de gvSIG Desktop.

[1] OLIVAS, J. *Elaboración de casos de prueba en gvSIG Desktop.* https://blog.gvsig.org/2020/09/18/elaboracion-de-casos-de-prueba-en-gvsig-desktop/ [Consulta: 04 de diciembre de 2020]

[2] OLIVAS, J. *Elaboración de casos de prueba en gvSIG Desktop.* https://blog.gvsig.org/2020/09/18/elaboracion-de-casos-de-prueba-en-gvsig-desktop/ [Consulta: 04 de diciembre de 2020]

5.2.3.3. Estructura de carpetas

La estructura de ficheros utilizada en la carpeta Testing se basa en tres subcarpetas;

- Carpeta casos de prueba
- Datos
- Plan

5.2.3.3.1. Carpeta casos de prueba

"Dicha carpeta es la más importante y en ella se almacenan los casos como tal. Esta carpeta almacena todos los casos de prueba del software divididos en varias "capas" de subcarpetas o carpetas hijas." [3]

Como ejemplo se muestra, Figura 11, la distribución de carpetas que contienen los casos de prueba que almacena la regla ejemplo "*Debe estar correctamente dentro de polígonos" (Must be properly inside polygons).*

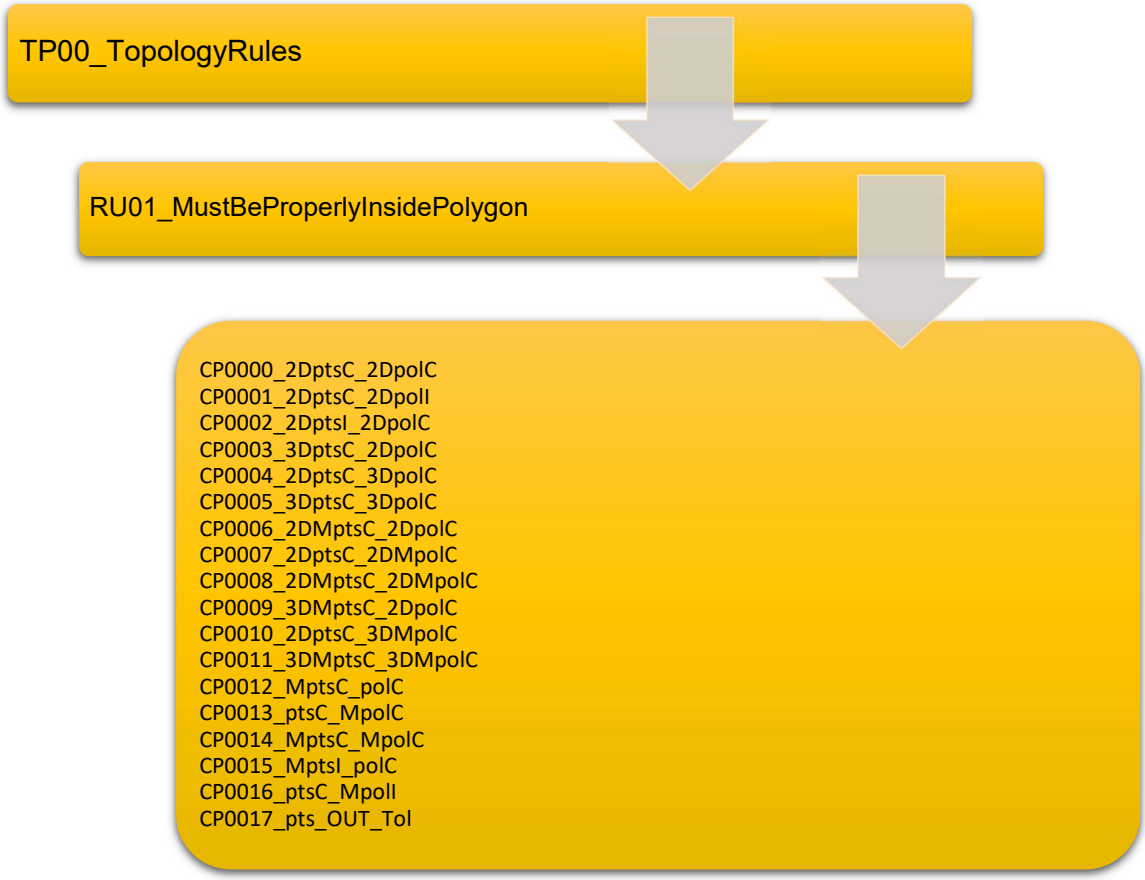


Figura 11. Esquema de estructura de la carpeta Casos de la regla topológica "Debe estar correctamente dentro de polígonos" (Must be properly inside polygons)

[3] OLIVAS, J. *Elaboración de casos de prueba en gvSIG Desktop.* https://blog.gvsig.org/2020/09/18/elaboracion-de-casos-de-prueba-en-gvsig-desktop/ [Consulta: 04 de diciembre de 2020]

La primera subcarpeta dentro de la carpeta madre *Casos* se llama *TP00_TopologyRules.* En ella se almacenarán todos los casos de prueba de topología cuando se unifiquen todas las reglas topológicas.

La siguiente carpeta en la distribución se denomina *RU01_MustBeProperlyInsidePolygon*, y almacena los casos de prueba concretos de la regla ejemplo. En el caso de *"Debe estar correctamente dentro de polígonos" (Must be properly inside polygons)*, se definieron dieciocho casos de prueba para evaluar el correcto funcionamiento del algoritmo, creándose dieciocho carpetas hijas de la anterior que almacenan el documento MD con el caso de prueba y los datos necesarios para su ejecución.

Cabe destacar que los primeros caracteres del nombre de las carpetas, hasta el carácter '_', conforman el identificador único de cada caso de prueba, un elemento necesario para el reporte de errores o la simple edificación de éstos.

5.2.3.3.2. Datos

"Esta carpeta al igual que la anterior se subdivide en otras subcarpetas que almacena si es necesario los datos para ejecutar los casos de prueba. Esta carpeta se suele utilizar si todos los casos de prueba de una herramienta presentan el mismo dataset, si cada caso presenta un conjunto de datos diferente este se almacena en la carpeta que contiene el archivo Markdown." [4]

5.2.3.3.3. Plan

En esta carpeta se guarda el plan de prueba.

5.3. Debe estar correctamente dentro de polígonos (Must be properly inside polygons)

Tras la definición "teórica" de los diferentes elementos que conforman la definición o creación de una nueva regla Topológica en gvSIG Desktop comienza ahora la definición práctica de cada una de las reglas implementadas durante la realización del GSOC 2020 comenzando por la regla que se ha tomado como ejemplo en el Apartado ***5.2 Estructura de la información,*** "*Debe estar correctamente dentro de polígonos" (Must be properly inside polygons).*

A continuación, se explica con detalle todo lo referente a esta regla, mientras que en las siguientes reglas solo se tratarán los cambios más sustanciales con respecto a ésta.

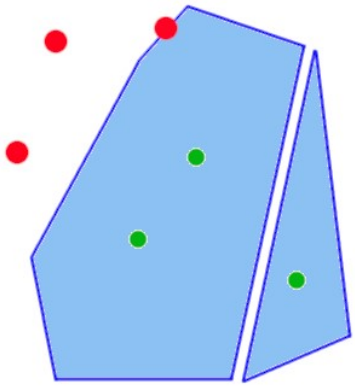
Pueden consultarse los archivos originales en el siguiente enlace web:

https://github.com/jolicar/TopologyRuleMustBeProperlyInsidePolygonsPoint

[4] OLIVAS, J. *Elaboración de casos de prueba en gvSIG Desktop.* https://blog.gvsig.org/2020/09/18/elaboracion-de-casos-de-prueba-en-gvsig-desktop/ [Consulta: 04 de diciembre de 2020]

5.3.1. Documentación

Debe estar correctamente dentro de polígonos (Must be properly inside polygons).



Tipo de regla: Regla de tipo punto.

Conjunto de datos primario: Dataset de tipo punto (2D, 2DM, 3D y 3DM) (Multigeometrías permitidas)

Conjunto de datos secundario: Dataset tipo polígono (2D, 2DM, 3D y 3DM) (Multigeometrías permitidas)

Descripción: La regla evalúa si los puntos se sitúan dentro o fuera de los polígonos. Si el punto se encuentra en su interior. Un punto cumplirá la regla cuando éste se sitúe en el interior del área un polígono, no fuera o en su borde. En la imagen incluida los puntos de color rojo no cumplen la regla, mientras que los puntos de color verde si lo hacen. Cuando se tratan datos de tipo 2DM, 3D y 3DM las coordenadas Z y M son ignoradas por la regla.

Limitaciones: Las dos fuentes de datos no pueden estar en diferentes proyecciones.

Comportamientos a destacar de la regla:

- Si la tolerancia es igual a cero, el comportamiento de la regla es el citado anteriormente. Si la tolerancia es mayor que cero, el punto evaluado se transforma en un polígono, si alguno de los puntos de ese nuevo polígono se sitúa dentro del área de algún polígono del dataset secundario, el nuevo polígono y por tanto el punto cumple la regla.
- Para multipuntos, si todas las geometrías de éstos se sitúan en el interior de un polígono o multipolígono del dataset secundario, la regla se cumplirá. Para multipolígonos si una de estas geometrías tiene al menos un punto o multipunto dentro del polígono o polígonos, la regla devuelve Verdadero (True).

Acciones correctoras:

Eliminar: La acción elimina los puntos que no cumplen la regla topológica.

Comportamientos a destacar de las acciones

(*Eliminar*) Si una geometría de un multipunto se encuentra fuera del área de un polígono o multipolígono la acción elimina el multipunto en su totalidad.

5.3.2. Código de programación

5.3.2.1. Ficheros propios de la regla.

mustBeProperlyInsidePolygonsPointRuleFactory.py

```
import gvsig
import sys

from gvsig import uselib
uselib.use_plugin("org.gvsig.topology.app.mainplugin")

from org.gvsig.fmap.geom import Geometry
from org.gvsig.tools.util import ListBuilder
from org.gvsig.topology.lib.api import TopologyLocator
from org.gvsig.topology.lib.spi import AbstractTopologyRuleFactory,
RuleResourceLoaderUtils

from java.io import File

from mustBeProperlyInsidePolygonsPointRule import
MustBeProperlyInsidePolygonsPointRule
```

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
class MustBeProperlyInsidePolygonsPointRuleFactory(AbstractTopologyRuleFactory):
```

Definición de la clase *MustBeProperlyInsidePolygonsPointRuleFactory* que implementa *AbstractTopologyRuleFactory* que a su vez implementa *TopologyRuleFactory*, clases que definen una factoría de una regla topológica en gvSIG Desktop.

```
    def __init__(self):
        AbstractTopologyRuleFactory.__init__(
            self,
            "MustBeProperlyInsidePolygonsPoint",
            "Must Be Properly Inside Polygons Poi
            "The rule evaluates the point ... the

ListBuilder().add(Geometry.TYPES.POINT).add(Geometry.TYPES.MULTIPOINT).asList(),

ListBuilder().add(Geometry.TYPES.POLYGON).add(Geometry.TYPES.MULTIPOLYGON).asList(
))
```

Constructor de la clase *MustBeProperlyInsidePolygonsPointRuleFactory*, se inicializa con los valores del nombre de la regla, etiqueta, descripción y los tipos de datos que necesita. En este caso un dataset primario de tipo punto y un dataset secundario de tipo polígono.

```
        pathName =
gvsig.getResource(__file__,'MustBeProperlyInsidePolygonsPoint.json')
        url = File(pathName).toURL()
        gvsig.logger(str(url))
        json = RuleResourceLoaderUtils.getRule(url)
        self.load_from_resource(url, json)
```

Bloque de líneas que en primer lugar obtiene el fichero auxiliar *MustBeProperlyInsidePolygonsPoint.json* y lo cargan en el área de descripción gráfica y alfanumérica de

```
    def createRule(self, plan, dataSet1, dataSet2, tolerance):
        rule = MustBeProperlyInsidePolygonsPointRule(plan, self, tolerance,
dataSet1, dataSet2)
        return rule
```

En este apartado se sobrescribe el método *createRule* de la clase abstracta *AbstractTopologyRuleFactory* con los datos necesarios para crear la regla topológica *MustBeProperlyInsidePolygonsPointRule*. Este método se ejecuta al crear el plan de topología, proceso detallado en el apartado ***3.2 Topología en gvSIG***.

```
def selfRegister():
    try:
        manager = TopologyLocator.getTopologyManager()
        manager.addRuleFactories(MustBeProperlyInsidePolygonsPointRuleFactory())
    except:
        ex = sys.exc_info()[1]
        gvsig.logger("Can't register rule. Class Name: " + ex.__class__.__name__ +
". Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

La función *selfRegister* registra la factoría de la regla en cuestión en el marco de topología. Para hacer lo anterior se basa en una estructura de *try-except*, recogiendo el error en el caso de existir.

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

mustBeProperlyInsidePolygonsPointRule.py

```
import gvsig
import sys

from gvsig import geom
from gvsig import uselib
uselib.use_plugin("org.gvsig.topology.app.mainplugin")

from org.gvsig.expressionevaluator import GeometryExpressionEvaluatorLocator,
ExpressionEvaluatorLocator
from org.gvsig.topology.lib.api import TopologyLocator
from org.gvsig.topology.lib.spi import AbstractTopologyRule
from org.gvsig.fmap.geom import GeometryLocator

from deletePointAction import DeletePointAction
```

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
class MustBeProperlyInsidePolygonsPointRule(AbstractTopologyRule):
```

Definición de la clase *MustBeProperlyInsidePolygonsPointRule* que implementa *AbstractTopologyRule* que a su vez implementa *TopologyRule*, clases que definen una regla topológica en gvSIG Desktop.

```
  def __init__(self, plan, factory, tolerance, dataSet1, dataSet2):
      AbstractTopologyRule.__init__(self, plan, factory, tolerance, dataSet1,
dataSet2)
      self.addAction(DeletePointAction())

      self.expression = ExpressionEvaluatorLocator.getManager().createExpression()
      self.expressionBuilder =
GeometryExpressionEvaluatorLocator.getManager().createExpressionBuilder()
      self.geomName=None
```

Constructor de la clase *MustBeProperlyInsidePolygonsPointRule*, se inicializa con los valores del plan de topología, la factoría de la regla, la tolerancia especificada y las fuentes de datos que necesita. En este apartado se inicializan algunas variables como el constructor de expresiones, *expressionBuilder* y se añaden las acciones correctoras a la regla, en este caso la acción Eliminar, *DeletePointAction*.

```
  def contains(self, point1, dataSet2):
    if dataSet2.getSpatialIndex() != None:
      for featureReference in dataSet2.query(point1):
        feature2 = featureReference.getFeature()
        polygon2 = feature2.getDefaultGeometry()
        if polygon2.contains(point1):
          return  True
      return False

    if self.geomName==None:
      store2 = dataSet2.getFeatureStore()
      self.geomName =
store2.getDefaultFeatureType().getDefaultGeometryAttributeName()

    self.expression.setPhrase(
      self.expressionBuilder.ifnull(
        self.expressionBuilder.column(self.geomName),
        self.expressionBuilder.constant(False),
        self.expressionBuilder.ST_Contains(
          self.expressionBuilder.column(self.geomName),
          self.expressionBuilder.geometry(point1)
        )
      ).toString()
    )
```

```
    if dataSet2.findFirst(self.expression) != None:
      return True
    return False
```

La función *contains* forma parte del bloque de operaciones auxiliares, que ayuda al método *check* a evaluar los puntos con respecto a los polígonos. La función en cuestión se utiliza cuando la tolerancia es igual a cero y necesita para ser ejecutada un punto y el conjunto total de polígonos. Su funcionamiento depende de la existencia de un índice espacial en las fuentes de datos.

- Si las fuentes de datos presentan índices espaciales la función evalúa mediante el método propio de gvSIG *query* si la envolvente (*envelop*) de algún polígono interseca con el punto. Entendiendo la envolvente (*envelop*) como el rectángulo formado con las coordenadas X e Y máximas y mínimas presente en la geometría (*bounding box*) y que por su puesto contiene a esta. Si el punto interseca no es síntoma de que esté en el interior de la geometría por lo que se ejecuta el comando *contain*, si el polígono contiene el punto se puede decir que cumple la regla y se devuelve un booleano *True* como respuesta y si no es así se devuelve *False*.
- Si las fuentes de datos no presentan índices espaciales se crea en primer lugar una expresión con el *expresionBuilder* de gvSIG que busca si alguna geometría contiene la geometría del punto mediante el método ST_Contains. El software espera al utilizar esta función ST_Contains el estándar OGC ST_Contains que al aplicarse a este tipo de dato (.CSV, .SHP…) es la función ST_Contain de gvSIG propia la cual utiliza ST_Contains de JTS. Tras la creación de la función se ejecuta el método *findFirst* sobre el dataset de polígonos aplicando la expresión anterior, de modo que si encuentra un polígono que contiene la geometría del punto, éste cumple la regla y se devuelve un booleano con valor *True*, si no es así se devuelve un valor *False* y el punto no cumple la regla topológica.

(1/2)

```
  def intersectsWithBuffer(self, point1, dataSet2):
    buffer1 = point1.buffer(self.getTolerance())

    if dataSet2.getSpatialIndex() != None:
      for featureReference in dataSet2.query(buffer1):
        feature2 = featureReference.getFeature()
        polygon2 = feature2.getDefaultGeometry()
        if polygon2.intersects(buffer1):
          return  True
      return False

    if self.geomName==None:
      store2 = dataSet2.getFeatureStore()
      self.geomName =
store2.getDefaultFeatureType().getDefaultGeometryAttributeName()

    self.expression.setPhrase(
      self.expressionBuilder.ifnull(
        self.expressionBuilder.column(self.geomName),
```

```
      self.expressionBuilder.constant(False),
      self.expressionBuilder.ST_Intersects(
        self.expressionBuilder.column(self.geomName),
        self.expressionBuilder.geometry(buffer1)
      )
    ).toString()
  )

  if dataSet2.findFirst(self.expression) != None:
    return True
  return False
```

La función *intersectsWithBuffer* es la función complementaria a la función *contains* utilizada cuando la tolerancia es superior a cero y al igual que *contains* necesita los mismos parámetros para su ejecución. Su funcionamiento también depende de la existencia de un índice espacial en las fuentes de datos. La única diferencia radica en que al punto se le aplica la función *buffer* para convertirlo en un polígono, lo que conlleva algunos cambios en el código.

- Si las fuentes de datos presentan índices espaciales la función evalúa mediante el método propio de gvSIG *query* si la envolvente (*envelop*) de algún polígono interseca con el polígono derivado del punto. Entendiendo la envolvente (*envelop*) como el rectángulo formado con las coordenadas X e Y máximas y mínimas presentes en la geometría (*bounding box*) y que por su puesto contiene a esta. Si la envolvente (*envelop*) del polígono derivado del punto interseca no garantiza que esté en el interior de la geometría por lo que se ejecuta el comando *intersects*. Si algún polígono interseca con algún punto del polígono derivado del punto se puede decir que el punto cumple la regla y se devuelve un booleano *True* como respuesta y si no es así se devuelve *False*.
- Si las fuentes de datos no presentan índices espaciales se crea en primer lugar una expresión con el *expresionBuilder* de gvSIG que busca si alguna geometría interseca la geometría del polígono derivado del punto mediante el método ST_Intersects. El software espera al utilizar esta función ST_ Intersects el estándar OGC ST_ Intersects que al aplicarse a este tipo de dato (.CSV, .SHP…) es la función ST_ Intersects de gvSIG propia la cual utiliza ST_ Intersects de JTS. Tras la creación de la función se ejecuta la función o método *findFirst* sobre el dataset de polígonos aplicando la expresión anterior, de modo que si encuentra un polígono que interseca con la geometría del polígono derivado del punto, el punto cumple la regla y se devuelve un booleano con valor *True*, si no es así se devuelve un valor *False* y el punto no cumple la regla topológica.

(2/2)

```
def check(self, taskStatus, report, feature1):
  try:
    point1 = feature1.getDefaultGeometry()
    tolerance = self.getTolerance()
```

```
dataSet2 = self.getDataSet2()
geometryType1 = point1.getGeometryType()

geomManager = GeometryLocator.getGeometryManager()
subtype = geom.D2

mustConvert2D=(not geometryType1.getSubType() == geom.D2)
```

La función *check* sobrescribe a la función con el mismo nombre de la clase AbstractTopologyRule. La funcionalidad de esta función consiste en evaluar un punto con respecto la totalidad de los polígonos ayudada por las funciones *contains* e *intersectsWithBuffer*.

Esta función en las primeras líneas de código evalúa si el tipo de dato del punto presenta geometrías de tipo 2D. Si es así almacena en una variable el booleano *True*, y si no almacena *False.*

(1/5)

```
if tolerance==0:
  operation=self.contains
else:
  operation=self.intersectsWithBuffer
```

Las líneas de código anteriores especifican cual de las funciones auxiliares se utiliza en función si el plan presenta tolerancia, almacenando la opción en una variable.

(2/5)

```
if geomManager.isSubtype(geom.POINT,geometryType1.getType()):
  if mustConvert2D:
    proj=point1.getProjection()
    point1 = geomManager.createPoint(point1.getX(),point1.getY(), subtype)
    point1.setProjection(proj)
  if not operation(point1, dataSet2):
    report.addLine(self,
      self.getDataSet1(),
      self.getDataSet2(),
      point1,
      point1,
      feature1.getReference(),
      None,
      -1,
      -1,
      False,
      "The point is not contained by the polygon.",
      ""
    )
```

El código anterior comienza el proceso de evaluación si la geometría del punto no es una multigeometría. Tras eso comprueba mediante el booleano citado en el primer cuadro de texto de este bloque si es necesario transformar el punto a 2D, si es así lo hace. Siguiendo el código se ejecuta el proceso de evaluación según la función de evaluación especificada en el segundo cuadro de texto de este bloque. Como se vio en el bloque de operaciones auxiliares, los puntos devuelven un valor booleano indicando si cumplen o no la regla, de modo que los que la cumplen pasan y los que no son registrados en una línea del reporte de errores de topología mediante la última estructura condicional del código anterior, la cual añade más información a parte del propio punto.

(3/5)

```
elif geomManager.isSubtype(geom.MULTIPOINT,geometryType1.getType()):
  if mustConvert2D:
    multipoint=geomManager.createMultiPoint(subtype)
    proj=point1.getProjection()
    multipoint.setProjection(proj)
    nPrimitives = point1.getPrimitivesNumber()
    for i in range(0, nPrimitives-1):
      point=point1.getPointAt(i)
      nPoint = geomManager.createPoint(point.getX(),point.getY(), subtype)
      multipoint.addPoint(nPoint)
  else:
    multipoint=point1
  if not operation(multipoint, dataSet2):
    report.addLine(self,
      self.getDataSet1(),
      self.getDataSet2(),
      multipoint,
      multipoint,
      feature1.getReference(),
      None,
      -1,
      -1,
      False,
      "The multipoint is not contained by the polygon.",
      ""
  )
```

El código anterior comienza el proceso de evaluación si la geometría del punto es una multigeometría. Básicamente realiza los mismos procesos citados en el cuadro de texto anterior.

(4/5)

```
else:
  report.addLine(self,
    self.getDataSet1(),
```

```
            self.getDataSet2(),
            point1,
            point1,
            feature1.getReference(),
            None,
            -1,
            -1,
            False,
            "Unsupported geometry type.",
            ""
        )
```

El código anterior registra en el reporte de errores el punto cuando presenta una geometría errónea o no contemplada en el algoritmo.

(5/5)

```
    except:
      ex = sys.exc_info()[1]
      gvsig.logger("Can't execute rule. Class Name: " + ex.__class__.__name__ + ".
Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

5.3.2.2. Ficheros de acciones correctoras.

deletePointAction.py

```
import gvsig
import sys

from org.gvsig.topology.lib.spi import AbstractTopologyRuleAction
```

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
class DeletePointAction(AbstractTopologyRuleAction):

    def __init__(self):
        AbstractTopologyRuleAction.__init__(
            self,
      "mustBeProperlyInsidePolygonsPoint",
            "DeletePointAction",
            "Delete Point Action",
            "The delete action removes points features for cases when Must be
properly inside polygons Topology Rule it is false. This point's rule return True
```

Definición de la clase *DeletePointAction* que implementa *AbstractTopologyRuleAction* que a su vez implementa *TopologyRuleAction*, clases que definen una acción correctora de una regla topológica en gvSIG Desktop.

```
when the points falls within the polygon's area, not on the boundary or out of
it."
        )
```

Constructor de la clase *DeletePointAction*, se inicializa con los valores del nombre de la regla donde se aplica, nombre de la acción, etiqueta, descripción.

```
    def execute(self, rule, line, parameters):
        try:
            dataSet = rule.getDataSet1()
            dataSet.delete(line.getFeature1())
        except:
            ex = sys.exc_info()[1]
            gvsig.logger("Can't execute action. Class Name: " +
ex.__class__.__name__ + ". Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

Función *execute* que sobrescríbela función con el mismo nombre en la clase abstracta *AbstractTopologyRuleAction*. Esta función necesita como parámetros de entrada la regla, la línea del reporte de errores que almacena el elemento con error y unos parámetros, que para esta acción contiene el valor *None*.

El código mediante estructura *try-except*, para capturar errores, obtiene el elemento de la línea del reporte de errores y la elimina.

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

5.3.2.3. Ficheros auxiliares.

En este apartado solo se trata el fichero JSON ya que el fichero PNG es una imagen igual que la presente en la documentación, pero a menor resolución.

MustBeProperlyInsidePolygonsPoint.json

```
{
    "id": "MustBeProperlyInsidePolygonsPoint",

    "name": "Must Be Properly Inside Polygons Point",

    "description": [
        "<img src=\"@@@.png\">\n",
        " This point's rule return True when the points falls within the polygon's
area, not on the boundary or out of it. The red points does the rule false and the
green points give a positive result on the rule. In 2DM, 3D and 3DM formats, the Z
coordinate or M coordinate are ignored.\n \n NOTE 1: If the Tolerance equals zero,
```

```
the rule does as above. If the tolerance is greater than zero, the point are
transformed into 'polygon'. If one point of this new polygon are inside of dataset
2 polygon, the rule return True. \n \n NOTE 2: For multipoints, if all of their
geometries are within the polygon or multipoligon, the rule returns True. For
Multipolygon, if one of these geometries has at least one point or multipoint
inside, the rule returns True. “],

    "actions" : {
        "DeletePointAction": {

            "name" : "Delete Point Action",

            "description": [
            "The delete action removes points features for cases when … ] “
        }
    }
}
```

Básicamente como se cita en el Apartado ***5.2.2.3 Ficheros auxiliares***, muestra información de la regla y la acción correctora que la complementa. La información es el identificador, su etiqueta de la regla y la descripción de la regla, el identificador de la acción, su etiqueta y la descripción de ésta.

En el archivo existe una línea que implementa el siguiente código `"<img src=\"@@@.png\">` el cual añade la imagen .PNG con el mismo nombre que el archivo tratado.

5.3.2.4. Ficheros funcionales.

Al igual que el anterior apartado solo se muestra uno de los dos archivos que componen este grupo, *autorun.py*, ya que el otro fichero no presenta ningún contenido, *__init__.py*.

autorun.py

```
import gvsig

from gvsig import uselib
uselib.use_plugin("org.gvsig.topology.app.mainplugin")

from mustBeProperlyInsidePolygonsPointRuleFactory import selfRegister
```

Bloque con todos los ‘imports’ necesarios para la ejecución del código del programa

```
def main(*args):
    selfRegister()
```

Programa principal del script. Su única función es ejecutar la función selfRegister de *MustBeProperlyInsidePolygonsPointRuleFactory que registra la regla*.

5.3.3. Testing

En este apartado se menciona el contenido de la carpeta *Testing* para la regla "*Debe estar correctamente dentro de polígonos" (Must be properly inside polygons)*.

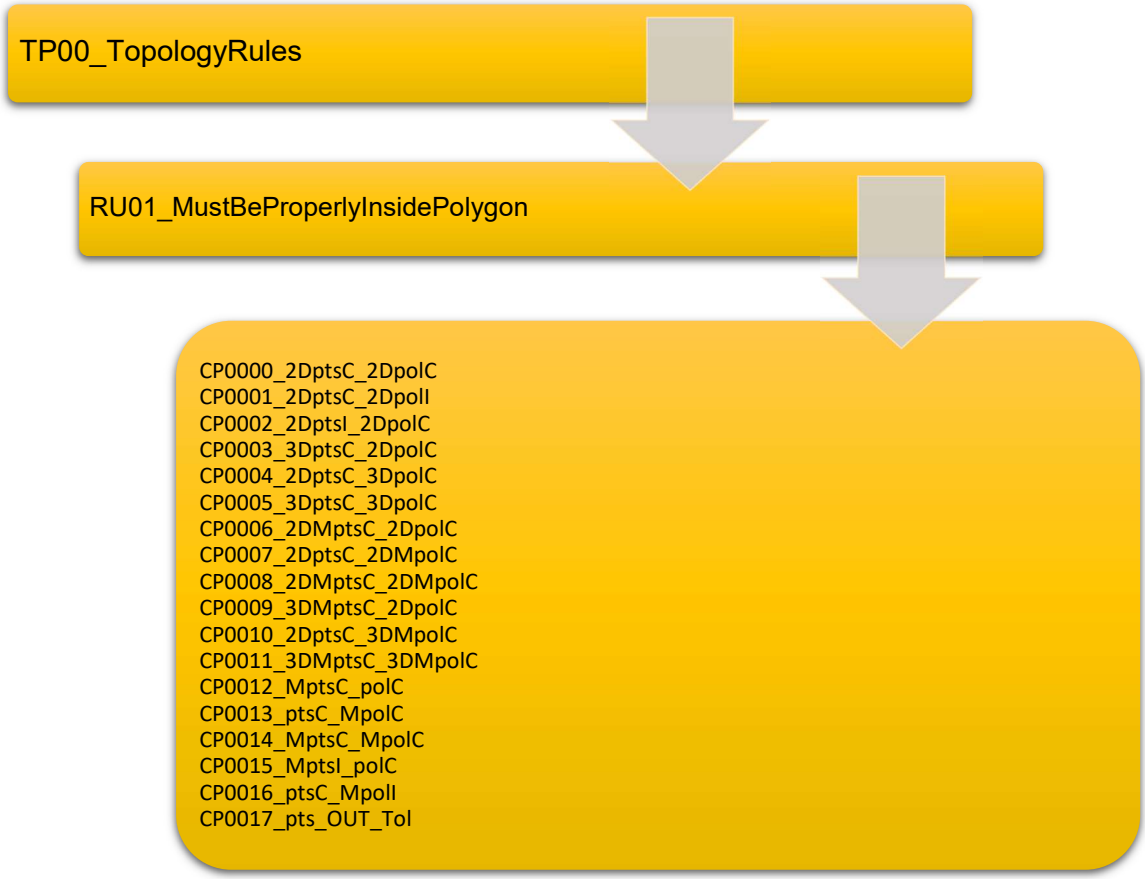


Figura 12. Esquema de estructura de la carpeta Casos de la regla topológica "Debe estar correctamente dentro de polígonos" (Must be properly inside polygons)

Tal y como se aprecia al final de la estructura de carpetas anterior, Figura 12, se almacenan dieciocho casos de prueba, los cuales están compuestos por un documento MD y dos capas de datos en formato CSV. Los casos evalúan el código según una serie de supuestos listados en la siguiente tabla, Tabla 1;

Tabla 1. Tabla con los casos de prueba de la regla topológica "Debe estar correctamente dentro de polígonos" (Must be properly inside polygons) y explicación de lo que evalúan

Caso de prueba	Descripción
CP0000_2DptsC_2DpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2D **sin** errores.
CP0001_2DptsC_2DpolI	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2D **con** errores.
CP0002_2DptsI_2DpolC	Evalúa una capa de puntos 2D **con** errores con una capa de polígonos 2D **sin** errores.
CP0003_3DptsC_2DpolC	Evalúa una capa de puntos 3D **sin** errores con una capa de polígonos 2D **sin** errores.

CP0004_2DptsC_3DpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 3D **sin** errores.
CP0005_3DptsC_3DpolC	Evalúa una capa de puntos 3D **sin** errores con una capa de polígonos 3D **sin** errores.
CP0006_2DMptsC_2DpolC	Evalúa una capa de puntos 2DM **sin** errores con una capa de polígonos 2D **sin** errores.
CP0007_2DptsC_2DMpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2DM **sin** errores.
CP0008_2DMptsC_2DMpolC	Evalúa una capa de puntos 2DM **sin** errores con una capa de polígonos 2DM **sin** errores.
CP0009_3DMptsC_2DpolC	Evalúa una capa de puntos 3DM **sin** errores con una capa de polígonos 2D **sin** errores.
CP0010_2DptsC_3DMpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 3DM **sin** errores.
CP0011_3DMptsC_3DMpolC	Evalúa una capa de puntos 3DM **sin** errores con una capa de polígonos 3DM **sin** errores.
CP0012_MptsC_polC	Evalúa una capa de multipuntos **sin** errores con una capa de polígonos **sin** errores.
CP0013_ptsC_MpolC	Evalúa una capa de puntos **sin** errores con una capa de multipolígonos **sin** errores.
CP0014_MptsC_MpolC	Evalúa una capa de multipuntos **sin** errores con una capa de multipolígonos **sin** errores.
CP0015_MptsI_polC	Evalúa una capa de multipuntos **con** errores con una capa de polígonos **sin** errores.
CP0016_ptsC_MpolI	Evalúa una capa de puntos **sin** errores con una capa de multipolígonos **con** errores.
CP0017_pts_OUT_Tol	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2D **sin** errores, utilizando una tolerancia mayor de cero.

Para no extender innecesariamente el documento se presentan como ejemplo los archivos que componen el primer caso de prueba, *CP0000_2DptsC_2DpolC*, ya que los restantes no presentan cambios suficientemente relevantes para ser mencionados como tal. Solo se presenta el fichero MD ya que los ficheros CSV solo almacenan coordenadas de las geometrías de los elementos a evaluar.

readme.md

TP00RU01CP0000 Test que todos los puntos 2D se encuentran dentro de los polígonos 2D.

En primer lugar, comprueba si ya existen peticiones sobre este caso.

Descripción:

Este caso de prueba comprueba la regla topológica cuando utilizamos fuentes de datos 2D sin errores. El resultado final es correcto.

Prerrequisitos:

1. Tener instalado *gvSIG desktop 2.5.1* y el plugin del marco de *Topología.*

2.Tener acceso a *TP00RU01CP0000_pol2D_C.csv* y *TP00RU01CP0000_pts2D_C.csv.*

Pasos

1. Cargar la capa *TP00RU01CP0000_pol2D_C.csv* en la vista.
2. Cargar la capa *TP00RU01CP0000_pts2D_C.csv* en la vista.
3. Crear un plan de topología vacío.
4. Rellenar dicho plan.
5. Añadir la capa *TP00RU01CP0000_pol2D_C.csv* como fuente de datos.
6. Añadir la capa *TP00RU01CP0000_pts2D_C.csv* file como fuente de datos.
7. Añadir parámetros en la pestaña *Reglas*.
8. Identificar el primer conjunto de datos, el segundo conjunto de datos, la regla "*Debe estar correctamente dentro de polígonos" (Must be properly inside polygons)* y la tolerancia como parámetros de la regla. La tolerancia puede ser cero o mayor.
9. Pulsar el botón 'OK'.
10. Pulsar el botón 'OK' para finalizar la creación del plan
11. Ejecutar el plan de topología.

Resultados esperados:

Los resultados esperados consisten básicamente en encontrar la ventana del Inspector de errores del Plan de topología vacía.

Reporte de errores…

En caso de que los resultados sean incorrectos, puede informar del problema en el servicio *redmine* de gvSIG desktop. Puedes encontrarlo en la siguiente dirección web:

https://redmine.gvsig.net/redmine/projects/gvsig-desktop/issues

Abrir una nueva petición con este test.

5.4. Contiene puntos (Contains point)

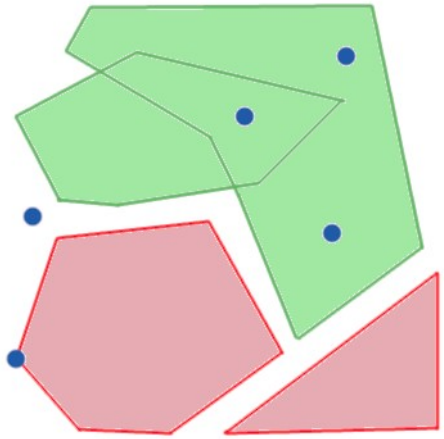
En la descripción de la regla *"Contiene puntos" (Contains point*), y el resto de reglas, únicamente se comentan las partes que difieren de manera representativa con respecto la regla ejemplo anterior, Apartado 5.3.

Pueden consultarse los archivos originales en el siguiente enlace web:

https://github.com/jolicar/TopologyRuleContainsPointPolygon

5.4.1. Documentación

Contiene puntos (Contains point)



Tipo de regla: Regla de tipo polígono.

Conjunto de datos primario: Dataset de tipo polígono (2D, 2DM, 3D y 3DM) (Multigeometrías permitidas)

Conjunto de datos secundario: Dataset tipo punto (2D, 2DM, 3D y 3DM) (Multigeometrías permitidas)

Descripción: La regla evalúa todos los polígonos del primer conjunto de datos. Si un polígono tiene al menos un punto en su interior, éste cumple la regla. Los puntos tienen que estar dentro del área del polígono, no fuera o en el borde. En la imagen incluida los polígonos de color rojo no cumplen la regla, mientras que los polígonos de color verde si lo hacen. Cuando se tratan datos de tipo 2DM, 3D y 3DM las coordenadas Z y M son ignoradas por la regla.

Limitaciones: Las dos fuentes de datos no pueden estar en diferentes proyecciones.

Comportamientos a destacar de la regla:

- Si la tolerancia es igual a cero, el comportamiento de la regla es el citado anteriormente. Si la tolerancia es mayor que cero, el polígono evaluado aumenta su tamaño con un buffer, si alguno de los puntos se sitúa en el interior del área de ese nuevo polígono, éste cumple la regla.
- Para multipolígonos, si alguna de sus geometrías contiene un punto, éste valida la regla. Para multipuntos si alguna de sus geometrías se encuentra en el interior del área de un polígono, dicho polígono cumple la regla.

Acciones correctoras:

Eliminar: La acción elimina los polígonos que no cumplen la regla topológica.

Crear punto: La acción crea un punto interno aleatorio dentro de los polígonos que no cumplen la regla.

Comportamientos a destacar de las acciones

(*Crear punto*) Si se ejecuta la acción sobre un multipolígono que no cumple la regla, el punto interno aleatorio se genera en la primera geometría.

5.4.2. Código de programación

5.4.2.1. Ficheros propios de la regla

El fichero *Rulefactory.py* no difiere demasiado con respecto al de la regla ejemplo, Apartado 5.3 Las únicas diferencias son los datos propios como el nombre, identificador y descripción y estos ya están en la documentación anterior.

containsPointPolygonRule.py

```
import gvsig
import sys

from gvsig import geom
from gvsig import uselib #para cargar plugins, scripting no tiene cargados todos
los plugins
uselib.use_plugin("org.gvsig.topology.app.mainplugin")

from org.gvsig.expressionevaluator import GeometryExpressionEvaluatorLocator,
ExpressionEvaluatorLocator
from org.gvsig.topology.lib.api import TopologyLocator
from org.gvsig.topology.lib.spi import AbstractTopologyRule
from org.gvsig.fmap.geom import GeometryLocator

from deletePolygonAction import DeletePolygonAction
from createPointAction import CreatePointAction
```

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
class ContainsPointPolygonRule(AbstractTopologyRule):
```

Definición de la clase *ContainsPointPolygonRule* que implementa *AbstractTopologyRule* que a su vez implementa *TopologyRule*, clases que definen una regla topológica en gvSIG Desktop.

```
  def __init__(self, plan, factory, tolerance, dataSet1, dataSet2):
      AbstractTopologyRule.__init__(self, plan, factory, tolerance, dataSet1,
dataSet2)
      self.addAction(DeletePolygonAction())
      self.addAction(CreatePointAction())
```

```
      self.expression = ExpressionEvaluatorLocator.getManager().createExpression()
      self.expressionBuilder =
GeometryExpressionEvaluatorLocator.getManager().createExpressionBuilder()
      self.geomName=None
```

Constructor de la clase *ContainsPointPolygonRule*, se inicializa con los valores del plan de topología, la factoría de la regla, la tolerancia especificada y las fuentes de datos que necesita. En este apartado se inicializan algunas variables como el constructor de expresiones, *expressionBuilder* y se añaden las acciones correctoras a la regla, en este caso la acción Eliminar, *DeletePolygonAction y Crear punto*, *CreatePointAction* .

```
  def contains(self, polygon1, dataSet2):
    if dataSet2.getSpatialIndex() != None:
      for featureReference in dataSet2.query(polygon1):
        feature2 = featureReference.getFeature()
        point2 = feature2.getDefaultGeometry()
        proj2=point2.getProjection()
        point2 = point2.force2D()
        point2.setProjection(proj2)
        if polygon1.contains(point2):
          return  True
      return False

    if self.geomName==None:
      store2 = dataSet2.getFeatureStore()
      self.geomName =
store2.getDefaultFeatureType().getDefaultGeometryAttributeName()

    self.expression.setPhrase(
      self.expressionBuilder.ifnull(
        self.expressionBuilder.geometry(polygon1),
        self.expressionBuilder.constant(False),
        self.expressionBuilder.ST_Contains(
          self.expressionBuilder.geometry(polygon1),

self.expressionBuilder.ST_Force2D(self.expressionBuilder.column(self.geomName))
        )
      ).toString()
    )
    if dataSet2.findFirst(self.expression) != None:
      return True
    return False
```

La función *contains* forma parte del bloque de operaciones auxiliares, que ayuda al método *check* a evaluar los polígonos con respecto a los puntos. La función en cuestión se utiliza cuando la tolerancia es igual a cero y necesita para ser ejecutada un polígono y el conjunto total de puntos. Su funcionamiento depende de la existencia de un índice espacial en las fuentes de datos.

- Si las fuentes de datos presentan índices espaciales la función evalúa mediante el método propio de gvSIG *query* si la envolvente (*envelop*) de algún punto interseca con el polígono. Entendiendo la envolvente (*envelop*) como el rectángulo formado con las coordenadas X e Y máximas y mínimas existentes en la geometría (*bounding box*) y que por su puesto contiene a esta. Si el polígono interseca no es síntoma de que éste presente un punto en su interior por lo que se ejecuta el comando *contain*, si el polígono contiene algún punto se puede decir que cumple la regla y se devuelve un booleano *True* como respuesta y si no es así se devuelve *False*.
- Si las fuentes de datos no presentan índices espaciales se crea en primer lugar una expresión con el *expresionBuilder* de gvSIG que busca si la geometría del polígono contiene geometrías de algún punto mediante el método ST_Contains. El software espera al utilizar esta función ST_Contains el estándar OGC ST_Contains que al aplicarse a este tipo de dato (.CSV, .SHP…) es la función ST_Contain de gvSIG propia la cual utiliza ST_Contains de JTS. Tras la creación de la función se ejecuta el método *findFirst* sobre el dataset de puntos aplicando la expresión anterior, de modo que si encuentra un punto que contenido en la geometría del polígono, éste cumple la regla y se devuelve un booleano con valor *True*, si no es así se devuelve un valor *False* y el polígono no cumple la regla topológica.

En este paso además se transforma de manera manual los datos de tipo punto al tipo 2D con el método force2D en el caso de la existencia de índices espaciales y ST_Force2D si no existen.

(1/2)

```
  def intersectsWithBuffer(self, polygon1, dataSet2):
    buffer1 = polygon1.buffer(self.getTolerance())
    if dataSet2.getSpatialIndex() != None:
      for featureReference in dataSet2.query(buffer1):
        feature2 = featureReference.getFeature()
        point2 = feature2.getDefaultGeometry()
        proj2=point2.getProjection()
        point2 = point2.force2D()
        point2.setProjection(proj2)
        if buffer1.contains(point2):
          return  True
      return False

    if self.geomName==None:
      store2 = dataSet2.getFeatureStore()
      self.geomName =
store2.getDefaultFeatureType().getDefaultGeometryAttributeName()
```

```
    self.expression.setPhrase(
      self.expressionBuilder.ifnull(
        self.expressionBuilder.geometry(buffer1),
        self.expressionBuilder.constant(False),
        self.expressionBuilder.ST_Contains(

self.expressionBuilder.ST_Force2D(self.expressionBuilder.column(self.geomName)),
          self.expressionBuilder.geometry(buffer1)
        )
      ).toString()
    )

    if dataSet2.findFirst(self.expression) != None:
      return True
    return False
```

La función *intersectsWithBuffer* es la función complementaria a la función *contains* utilizada cuando la tolerancia es superior a cero y al igual que *contains* necesita los mismos parámetros para su ejecución. Su funcionamiento también depende de la existencia de un índice espacial en las fuentes de datos. La única diferencia radica en que al polígono se le aplica la función *buffer* para hacerlo más grande. Por lo que el código es igual.

(2/2)

```
  def check(self, taskStatus, report, feature1): #feature1=polygon
    try:
      polygon1 = feature1.getDefaultGeometry()
      tolerance = self.getTolerance()
      dataSet2 = self.getDataSet2()
      geometryType1 = polygon1.getGeometryType()

      geomManager = GeometryLocator.getGeometryManager()
      subtype = geom.D2

      mustConvert2D=(not geometryType1.getSubType() == geom.D2)
```

La función *check* sobrescribe a la función con el mismo nombre de la clase AbstractTopologyRule. La funcionalidad de esta función consiste en evaluar un polígono con respecto la totalidad de los puntos ayudada por las funciones *contains* e *intersectsWithBuffer*.

Esta función en las primeras líneas de código evalúa si el tipo de dato del polígono presenta geometría de tipo 2D. Si es así almacena en una variable el booleano *True*, y si no almacena *False*.

(1/5)

```
if tolerance==0:
  operation=self.contains
else:
  operation=self.intersectsWithBuffer
```

Las líneas de código anteriores especifican cuál de las funciones auxiliares se utiliza en función del plan presenta tolerancia, almacenando la opción en una variable.

(2/5)

```
if geomManager.isSubtype(geom.POLYGON,geometryType1.getType()):
  if mustConvert2D:
    proj=polygon1.getProjection()
    polygon1=polygon1.force2D()
    polygon1.setProjection(proj)

  if not operation(polygon1, dataSet2):
    report.addLine(self,
      self.getDataSet1(),
      self.getDataSet2(),
      polygon1,
      polygon1,
      feature1.getReference(),
      None,
      -1,
      -1,
      False,
      "The polygon dont have any internal point.",
      ""
    )
```

El código anterior comienza el proceso de evaluación si la geometría del polígono no es una multigeometría. Tras eso comprueba mediante el booleano citado en el primer cuadro de texto de este bloque si es necesario transformar el punto a 2D, si es así lo hace con el método *force2D*. Siguiendo el código se ejecuta el proceso de evaluación según la función de evaluación especificada en el segundo cuadro de texto de este bloque. Como se vio en el bloque de operaciones auxiliares, los polígonos devuelven un valor booleano indicando si cumplen o no la regla, de modo que los que la cumplen pasan y los que no cumplen son registrados en una línea del reporte de errores de topología mediante la última estructura condicional del código anterior, la cual añade más información a parte del propio polígono.

```
elif geomManager.isSubtype(geom.MULTIPOLYGON,geometryType1.getType()):
  proj=polygon1.getProjection()
  if mustConvert2D:
    polygon1=polygon1.force2D()
    polygon1.setProjection(proj)
```

```
        if not operation(polygon1, dataSet2):
          report.addLine(self,
            self.getDataSet1(),
            self.getDataSet2(),
            multipolygon,
            multipolygon,
            feature1.getReference(),
            None,
            -1,
            -1,
            False,
            "The multipolygon dont have any internal point.",
            ""
        )
```

El código anterior comienza el proceso de evaluación si la geometría del polígono es una multigeometría. Básicamente realiza los mismos procesos citados en el cuadro de texto anterior.

(4/5)

```
      else:
        report.addLine(self,
          self.getDataSet1(),
          self.getDataSet2(),
          point1,
          point1,
          feature1.getReference(),
          None,
          -1,
          -1,
          False,
          "Unsupported geometry type.",
          ""
      )

    except:
      ex = sys.exc_info()[1]
      gvsig.logger("Can't execute rule. Class Name: " + ex.__class__.__name__ + ".
Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

El código anterior registra en el reporte de errores el polígono cuando presenta una geometría errónea o no contemplada en el algoritmo.

(5/5)

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

5.4.2.2. Ficheros acciones correctoras

Como se ve en la documentación, la regla topológica contiene dos acciones correctoras; deletePolygonAction y createPointAction. A continuación, se explica la segunda ya que la primera acción es prácticamente igual que la deletePointAction de la regla ejemplo explicada en el Apartado ***5.3.2.2 Fichero de acciones correctoras.***

createPointAction.py

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
import gvsig
import sys

from org.gvsig.topology.lib.spi import AbstractTopologyRuleAction
from org.gvsig.fmap.geom import GeometryLocator
from gvsig import geom
```

Definición de la clase *CreatePointAction* que implementa *AbstractTopologyRuleAction* que a su vez implementa *TopologyRuleAction*, clases que definen una acción correctora de una regla topológica en gvSIG Desktop.

```
class CreatePointAction(AbstractTopologyRuleAction):
```

```
    def __init__(self):
        AbstractTopologyRuleAction.__init__(
            self,
      "containsPointPolygon",
            "CreatePointAction",
            "Create Point Action",
            " This action creates a new aleatory internal point feature on the
wrong polygon feature. The behavior of the create action in multigeometries is
simple. If the multipolygon don't have at lest one point on his geometry, the
fixed action create a new aleatory internal point feature on his first geometry."
        )
```

Constructor de la clase *DeletePointAction*, se inicializa con los valores del nombre de la regla donde se aplica, nombre de la acción, etiqueta, descripción.

```
    def execute(self, rule, line, parameters):
      geomManager = GeometryLocator.getGeometryManager()
      try:
        dataSet1 = rule.getDataSet1()
        dataSet2 = rule.getDataSet2()

        store1=dataSet1.getFeatureStore()
        store2=dataSet2.getFeatureStore()

        if
geomManager.isSubtype(geom.MULTIPOLYGON,store1.getFeatures()[0].getDefaultGeometry
().getGeometryType().getType()):
          polygon=line.getGeometry().getPrimitiveAt(0)
```

```
        polygon=line.getGeometry()

        newPoint=polygon.getInteriorPoint()

        newFeature=store2.createNewFeature()

        newFeature.setDefaultGeometry(newPoint)

        dataSet2.insert(newFeature)

      except:
        ex = sys.exc_info()[1]
        gvsig.logger("Can't execute action. Class Name: " + ex.__class__.__name__
+ ". Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

Función *execute* que sobrescríbela función con el mismo nombre en la clase abstracta *AbstractTopologyRuleAction.* Esta función necesita como parámetros de entrada la regla, la línea del reporte de errores que almacena el elemento con error y unos parámetros, que para esta acción contiene el valor *None*.

El código se basa en una estructura *try-except*, para capturar errores, obtiene el elemento de la línea del reporte de errores y dependiendo si es un polígono o multipolígono introduce en el conjunto de datos secundario, puntos, un punto interior a la geometría del polígono o la primera geometría del multipolígono.

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

El resto de scripts de código son muy parecidos a los definidos en la regla ejemplo por lo que no se detallan para no sobrecargar el documento.

5.4.3. Testing

En este apartado se menciona el contenido de la carpeta *Testing* para la regla "*Contiene puntos" (Contains point)*.

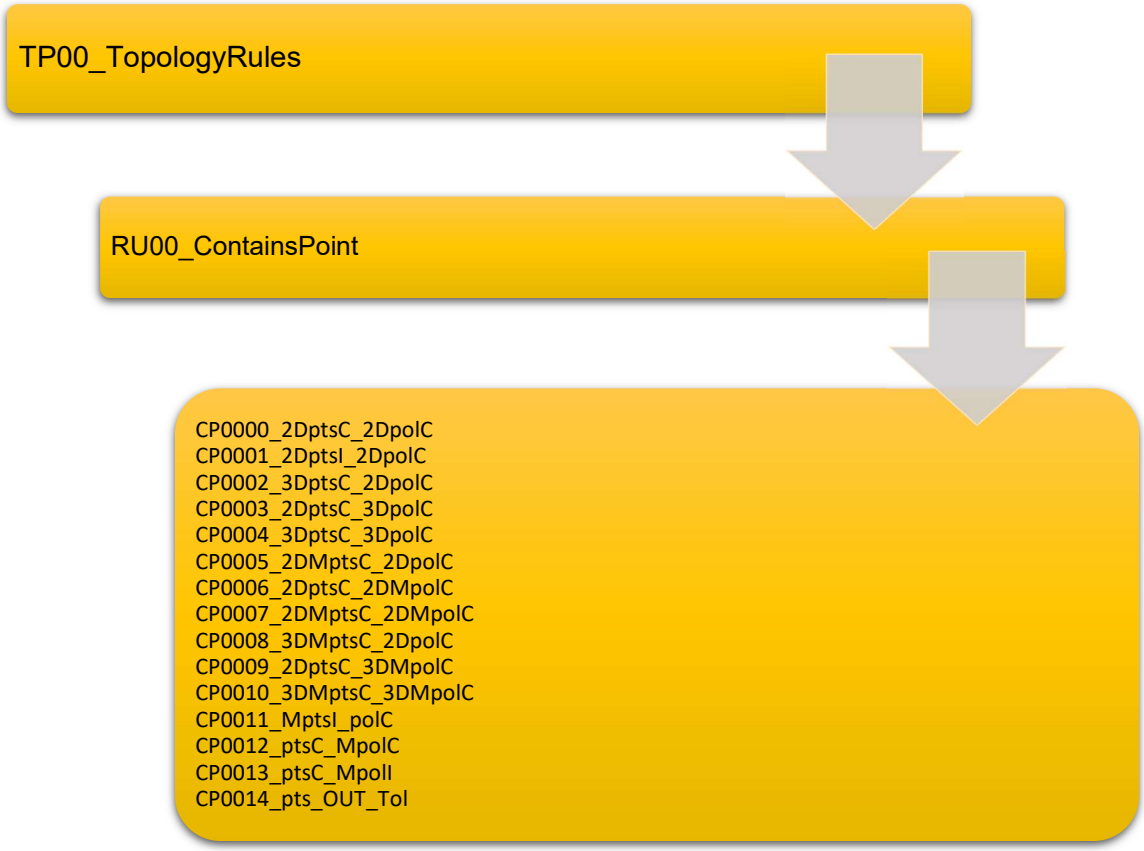


Figura 13. Esquema de estructura de la carpeta Casos de la regla topológica "Contiene puntos" (Contains point)

Tal y como se aprecia al final de la estructura de carpetas anterior, Figura 13, se almacenan quince casos de prueba, los cuales están compuestos por un documento MD y dos capas de datos en formato CSV. Los casos evalúan el código según una serie de supuestos listados en la Tabla 2;

Tabla 2. Tabla con los casos de prueba de la regla topológica "Contiene puntos" (Contains point) y explicación de lo que evalúan

Caso de prueba	Descripción
CP0000_2DptsC_2DpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2D **sin** errores.
CP0001_2DptsI_2DpolC	Evalúa una capa de puntos 2D **con** errores con una capa de polígonos 2D **sin** errores.
CP0002_3DptsC_2DpolC	Evalúa una capa de puntos 3D **sin** errores con una capa de polígonos 2D **sin** errores.
CP0003_2DptsC_3DpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 3D **sin** errores.
CP0004_3DptsC_3DpolC	Evalúa una capa de puntos 3D **sin** errores con una capa de polígonos 3D **sin** errores.

CP0005_2DMptsC_2DpolC	Evalúa una capa de puntos 2DM **sin** errores con una capa de polígonos 2D **sin** errores.
CP0006_2DptsC_2DMpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2DM **sin** errores.
CP0007_2DMptsC_2DMpolC	Evalúa una capa de puntos 2DM **sin** errores con una capa de polígonos 2DM **sin** errores.
CP0008_3DMptsC_2DpolC	Evalúa una capa de puntos 3DM **sin** errores con una capa de polígonos 2D **sin** errores.
CP0009_2DptsC_3DMpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 3DM **sin** errores.
CP0010_3DMptsC_3DMpolC	Evalúa una capa de puntos 3DM **sin** errores con una capa de polígonos 3DM **sin** errores.
CP0011_MptsI_polC	Evalúa una capa de multipuntos **con** errores con una capa de polígonos **sin** errores.
CP0012_ptsC_MpolC	Evalúa una capa de puntos **sin** errores con una capa de multipolígonos **sin** errores.
CP0013_ptsC_MpolI	Evalúa una capa de puntos **sin** errores con una capa de multipolígonos **CON** errores.
CP0014_pts_OUT_Tol	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2D **sin** errores, utilizando una tolerancia mayor de cero.

Los casos de prueba presentan siempre la misma estructura sea la herramienta que sea por lo que no se exponen ninguno de la regla en cuestión, evitando así sobrecargar el documento.

5.5. Contiene un punto (Contains one point)

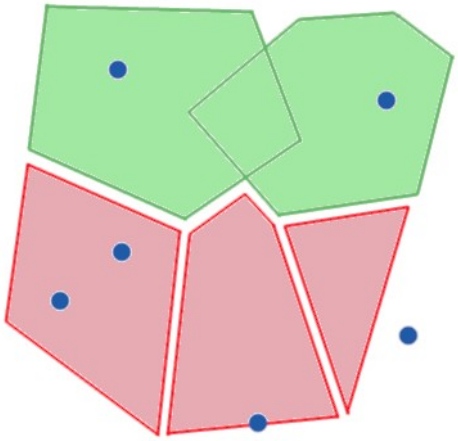
En la regla *"Contiene un punto" (Contains one point*) únicamente se destacan las partes que difieren de manera representativa con respecto la regla ejemplo, Apartado 5.3.

Pueden consultarse los archivos originales en el siguiente enlace web:

https://github.com/jolicar/TopologyRuleContainsOnePointPolygon

5.5.1. Documentación

Contiene un punto (Contains one point)



Tipo de regla: Regla de tipo polígono.

Conjunto de datos primario: Dataset de tipo polígono (2D, 2DM, 3D y 3DM) (Multigeometrías permitidas)

Conjunto de datos secundario: Dataset tipo punto (2D, 2DM, 3D y 3DM) (Multigeometrías permitidas)

Descripción: La regla evalúa todos los polígonos del primer conjunto de datos. Si un polígono tiene solo un punto en su interior, éste cumple la regla. El punto tiene que estar dentro del área del polígono, no fuera o en el borde. En la imagen incluida los polígonos de color rojo no cumplen la regla, mientras que los polígonos de color verde si lo hacen. Cuando se tratan datos de tipo 2DM, 3D y 3DM las coordenadas Z y M son ignoradas por la regla.

Limitaciones: Las dos fuentes de datos no pueden estar en diferentes proyecciones.

Comportamientos a destacar de la regla:

- Si la tolerancia es igual a cero, el comportamiento de la regla es el citado anteriormente. Si la tolerancia es mayor que cero, el polígono evaluado aumenta su tamaño con un buffer, si un punto se sitúa en el interior del área de ese nuevo polígono este cumple la regla.
- Para multipolígonos, si alguna de sus geometrías contiene un punto, éste valida la regla. Para multipuntos si solo una de sus geometrías se encuentra en el interior del área de un polígono, dicho polígono cumple la regla.

Acciones correctoras:

Eliminar: La acción elimina los polígonos que no cumplen la regla topológica.

Crear punto: La acción crea un punto interno aleatorio dentro de los polígonos que no cumplen la regla.

Comportamientos a destacar de las acciones

(*Crear punto*) Si se ejecuta la acción sobre un multipolígono que no cumple la regla, el punto interno aleatorio se genera en la primera geometría.

5.5.2. Código de programación

5.5.2.1. Ficheros propios de la regla

El fichero *Rulefactory.py* no difiere demasiado con respecto al de la regla ejemplo por lo que se muestran únicamente las partes más destacadas de esta. Las únicas diferencias son los datos propios como el nombre, identificador y descripción y estos ya están en la documentación anterior.

containsOnePointPolygonRule.py

```
import gvsig
import sys

from gvsig import geom
from gvsig import uselib
uselib.use_plugin("org.gvsig.topology.app.mainplugin")

from org.gvsig.expressionevaluator import GeometryExpressionEvaluatorLocator,
ExpressionEvaluatorLocator
from org.gvsig.topology.lib.api import TopologyLocator
from org.gvsig.topology.lib.spi import AbstractTopologyRule
from org.gvsig.fmap.geom import GeometryLocator

from deletePolygonAction import DeletePolygonAction
from createPointAction import CreatePointAction
```

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
class ContainsOnePointPolygonRule(AbstractTopologyRule):
```

Definición de la clase *ContainsOnePointPolygonRule* que implementa *AbstractTopologyRule* que a su vez implementa *TopologyRule*, clases que definen una regla topológica en gvSIG Desktop.

```
  def __init__(self, plan, factory, tolerance, dataSet1, dataSet2):
      AbstractTopologyRule.__init__(self, plan, factory, tolerance, dataSet1,
dataSet2)
      self.addAction(DeletePolygonAction())
      self.addAction(CreatePointAction())

      self.expression = ExpressionEvaluatorLocator.getManager().createExpression()
      self.expressionBuilder =
GeometryExpressionEvaluatorLocator.getManager().createExpressionBuilder()
      self.geomName=None
```

Constructor de la clase *ContainsOnePointPolygonRule*, se inicializa con los valores del plan de topología, la factoría de la regla, la tolerancia especificada y las fuentes de datos que necesita. En este apartado se inicializan algunas variables como el constructor de expresiones, *expressionBuilder* y se añaden las acciones correctoras a la regla, en este caso la acción Eliminar, *DeletePolygonAction y Crear punto*, *CreatePointAction* .

```
  def contains(self, polygon1, dataSet2):
    if dataSet2.getSpatialIndex() != None:
      i=0
      for featureReference in dataSet2.query(polygon1):
        feature2 = featureReference.getFeature()
        point2 = feature2.getDefaultGeometry()
        proj2=point2.getProjection()
        point2 = point2.force2D()
        point2.setProjection(proj2)
        if polygon1.contains(point2):
          i+=1
        if i>1:
          return False
      if i==1:
        return True

    store2 = dataSet2.getFeatureStore()
    if self.geomName==None:
      store2 = dataSet2.getFeatureStore()
      self.geomName =
store2.getDefaultFeatureType().getDefaultGeometryAttributeName()

    self.expression.setPhrase(
      self.expressionBuilder.ifnull(
        self.expressionBuilder.geometry(polygon1),
        self.expressionBuilder.constant(False),
        self.expressionBuilder.ST_Contains(
          self.expressionBuilder.geometry(polygon1),
   self.expressionBuilder.ST_Force2D(self.expressionBuilder.column(self.geomName))
        )
      ).toString()
    )
    fSet2=store2.getFeatureSet(self.expression)
    if fSet2.getSize()==1:
      return True
    return False
```

La función *contains* forma parte del bloque de operaciones auxiliares, que ayuda al método *check* a evaluar los polígonos con respecto a los puntos. La función en cuestión se utiliza cuando la tolerancia es igual a cero y necesita para ser ejecutada un polígono y el conjunto total de puntos. Su funcionamiento depende de la existencia de un índice espacial en las fuentes de datos y es igual que la función *contains* de la regla *"Contiene puntos" (Contains point).* La única diferencia se da al evaluar los polígonos de modo que, si existen dos puntos en el interior de un polígono, éste no cumple la regla.

(1/2)

```
  def intersectsWithBuffer(self, polygon1, dataSet2):
```

```
    buffer1 = polygon1.buffer(self.getTolerance())

    if dataSet2.getSpatialIndex() != None:
      i=0
      for featureReference in dataSet2.query(buffer1):
        feature2 = featureReference.getFeature()
        point2 = feature2.getDefaultGeometry()
        proj2=point2.getProjection()
        point2 = point2.force2D()
        point2.setProjection(proj2)
        if buffer1.intersects(point2):
          i+=1
        if i>1:
          return False
      if i==1:
        return True

    store2 = dataSet2.getFeatureStore()
    if self.geomName==None:
      self.geomName =
store2.getDefaultFeatureType().getDefaultGeometryAttributeName()

    self.expression.setPhrase(
      self.expressionBuilder.ifnull(
        self.expressionBuilder.geometry(buffer1),
        self.expressionBuilder.constant(False),
        self.expressionBuilder.ST_Intersects(

self.expressionBuilder.ST_Force2D(self.expressionBuilder.column(self.geomName)),
          self.expressionBuilder.geometry(buffer1)
        )
      ).toString()
    )

    fSet2=store2.getFeatureSet(self.expression)
    if fSet2.getSize()==1:
      return True
    return False
```

La función *intersectsWithBuffer* es la función complementaria a la función *contains* utilizada cuando la tolerancia es superior a cero y al igual que *contains* necesita los mismos parámetros para su ejecución. Su funcionamiento también depende de la existencia de un índice espacial en las fuentes de datos. La única diferencia radica en que al polígono se le aplica la función *buffer* para hacerlo más grande. Por lo que el código es igual.

(2/2)

```
def check(self, taskStatus, report, feature1): #feature1=polygon
  try:
    polygon1 = feature1.getDefaultGeometry()
    tolerance = self.getTolerance()
    dataSet2 = self.getDataSet2()
    geometryType1 = polygon1.getGeometryType()

    geomManager = GeometryLocator.getGeometryManager()

    mustConvert2D=(not geometryType1.getSubType() == geom.D2)
```

La función *check* sobrescribe a la función con el mismo nombre de la clase AbstractTopologyRule. La funcionalidad de esta función consiste en evaluar un polígono con respecto a la totalidad de los puntos ayudada por las funciones *contains* e *intersectsWithBuffer*.

Esta función en las primeras líneas de código evalúa si el tipo de dato del polígono presenta geometría de tipo 2D. Si es así almacena en una variable el booleano *True*, y si no almacena *False.*

(1/5)

```
    if tolerance==0:
      operation=self.contains
    else:
      operation=self.intersectsWithBuffer
```

Las líneas de código anteriores especifican cuál de las funciones auxiliares se utiliza en función si el plan presenta tolerancia o no, almacenado este parámetro en una variable.

(2/5)

```
    if geomManager.isSubtype(geom.POLYGON,geometryType1.getType()):
      if mustConvert2D:
        proj=polygon1.getProjection()
        polygon1=polygon1.force2D()
        polygon1.setProjection(proj)

      if not operation(polygon1, dataSet2):
        report.addLine(self,
          self.getDataSet1(),
          self.getDataSet2(),
          polygon1,
          polygon1,
          feature1.getReference(),
          None,
          -1,
          -1,
          False,
```

```
            "The polygon dont have any internal point or has more than one
internal point.",
            ""
          )
```

El código anterior comienza el proceso de evaluación si la geometría del polígono no es una multigeometría. Tras eso comprueba mediante el booleano citado en el primer cuadro de texto de este bloque si es necesario transformar el punto a 2D, si es así lo hace con el método *force2D*. Siguiendo el código se ejecuta el proceso de evaluación según la función de evaluación especificada en el segundo cuadro de texto de este bloque. Como se vio en el bloque de operaciones auxiliares, los polígonos devuelven un valor booleano indicando si cumplen o no la regla, de modo que los que la cumplen pasan y los que no la cumplen son registrados en una línea del reporte de errores de topología mediante la última estructura condicional del código anterior, la cual añade más información a parte del propio polígono.

(3/5)

```
      elif geomManager.isSubtype(geom.MULTIPOLYGON,geometryType1.getType()):
        proj=polygon1.getProjection()
        if mustConvert2D:
          polygon1=polygon1.force2D()
          polygon1.setProjection(proj)
        if not operation(polygon1, dataSet2):
          report.addLine(self,
            self.getDataSet1(),
            self.getDataSet2(),
            polygon1,
            polygon1,
            feature1.getReference(),
            None,
            -1,
            -1,
            False,
            "The multipolygon dont have any internal point or has more than one
internal point.",
            ""
        )
```

El código anterior comienza el proceso de evaluación si la geometría del polígono es una multigeometria. Básicamente realiza los mismos procesos citados en el cuadro de texto anterior.

(4/5)

```
      else:
        report.addLine(self,
          self.getDataSet1(),
          self.getDataSet2(),
          polygon1,
          polygon1,
          feature1.getReference(),
          None,
          -1,
          -1,
          False,
          "Unsupported geometry type.",
          ""
      )

    except:
      ex = sys.exc_info()[1]
      gvsig.logger("Can't execute rule. Class Name: " + ex.__class__.__name__ + ".
Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

El código anterior registra en el reporte de errores el polígono cuando presenta una geometría errónea o no contemplada en el algoritmo.

(5/5)

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

El resto de scripts de código son muy parecidos a los definidos en la regla ejemplo, Apartado 5.3, o alguna otra regla de las anteriores, por lo que no se detallan para no sobrecargar el documento.

5.5.3. Testing

En este apartado se menciona el contenido de la carpeta *Testing* para la regla "*Contiene un punto" (Contains one point)*.

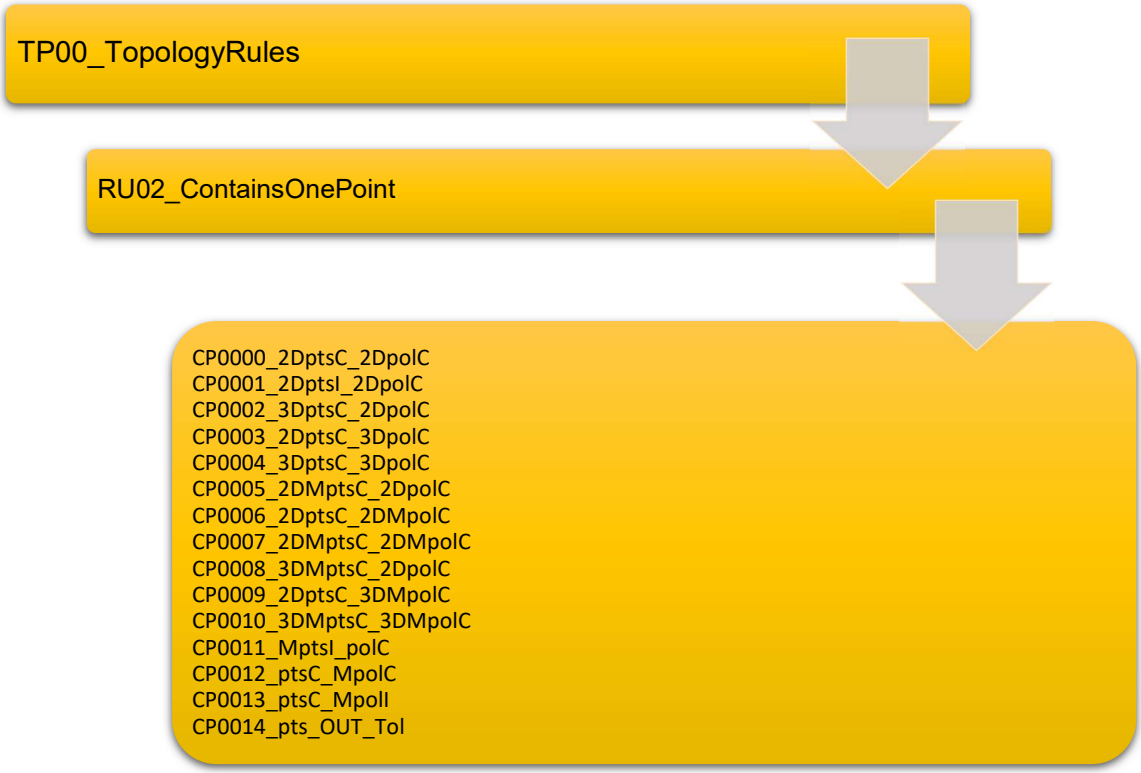


Figura 14. Esquema de estructura de la carpeta Casos de la regla topológica "Contiene un punto" (Contains one point)

Tal y como se aprecia al final de la estructura de carpetas anterior, Figura 14, se almacenan quince casos de prueba, los cuales están compuestos por un documento MD y dos capas de datos en formato CSV. Los casos evalúan el código según una serie de supuestos listados en la Tabla 3;

Tabla 3. Tabla con los casos de prueba de la regla topológica "Contiene puntos" (Contains point) y explicación de lo que evalúan.

Caso de prueba	Descripción
CP0000_2DptsC_2DpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2D **sin** errores.
CP0001_2DptsI_2DpolC	Evalúa una capa de puntos 2D **con** errores con una capa de polígonos 2D **sin** errores.
CP0002_3DptsC_2DpolC	Evalúa una capa de puntos 3D **sin** errores con una capa de polígonos 2D **sin** errores.
CP0003_2DptsC_3DpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 3D **sin** errores.
CP0004_3DptsC_3DpolC	Evalúa una capa de puntos 3D **sin** errores con una capa de polígonos 3D **sin** errores.

CP0005_2DMptsC_2DpolC	Evalúa una capa de puntos 2DM **sin** errores con una capa de polígonos 2D **sin** errores.
CP0006_2DptsC_2DMpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2DM **sin** errores.
CP0007_2DMptsC_2DMpolC	Evalúa una capa de puntos 2DM **sin** errores con una capa de polígonos 2DM **sin** errores.
CP0008_3DMptsC_2DpolC	Evalúa una capa de puntos 3DM **sin** errores con una capa de polígonos 2D **sin** errores.
CP0009_2DptsC_3DMpolC	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 3DM **sin** errores.
CP0010_3DMptsC_3DMpolC	Evalúa una capa de puntos 3DM **sin** errores con una capa de polígonos 3DM **sin** errores.
CP0011_MptsI_polC	Evalúa una capa de multipuntos **con** errores con una capa de polígonos **sin** errores.
CP0012_ptsC_MpolC	Evalúa una capa de puntos **sin** errores con una capa de multipolígonos **sin** errores.
CP0013_ptsC_MpolI	Evalúa una capa de puntos **sin** errores con una capa de multipolígonos **con** errores.
CP0014_pts_OUT_Tol	Evalúa una capa de puntos 2D **sin** errores con una capa de polígonos 2D **sin** errores, utilizando una tolerancia mayor de cero.

Los casos de prueba presentan siempre la misma estructura sea la herramienta que sea por lo que no se exponen ninguno de la regla en cuestión, evitando así sobrecargar el documento.

5.6. Deben cubrirse unos polígonos a otros (Must cover each other)

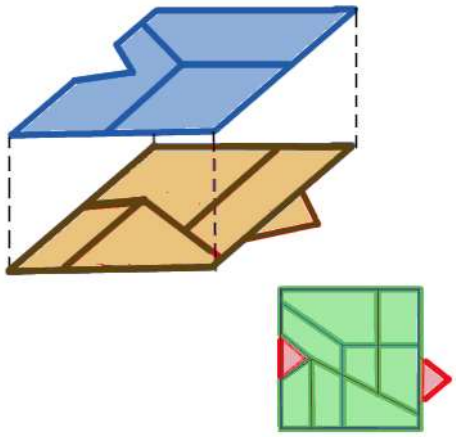
En la regla topológica *"Deben cubrirse unos polígonos a otros" (Must cover each other)* únicamente se destacan las partes que difieren de manera representativa con respecto la regla ejemplo.

Pueden consultarse los archivos originales en el siguiente enlace web:

https://github.com/jolicar/TopologyRuleMustCoverEachOtherPolygon

5.6.1. Documentación

Deben cubrirse unos polígonos a otros (Must cover each other)



Tipo de regla: Regla de tipo polígono.

Conjunto de datos primario: Dataset de tipo polígono (2D, 2DM, 3D y 3DM) (Multigeometrías permitidas)

Conjunto de datos secundario: Dataset de tipo polígono (2D, 2DM, 3D y 3DM) (Multigeometrías permitidas)

Descripción: Todos los polígonos del conjunto de datos secundario tienen que cubrir al conjunto de polígonos de la fuente de datos primario y viceversa. La regla no se cumple cuando un polígono o parte de uno no queda cubierto. Cuando se tratan datos de tipo 2DM, 3D y 3DM las coordenadas Z y M son ignoradas por la regla.

Limitaciones: Las dos fuentes de datos no pueden estar en diferentes proyecciones.

Comportamientos a destacar de la regla:

- El parámetro tolerancia no tiene sentido en esta regla.

Acciones correctoras:

Eliminar: La acción elimina los polígonos que no cumplen la regla topológica.

Crear polígono: La acción crea un nuevo elemento en la zona no cubierta.

Comportamientos a destacar de las acciones

(*Crear polígono*) Los polígonos de ambas fuentes de datos tiene que quedar perfectamente cubiertos.

5.6.2. Código de programación

5.6.2.1. Ficheros propios de la regla

El fichero *Rulefactory.py* no difiere demasiado con respecto al de la regla ejemplo, Apartado 5.3, por lo que solo se muestran las diferencias relevantes. Las únicas diferencias son los datos propios como el nombre, identificador y descripción y estos ya están en la documentación anterior.

mustCoverEachOtherPolygonRule.py

```
import sys

from gvsig import geom
from gvsig import uselib
uselib.use_plugin("org.gvsig.topology.app.mainplugin")

from org.gvsig.expressionevaluator import GeometryExpressionEvaluatorLocator,
ExpressionEvaluatorLocator
from org.gvsig.topology.lib.api import TopologyLocator
from org.gvsig.topology.lib.spi import AbstractTopologyRule
from org.gvsig.fmap.geom import GeometryLocator

from org.gvsig.tools.task import SimpleTaskStatus

from deletePolygonAction import DeletePolygonAction
from createPolygonAction import CreatePolygonAction
```

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
class MustCoverEachOtherPolygonRule(AbstractTopologyRule):
```

Definición de la clase *MustCoverEachOtherPolygonRule* que implementa *AbstractTopologyRule* que a su vez implementa *TopologyRule*, clases que definen una regla topológica en gvSIG Desktop.

```
  def __init__(self, plan, factory, tolerance, dataSet1, dataSet2):
      AbstractTopologyRule.__init__(self, plan, factory, tolerance, dataSet1,
dataSet2)
      self.addAction(DeletePolygonAction())
      self.addAction(CreatePolygonAction())

      self.expression = ExpressionEvaluatorLocator.getManager().createExpression()
      self.expressionBuilder =
GeometryExpressionEvaluatorLocator.getManager().createExpressionBuilder()
      self.geomName=None
```

Constructor de la clase *MustCoverEachOtherPolygonRule*, se inicializa con los valores del plan de topología, la factoría de la regla, la tolerancia especificada y las fuentes de datos que necesita. En este apartado se inicializan algunas variables como el constructor de expresiones, *expressionBuilder* y se añaden las acciones correctoras a la regla, en este caso la acción Eliminar, *DeletePolygonAction y Crear punto*, *CreatePolygonAction* .

```
  def getSteps(self):
      return self.getDataSet1().getSize()+self.getDataSet2().getSize()
```

```
  def execute(self, taskStatus, report):
    try:
      dataSet1 = self.getDataSet1()
      dataSet2 = self.getDataSet2()

      store1=dataSet1.getFeatureStore()
      store2=dataSet2.getFeatureStore()

      if store1.getSRSDefaultGeometry()==store2.getSRSDefaultGeometry():
        AbstractTopologyRule.execute(self, taskStatus, report)
      else:
        report.addLine(self,
          self.getDataSet1(),
          self.getDataSet2(),
          None,
          None,
          None,
          None,
          -1,
          -1,
          False,
          "Can't execute rule. The two datasets cant have a different
projection.",
          ""
        )
```

La función *execute* sobrescribe a la función con el mismo nombre de la clase AbstractTopologyRule. El cometido de esta función es evaluar las dos fuentes de datos de tipo polígono y ver qué elementos de estas la cumplen.

Esta función en las primeras líneas de código evalúa si los dos conjuntos de datos presentan el mismo sistema de referencia, si no es así introduce un registro en el reporte de errores de topología.

(1/8)

```
      features1=store1.getFeatures()
      features2=store2.getFeatures()

      candidateEr1List=[]
      candidateEr2List=[]
      er1ListAux=[]
      er2ListAux=[]

      for feature1 in features1:
        taskStatus.incrementCurrentValue()
        feature1M=convert2D(self, feature1, report)
        checkIntersects=False
```

El código anterior inicia el proceso de recorrer todos los polígonos de la primera fuente de datos e inicializa un variable booleana de control con *False*. Inicia un bucle anidado que evaluará todos los polígonos del dataset 2 por cada polígono del dataset 1.

(2/8)

```
for feature2 in features2:
  feature2M=convert2D(self, feature2, report)
  if feature1M.intersects(feature2M):
    candidateEr1=feature2M.difference(feature1M)
    checkIntersects=True
    if candidateEr1!=None:
      candidateEr1Dic={}
      candidateEr1Dic["geom"]=candidateEr1
      candidateEr1Dic["ref"]=feature2.getReference()
      candidateEr1List.append(candidateEr1Dic)
```

El código anterior empieza a recorrer los polígonos de la segunda fuente de datos para realizar el bucle embebido citado anteriormente. El objetivo del bucle es aplicar el método *difference* que proporciona la geometría diferencia de una geometría respecto a otras geometrías si estas dos geometrías previamente intersecan, dato que se obtiene con el método *intersects*.

Si se obtiene la diferencia ente las geometrías del dataset 2 y la geometría del dataset 1, la variable de control toma el valor *True.* Además, se evalúa esa diferencia y si es diferente de *None* se almacena en una lista de diccionarios, presentando cada diccionario la diferencia en cuestión y la referencia del polígono del dataset 2 que la genera. Esa lista almacena los presuntos errores candidatos del conjunto de datos 1 y se llama *candidateEr1List*.

(3/8)

```
if checkIntersects==False:
  candidateEr1Dic={}
  candidateEr1Dic["geom"]=feature1.getDefaultGeometry()
  candidateEr1Dic["ref"]=feature1.getReference()
  er1ListAux.append(candidateEr1Dic)
```

Si ninguno de los elementos del conjunto de datos 2 intersecan con el polígono del conjunto de datos 1, la variable control permanece con el valor *False.* Lo anterior hace falsa la regla por lo que se incluye al elemento del conjunto de datos número 1 en una lista de errores, *er1ListAux*.

(4/8)

```
for dic in candidateEr1List:
  for feature1 in features1:
    feature1M=convert2D(self, feature1, report)
```

```
      if dic["geom"] == None:
        continue
      if dic["geom"].intersects(feature1M):
        dic["geom"]=dic["geom"].difference(feature1M)
```

El código anterior genera de nuevo un bucle embebido que recorre ahora los elementos de la lista de candidatos de error del dataset 1 en primer lugar y en segundo lugar los elementos del dataset 1.

Dentro de bucle se evalúan las geometrías de la lista, si ésta es *None* se pasa al siguiente elemento y si no, se comprueba si las dos geometrías intersecan. Si es así se obtiene la diferencia entre la geometría de ese error candidato y la geometría de los elementos del conjunto de datos 1 y se almacena en la misma lista, *candidateEr1List.* Este proceso refina los supuestos errores.

(5/8)

```
er1List=[]
for dic in candidateEr1List:
  if dic not in er1List:
    er1List.append(dic)

er1List+=er2ListAux
```

Para ir finalizando el proceso de evaluación se crea una lista de errores vacía, *er1List*, y se recorre con un bucle la lista de errores candidatos, *candidateEr1List.* Ese bucle busca rellenar la lista de errores definitivos con la de candidatos sin repetir ningún elemento.

Tras eso se fusionan la lista anterior con la lista de errores procedentes de no intersecar elementos del dataset 2 con el dataset 1, *er2ListAux;*

(6/8)

```
for dic in er1List:
  if dic["geom"] == None:
    continue
  report.addLine(self,
    self.getDataSet1(),
    self.getDataSet2(),
    dic["ref"].getFeature().getDefaultGeometry(),
    dic["geom"],
    None,
    dic["ref"],
    -1,
    -1,
    False,
    "The dataset 1 polygons dont cover dataset 2 polygons",
    ""
  )
```

En última instancia se recorre la lista de errores y se añaden líneas al reporte de errores con cada elemento de la lista.

(7/8)

```
for feature2 in features2:
  taskStatus.incrementCurrentValue()
  feature2M=convert2D(self, feature2, report)
  checkIntersects=False

  for feature1 in features1:
    feature1M=convert2D(self, feature1, report)
    if feature2M.intersects(feature1M):
      candidateEr2=feature1M.difference(feature2M)
      checkIntersects=True
      if candidateEr2!= None:
        candidateEr2Dic={}
        candidateEr2Dic["geom"]=candidateEr2
        candidateEr2Dic["ref"]=feature1.getReference()
        candidateEr2List.append(candidateEr2Dic)
  if checkIntersects==False:
    candidateEr2Dic={}
    candidateEr2Dic["geom"]=feature2.getDefaultGeometry()
    candidateEr2Dic["ref"]=feature2.getReference()
    er2ListAux.append(candidateEr2Dic)

for dic in candidateEr2List:
  for feature2 in features2:
    feature2M=convert2D(self, feature2, report)
    if dic["geom"] == None:
      continue
    if dic["geom"].intersects(feature2M):
      dic["geom"]=dic["geom"].difference(feature2M)

er2List=[]
for dic in candidateEr2List:
  if dic not in er2List:
    er2List.append(dic)

er2List+=er1ListAux

for dic in er2List:
  if dic["geom"] == None:
    continue
  report.addLine(self,
    self.getDataSet1(),
```

```
          self.getDataSet2(),
          dic["ref"].getFeature().getDefaultGeometry(),
          dic["geom"],
          dic["ref"],
          None,
          -1,
          -1,
          False,
          "The dataset 2 polygons dont cover dataset 1 polygons",
          ""
        )

    except:
      ex = sys.exc_info()[1]
      gvsig.logger("Can't execute rule. Class Name: " + ex.__class__.__name__ + ".
Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

Las líneas anteriores repiten el mismo proceso pero evaluando el conjunto de datos 2 con respecto al conjunto de datos 1

(8/8)

```
def convert2D(self, feature, report):
    polygon= feature.getDefaultGeometry()
    geometryType = polygon.getGeometryType()

    mustConvert2D=(not geometryType.getSubType() == geom.D2)

    proj=polygon.getProjection()
    geomManager = GeometryLocator.getGeometryManager()
    if geomManager.isSubtype(geom.POLYGON,geometryType.getType()):
      if mustConvert2D:
        polygon=polygon.force2D()
        polygon.setProjection(proj)
        return polygon
      return polygon
    elif geomManager.isSubtype(geom.MULTIPOLYGON,geometryType.getType()):
      if mustConvert2D:
        polygon=polygon.force2D()
        polygon.setProjection(proj)
        return polygon
      return polygon
    else:
      report.addLine(self,
        self.getDataSet1(),
        self.getDataSet2(),
```

```
        polygon,
        polygon,
        feature.getReference(),
        None,
        -1,
        -1,
        False,
        "Unsupported geometry type.",
        ""
    )
```

La función convert2D trasforma las coordenadas de las geometrías a 2D. En esta función se comprueba y en el caso de error se reporta si los polígonos presentan tipos de geometrías no válidas.

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

5.6.2.2. Ficheros de acciones correctoras

Como se ve en la documentación de la regla topológica, ésta contiene dos acciones correctoras; deletePolygonAction y createPolygonAction. A continuación, se explica la segunda ya que la primera acción es prácticamente igual que la deletePointAction de la regla ejemplo explicada en el Apartado ***5.3.2.2 Fichero de acciones correctoras.***

createPolygonAction.py

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
import gvsig
import sys

from org.gvsig.topology.lib.spi import AbstractTopologyRuleAction
from org.gvsig.fmap.geom import GeometryLocator
from gvsig import geom
```

```
class CreatePolygonAction(AbstractTopologyRuleAction):
```

Definición de la clase *CreatePolygonAction* que implementa *AbstractTopologyRuleAction* que a su vez implementa *TopologyRuleAction*, clases que definen una acción correctora de una regla topológica en gvSIG Desktop.

```
    def __init__(self):
        AbstractTopologyRuleAction.__init__(
            self,
```

```
            "MustCoverEachOtherPolygon",
            "CreatePolygonAction",
            "Create Polygon Action",
            "This action creates a new dataset 1 feature on not overlapping part."
        )
```

Constructor de la clase *CreatePolygonAction*, se inicializa con los valores del nombre de la regla donde se aplica, nombre de la acción, etiqueta, descripción.

```
    def execute(self, rule, line, parameters):
      try:
        dataSet1 = rule.getDataSet1()
        dataSet2 = rule.getDataSet2()

        store1=dataSet1.getFeatureStore()
        store2=dataSet2.getFeatureStore()

        geomEr=line.getError()

        if line.getFeature1()!=None:
          newFeature=store2.createNewFeature()
          newFeature.setDefaultGeometry(geomEr)
          dataSet2.insert(newFeature)

        if line.getFeature2()!=None:
          newFeature=store1.createNewFeature()
          newFeature.setDefaultGeometry(geomEr)
          dataSet1.insert(newFeature)

      except:
        ex = sys.exc_info()[1]
        gvsig.logger("Can't execute action. Class Name: " + ex.__class__.__name__
+ ". Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

Función *execute* que sobrescríbela función con el mismo nombre en la clase abstracta *AbstractTopologyRuleAction*. Esta función necesita como parámetros de entrada la regla, la línea del reporte de errores que almacena el elemento con error y unos parámetros, que para esta acción contiene el valor *None*.

El código utiliza una estructura *try-except*, para capturar errores, obtiene el elemento de la línea del reporte de errores y dependiendo del conjunto de datos que presenta el error introduce en el dataset un nuevo polígono con geometría igual al error.

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

El resto de scripts de código son muy parecidos a los definidos en la regla ejemplo o alguna de las anteriores, por lo que no se detallan para no sobrecargar el documento.

5.6.3. Testing

En este apartado se menciona el contenido de la carpeta *Testing* para la regla *"Deben cubrirse unos polígonos a otros" (Must cover each other)*.

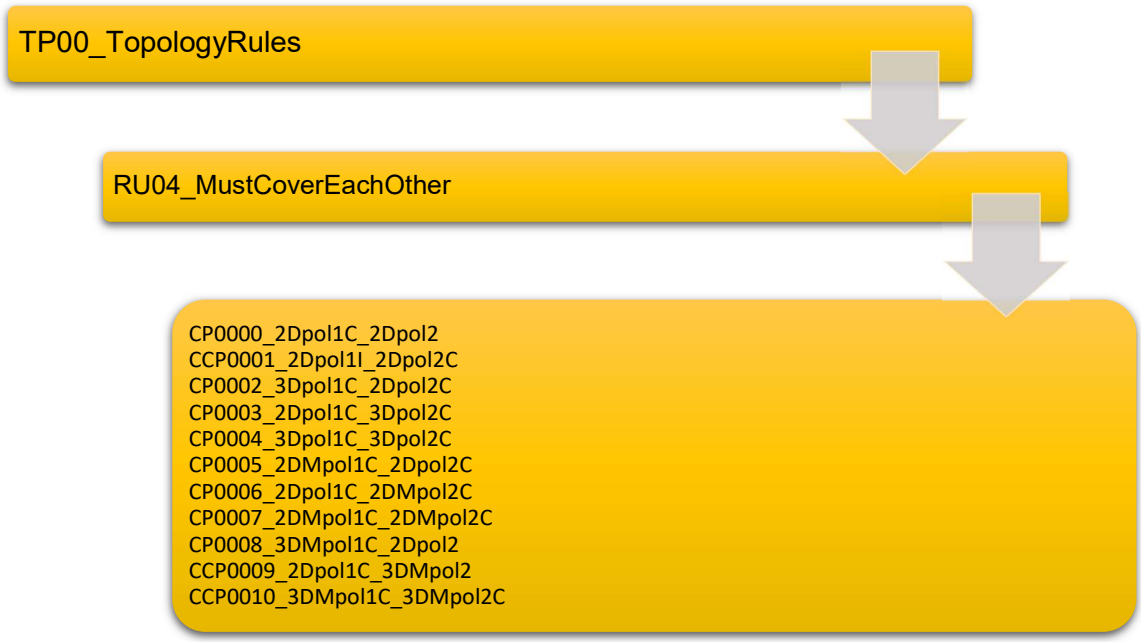


Figura 15. Esquema de estructura de la carpeta Casos de la regla topológica "Deben cubrirse unos polígonos a otros" (Must cover each other)

Tal y como se aprecia al final de la estructura de carpetas anterior, Figura 15, se almacenan once casos de prueba, los cuales están compuestos por un documento MD y dos capas de datos en formato CSV. Los casos evalúan el código según una serie de supuestos listados en la Tabla 4;

Tabla 4. Tabla con los casos de prueba de la regla "Deben cubrirse unos polígonos a otros" (Must cover each other) y explicación de lo que evalúan.

Caso de prueba	Descripción
CP0000_2Dpol1C_2Dpol2C	Evalúa una capa de polígonos 2D **sin** errores con una capa de polígonos 2D **sin** errores.
CP0001_2Dpol1I_2Dpol2C	Evalúa una capa de polígonos 2D **con** errores con una capa de polígonos 2D **sin** errores.
CP0002_3Dpol1C_2Dpol2C	Evalúa una capa de polígonos 3D **sin** errores con una capa de polígonos 2D **sin** errores.

CP0003_2Dpol1C_3Dpol2C	Evalúa una capa de polígonos 2D **sin** errores con una capa de polígonos 3D **sin** errores.
CP0004_3Dpol1C_3Dpol2C	Evalúa una capa de polígonos 3D **sin** errores con una capa de polígonos 3D **sin** errores.
CP0005_2DMpol1C_2Dpol2C	Evalúa una capa de polígonos 2DM **sin** errores con una capa de polígonos 2D **sin** errores.
CP0006_2Dpol1C_2DMpol2C	Evalúa una capa de polígonos 2D **sin** errores con una capa de polígonos 2DM **sin** errores.
CP0007_2DMpol1C_2DMpol2C	Evalúa una capa de polígonos 2DM **sin** errores con una capa de polígonos 2DM **sin** errores.
CP0008_3DMpol1C_2Dpol2C	Evalúa una capa de polígonos 3DM **sin** errores con una capa de polígonos 2D **sin** errores.
CP0009_2Dpol1C_3DMpol2C	Evalúa una capa de polígonos 2D **sin** errores con una capa de polígonos 3DM **sin** errores.
CP0010_3DMpol1C_3DMpol2C	Evalúa una capa de puntos 3DM **sin** errores con una capa de polígonos 3DM **sin** errores.

Los casos de prueba presentan siempre la misma estructura sea la herramienta que sea por lo que no se exponen ninguno de la regla en cuestión, evitando así sobrecargar el documento.

5.7. Regla definida por el usuario (UDR o User defined rule)

En cuanto a la regla topológica *"Regla definida por el usuario" (UDR o User defined rule)* únicamente se destacan las partes que difieren de manera representativa con respecto la regla ejemplo.

Pueden consultarse los archivos originales en el siguiente enlace web:

https://github.com/jolicar/TopologyRuleUserDefinedRule

5.7.1. Consideraciones previas

Antes de continuar con la estructura de datos utilizada para explicar la regla, conviene destacar la implementación previa a su definición. Para su desarrollo se realizaron modificaciones en el motor de topología de gvSIG Desktop concretamente en las clases *TopologyRuleFactory.java*, *TopologyRule.java* y *CreateRuleDialog.java*.

En la clase *TopologyRuleFactory* se añadieron los métodos *createRuleParameters* y *hasRuleParameters*. En la clase *TopologyRule* se añadieron los métodos setParameter y getParameter. Por último, en la clase *CreateRuleDialog* se realizaron modificaciones de modo que, si la regla tiene parámetros, el usuario pudiera introducirlos.

5.7.2. Documentación

Regla definida por el usuario (UDR o User defined rule)

Tipo de regla: Cualquier tipo.

Conjunto de datos primario: Cualquier dato (2D, 2DM, 3D y 3DM) (Multigeometrías permitidas)

Descripción: Esta regla permite al usuario definir una expresión de evaluación y una expresión de corrección propia para la regla. Los elementos de la fuente de datos serán evaluados por la expresión correspondiente proporcionando un valor booleano si cumplen o no ésta. Si el valor es *True*, el elemento geométrico (feature) cumple la regla y, si el valor es *False*, no la cumple y se registra en una entrada en el inspector de errores del plan de topología. Una vez terminada la evaluación en la ventana de reporte de errores podemos decidir qué acción aplicar, eliminar el elemento o la formulada con la expresión de corrección.

Limitaciones: Las expresiones de evaluación y corrección deben estar definidas en el lenguaje de programación propio de gvSIG Desktop llamado cosa.

Acciones correctoras:

Eliminar: La acción elimina los elementos que no cumplen la regla topológica.

Expresión de corrección del usuario: Puede ser diferente para cada trabajo.

5.7.3. Código de programación

5.7.3.1. Ficheros propios de la regla

userDefinedRuleFactory.py

```
import gvsig
import sys

from gvsig import uselib
uselib.use_plugin("org.gvsig.topology.app.mainplugin")

from org.gvsig.fmap.geom import Geometry
from org.gvsig.tools.util import ListBuilder
from org.gvsig.topology.lib.api import TopologyLocator
from org.gvsig.topology.lib.spi import AbstractTopologyRuleFactory,
RuleResourceLoaderUtils
from org.gvsig.tools import ToolsLocator

from java.io import File

from userDefinedRule import UserDefinedRule
```

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
class UserDefinedRuleFactory(AbstractTopologyRuleFactory):
```

Definición de la clase *UserDefinedRuleFactory* que implementa *AbstractTopologyRuleFactory* que a su vez implementa *TopologyRuleFactory*, clases que definen una factoría de una regla topológica en gvSIG Desktop.

```
    def __init__(self):
        AbstractTopologyRuleFactory.__init__(
            self,
            "UserDefinedRule",
            "User defined rule GSoC2020",
            "This rule allows the user to define a data check expression and
another expression to correct it. The dataset features are evaluated for the check
expression and it gives a boolean. If the result is False,the feature dont fulfill
the rule and it creates a entry in the error report. If the result is True, the
feature fulfills the expression. On Inspector de errores del Plan de topologia
window we can remove the feature or apply the corrective expression. \n\n The
check and corrective expression are create on gvSIG Cosa languaje.",
            None
        )
```

Constructor de la clase *UserDefinedRuleFactory*, se inicializa con los valores del nombre de la regla, etiqueta, descripción y los tipos de datos que necesita. En este caso *None*.

```
        pathName = gvsig.getResource(__file__,'UserDefinedRule.json')
        url = File(pathName).toURL()
        gvsig.logger(str(url))
        json = RuleResourceLoaderUtils.getRule(url)
        self.load_from_resource(url, json)
```

Bloque de líneas que en primer lugar obtiene el fichero auxiliar *UserDefinedRule.json* y lo cargan en el área de descripción gráfica y alfanumérica de la ventana *'Añadir regla'*.

```
        dynObjectManager = ToolsLocator.getDynObjectManager()
        self.parametersDefinition =
dynObjectManager.createDynClass("UserDefinedRuleParameters",
"UserDefinedRuleParameters")
self.parametersDefinition.addDynFieldString("CheckExpression").setLabel("Check
expression").setDescription("")
self.parametersDefinition.addDynFieldString("CorrectiveExpression").setLabel("Corr
ective expression").setDescription("")
```

Bloque de definición de parámetros. En este conjunto de líneas de código se crean los parámetros y se definen. Los parámetros son dos, la expresión de evaluación y la expresión de corrección.

```
    def createRule(self, plan, dataSet1, dataSet2, tolerance):
        rule = UserDefinedRule(plan, self, tolerance, dataSet1)
        return rule
```

En este apartado se sobrescribe el método *createRule* de la clase abstracta *AbstractTopologyRuleFactory* con los datos necesarios para crear la regla topológica *UserDefinedRule*. Este método se ejecuta al crear el plan de topología, proceso detallado en el apartado ***3.2 Topología en gvSIG***.

```
    def hasRuleParameters(self):
        return True

    def createRuleParameters(self):
        dynObjectManager = ToolsLocator.getDynObjectManager()
        parameters=dynObjectManager.createDynObject(self.parametersDefinition)
        return parameters
```

Las funciones *hasRuleParameters* y *createRuleParameters* sobrescriben los métodos *con* el mismo nombre de la clase abstracta *AbstractTopologyRuleFactory*. La primera indica que la regla tiene parámetros y la segunda crea los parámetros gracias a la definición de éstos en el bloque de definición de parámetros.

```
def selfRegister():
    try:
        manager = TopologyLocator.getTopologyManager()
        manager.addRuleFactories(UserDefinedRuleFactory())
    except:
        ex = sys.exc_info()[1]
        gvsig.logger("Can't register rule. Class Name: " + ex.__class__.__name__ +
". Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

La función *selfRegister* registra la factoría de la regla en cuestión en el marco de topología. Para hacer lo anterior se basa en una estructura de *try-except*, recogiendo el error en el caso de existir.

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

userDefinedRule.py

```
import gvsig
import sys

from gvsig import uselib
uselib.use_plugin("org.gvsig.topology.app.mainplugin")
```

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
from org.gvsig.expressionevaluator import GeometryExpressionEvaluatorLocator,
ExpressionEvaluatorLocator
from org.gvsig.topology.lib.api import TopologyLocator
from org.gvsig.topology.lib.spi import AbstractTopologyRule

from org.gvsig.expressionevaluator import ExpressionUtils
from org.gvsig.fmap.dal import DALLocator

from deleteFeatureAction import DeleteFeatureAction
from correctiveExpressionAction import CorrectiveExpressionAction
```

```
class UserDefinedRule(AbstractTopologyRule):
```

Definición de la clase *UserDefinedRule* que implementa *AbstractTopologyRule* que a su vez implementa *TopologyRule*, clases que definen una regla topológica en gvSIG Desktop.

```
  def __init__(self, plan, factory, tolerance, dataSet1):
      AbstractTopologyRule.__init__(self, plan, factory, tolerance, dataSet1)
      self.addAction(DeleteFeatureAction())
      self.addAction(CorrectiveExpressionAction())

      self.checkExpression=None
      self.fst=None
```

Constructor de la clase *UserDefinedRule*, se inicializa con los valores del plan de topología, la factoría de la regla, la tolerancia especificada y las fuentes de datos que necesita. En este apartado se inicializan algunas variables como el constructor de expresiones, *expressionBuilder* y se añaden las acciones correctoras a la regla, en este caso la acción Eliminar, *DeleteFeatureAction y la expresión de corrección del usuario*, *CorrectiveExpressionAction*.

```
  def check(self, taskStatus, report, feature1):
      try:
          if self.checkExpression==None:
              formula = self.getParameters().getDynValue("CheckExpression")
              self.checkExpression = ExpressionUtils.createExpression(formula)
              datamanager = DALLocator.getDataManager()
              self.fst = datamanager.createFeatureSymbolTable()
          self.fst.setFeature(feature1)
          x=self.checkExpression.execute(self.fst)
          if not x:
              report.addLine(self,
                self.getDataSet1(),
                None,
                feature1.getDefaultGeometry(),
```

```
                feature1.getDefaultGeometry(),
                feature1.getReference(),
                None,
                -1,
                -1,
                False,
                "This feature does False the Check Expression",
                ""
            )
        except:
            ex = sys.exc_info()[1]
            gvsig.logger("Can't execute rule. Class Name: " + ex.__class__.__name__
+ ". Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

La función *check* sobrescribe a la función con el mismo nombre de la clase AbstractTopologyRule. La funcionalidad de esta función consiste en evaluar los elementos con respecto a la expresión de evaluación. El código ejecuta la citada expresión sobre los elementos, si devuelve *False* se registra una línea en el inspector de errores del plan de topología y si devuelve True, el elemento cumple la expresión de evaluación y por tanto la regla.

5.7.3.2. Ficheros de acciones correctoras

Como se ve en la documentación la regla topológica, ésta contiene dos acciones correctoras; *deleteFeatueAction* y *correctiveExpressionAction*. A continuación, se explica la segunda ya que la primera acción es prácticamente igual que la deletePointAction de la regla ejemplo explicada en el Apartado ***5.3.2.2 Fichero de acciones correctoras.***

correctiveExpressionAction.py

Bloque con todos los 'imports' necesarios para la ejecución del código del programa

```
import gvsig
import sys

from org.gvsig.topology.lib.spi import AbstractTopologyRuleAction

from org.gvsig.expressionevaluator import ExpressionUtils
from org.gvsig.fmap.dal import DALLocator
```

```
class CorrectiveExpressionAction(AbstractTopologyRuleAction):
```

Definición de la clase *CorrectiveExpressionAction* que implementa Abstract*TopologyRuleAction* que a su vez implementa *TopologyRuleAction*, clases que definen una acción correctora de una regla topológica en gvSIG Desktop.

```
    def __init__(self):
        AbstractTopologyRuleAction.__init__(
```

```
            self,
            "userDefinedRule",
            "CorrectiveExpressionAction",
            "Corrective Expression Action",
            "This action uses a user corrective expression for cases when User
Defined rule Topology Rule it is false. This action will be different on each
work"
        )
        self.correcExpression=None
        self.fst=None
```

Constructor de la clase *CorrectiveExpressionAction*, se inicializa con los valores del nombre de la regla donde se aplica, nombre de la acción, etiqueta, descripción. También se inicializan en este apartado algunas variables.

```
    def execute(self, rule, line, parameters):
        try:
          if self.correcExpression==None:
            formula = rule.getParameters().getDynValue("CorrectiveExpression")
            if formula==None or formula=="":
                pass
            self.correcExpression = ExpressionUtils.createExpression(formula)
            datamanager = DALLocator.getDataManager()
            self.fst = datamanager.createFeatureSymbolTable()
            self.fst.setFeature(line.getFeature1().getFeature())
            self.correcExpression.execute(self.fst)
        except:
            ex = sys.exc_info()[1]
            gvsig.logger("Can't execute action. Class Name: " +
ex.__class__.__name__ + ". Exception: " + str(ex), gvsig.LOGGER_ERROR)
```

Función *execute* que sobrescríbela función con el mismo nombre en la clase abstracta *AbstractTopologyRuleAction.* Esta función necesita como parámetros de entrada la regla, la línea del reporte de errores que almacena el elemento con error y unos parámetros, que para esta acción contiene el valor *None*.

El código presenta una estructura *try-except*, para capturar errores, obtiene el elemento de la línea del reporte de errores y ejecuta la expresión de corrección del usuario sobre él.

```
def main(*args):
    pass
```

Programa principal del script. No realiza nada.

5.7.4. Testing

En este apartado se menciona el contenido de la carpeta *Testing* para la regla *"Regla definida por el usuario" (UDR o User defined rule).*

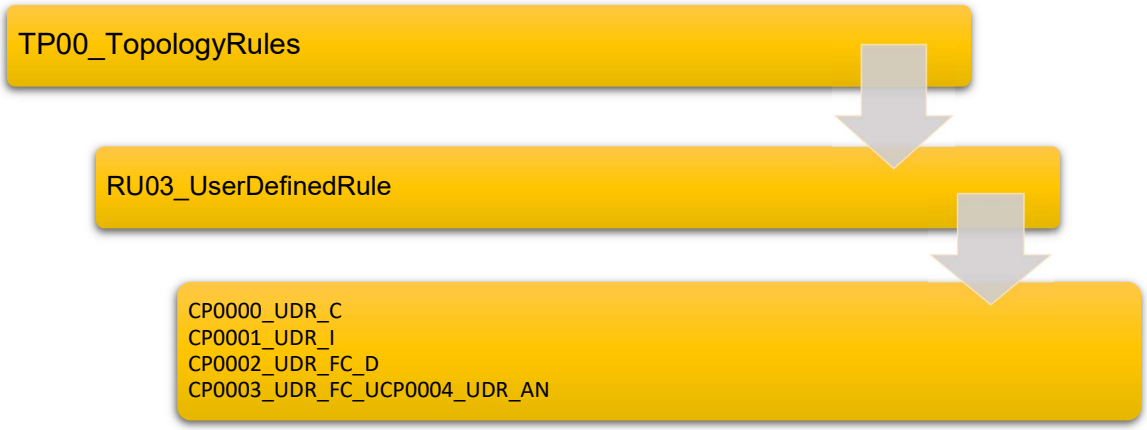


Figura 16. Esquema de estructura de la carpeta Casos de la regla "Regla definida por el usuario" (UDR o User defined rule)

Tal y como se aprecia al final de la estructura de carpetas anterior, Figura 16, se almacenan cuatro casos de prueba, los cuales están compuestos por un documento MD y dos capas de datos en formato CSV. Los casos evalúan el código según una serie de supuestos listados en la Tabla 5;

Tabla 5. Tabla con los casos de prueba de la regla "Regla definida por el usuario" (UDR o User defined rule) y explicación de lo que evalúan.

Caso de prueba	Descripción
CP0000_UDR_C	Evalúa una capa de polígonos 2D **sin** errores con una capa de polígonos 2D **sin** errores respecto a una expresión de evaluación.
CP0001_UDR_I	Evalúa una capa de polígonos 2D **con** errores con una capa de polígonos 2D **sin** errores respecto a una expresión de evaluación.
CP0002_UDR_FC_D	Evalúa la acción correctora de eliminar elementos.
CP0003_UDR_FC_U	Evalúa la acción correctora resultante al definir el usuario una expresión correctora.
CP0004_UDR_AN	Evalúa un dataset de datos alfanuméricos respecto a una expresión de evaluación.

Los casos de prueba presentan siempre la misma estructura sea la herramienta que sea por lo que no se exponen ninguno de la regla en cuestión, evitando así sobrecargar el documento.

6. Conclusiones

El desarrollo del presente proyecto titulado *"Nuevas reglas para el marco topológico de gvSIG Desktop"* ha cubierto los objetivos planteados inicialmente y permite plantear un conjunto de conclusiones que se exponen a continuación.

El contenido principal del Trabajo Fin de Máster ha consistido en la definición, implementación y testeo de un conjunto de cinco reglas topológicas, así como la elaboración de documentación, plan de pruebas y acciones correctoras para dichas reglas, lo cual ha supuesto una magnífica oportunidad de aplicar los conceptos estudiados en el Máster Universitario en Ingeniería Geomática y Geoinformación (MUIGG).

El desarrollo se ha llevado a cabo en estrecha colaboración con la Asociación gvSIG, y con los creadores del propio sistema gvSIG. Esta experiencia directa con los compañeros de la Asociación ha sido enormemente enriquecedora y ha permitido adquirir una formación muy ligada a las aplicaciones del mundo real que complementan la formación académica recibida en el MUIGG.

Usar como base el proyecto con el mismo nombre realizado en el Google Summer of Code (GSOC) 2020 ha aportado un carácter distintivo al trabajo, ya que ha permitido aplicar un conjunto de conocimientos teórico-prácticos que van más allá del objetivo académico. Ha sido necesario adquirir una serie de habilidades que cubren los requerimientos y expectativas académicas de este tipo de trabajos finales de carrera y, además, ha generado un producto destinado a mejorar la comunidad de desarrollo de software libre.

Otra conclusión se deriva del concepto central del trabajo, la topología. La implementación de ésta es necesaria en los sistemas de información geográfica debido al carácter cuantitativo del dato que aporta al software. Esta funcionalidad topológica hace al software más robusto en procesos de filtrado y limpieza de datos lo que mejora, y a veces incluso posibilita, la obtención de resultados. Un mayor número de reglas topológicas hace más eficaz o competitivo al software que las implementa pues dispone de más herramientas que garantizan un dato correcto para el posterior procesamiento.

Es importante destacar también en este apartado la importancia en cualquier desarrollo informático de la estructuración de contenidos de manera estándar a un patrón especificado previamente, así como las partes menos relacionadas con el scripting o codificación. Es decir, la generación de documentación y estructuras que garanticen el correcto funcionamiento del algoritmo (testing). Durante el presente documento se ha puesto énfasis en lo importante y necesaria que es la información que da sentido a un conjunto de scripts, ya que sin esta solo serían conjuntos de líneas de código. Además, con la implementación del conjunto de información almacenado en la sección de testing se ha definido y comprobado un sistema que garantiza el funcionamiento correcto del código en el caso de futuros desarrollos, no solo en el presente.

Como propuestas de mejora al trabajo hay que decir que alguna regla puede mejorar su código, concretamente "Deben cubrirse unos polígonos a otros" (Must cover each other) ya que utiliza diccionarios y listas para almacenar datos internos durante el procesamiento. La utilización de ese tipo de estructuras no es la más acertada para archivos con grandes volúmenes de información ya que sobrecarga en exceso la memoria y baja el rendimiento de la máquina. Otra propuesta de mejora no del proyecto

en sí, sino del marco de topología de gvSIG sería la ampliación de la colección de reglas topológicas lo cual, como se detalla unos párrafos atrás, mejoraría notablemente al sistema en procesos de validación de datos.

En definitiva, con el proyecto GSOC 2020 y el presente trabajo se han definido y desarrollado un conjunto de herramientas que permiten la validación de datos y un sistema de testeo que permite comprobar en el tiempo su funcionamiento. Estas herramientas, junto a las ya existentes aumentan el potencial de gvSIG Desktop. Además, se ha implantado con el trabajo una metodología de desarrollo de este tipo de herramientas la cual funciona y actualmente se sigue usando en otros proyectos de desarrollo de nuevas reglas.

7. Bibliografía

Núñez Valdés, J., Alfonso Pérez, M., Bueno Guillén, S., Diánez del Valle, M. d. R., y De Elías Olivenza, M. C. (2004) *"Siete puentes, un camino: Königsberg"* <https://idus.us.es/bitstream/handle/11441/45044/Siete%20puentes%2c%20un%20camino%20K%c3%b6nigsberg.pdf?sequence=1&isAllowed=y> [Consulta: noviembre de 2020]

Theobald, D. M. (2001). *Topology revisited: representing spatial relations. International Journal of Geographical Information Science*, 15-8, 689-705, DOI: 10.1080/13658810110074519. [Consulta: noviembre de 2020]

Esri. *Understanding Topology and Shapefiles.* <*https://www.esri.com/news/arcuser/0401/topo.html*> [Consulta: noviembre de 2020]

Esri. *Reglas topológicas de las geodatabases y soluciones a los errores de topología. <https://desktop.arcgis.com/es/arcmap/10.3/manage-data/editing-topology/geodatabase-topology-rules-and-topology-error-fixes.htm>* [Consulta: noviembre de 2020]

Asociación gvSIG. *Documentación de Scripting en gvSIG 2.3*. <http://downloads.gvsig.org/download/web/es/build/html/scripting_devel_guide/2.4/trabajando_con_esquemas.html> [Consulta: noviembre de 2020]

Google. *Google summer of code.* <https://summerofcode.withgoogle.com/> [Consulta: noviembre de 2020]

Olivas Carriquí, J (2020) *"Elaboración de casos de prueba en gvSIG Desktop"* en gvSIG Blog, 18 de septiembre. <https://blog.gvsig.org/2020/09/18/elaboracion-de-casos-de-prueba-en-gvsig-desktop/> [Consulta: diciembre de 2020]