

PROJECTE FINAL DE CARRERA

REPRESENTACIÓ GRÀFICA D'ONTOLOGIES

Albert Bernaus Carmona
ETIG

Directors del projecte:
Sr. David Sánchez Ruenes
Dr. Antonio Moreno Ribas

Escola Tècnica Superior d'Enginyeria (ETSE)
Universitat Rovira i Virgili (URV)

INDEX

1- INTRODUCCIO.....	1
2- DESCRIPCIÓ DE LA TECNOLOGÍA.....	5
2.1 Ontologies.....	5
2.2 Tecnologia OWL.....	7
2.2.1 Representació de coneixement.....	9
2.2.2 Mecanismes de raonament.....	12
2.3 Requisits.....	13
2.4 Descripció de les alternatives.....	15
2.4.1 GraphViz (integrat).....	18
2.4.2 OntoRama (tipus hiperbòlic).....	20
2.4.3 TouchGraph (tipus hiperbòlic).....	22
2.4.4 Aduna ClusterMap Library 2005.1 (tipus clustering).....	24
2.5 JGraph (simple).....	29
3- ESPECIFICACIÓ DEL PROJECTE.....	37
3.1 Introducció.....	37
3.2 Especificació dels fitxers OWL.....	40
3.3 Lectura de les ontologies.....	43
3.4 Estructura de dades.....	48
3.5 Visualització.....	51
3.6 Tecnologia Web.....	56
4- JOC DE PROVES.....	59
4.1 Prova 1: Exploració taxonòmica.....	59
4.2 Prova 2: Filtrar informació taxonòmica.....	61

4.3 Prova 3: Exploració no taxonòmica.....	62
4.4 Prova 4: Filtrar informació no taxonòmica.....	65
4.5 Prova 5: Classes mestres.....	66
4.6 Prova 6: Responent als events del ratolí.....	67
4.7 Prova 7: Zooms.....	71
4.8 Prova 8: Exploració de fitxers CLEAN.....	72
4.9 Prova 9: Respostes del ratolí en fitxers CLEAN.....	73
4.10 Prova 10: Captura del graf.....	74
4.11 Prova 11: Integració web.....	75
5- CONCLUSIONS.....	77
6- REFERENCIES.....	79

1. INTRODUCCIO

Amb el desenvolupament de la societat de la informació, la quantitat de coneixement disponible ha augmentat molt en els darrers anys. Aquest coneixement s'expressa típicament en llenguatge natural i no pot ser interpretat directament per un ordinador.

Per tal de donar accés al coneixement d'un domini als processos computacionals, s'utilitzen les ontologies.

Per tal de definir una aproximació semàntica per classificar i estructurar el coneixement d'un domini, es fa necessari l'ús d'ontologies.

Una ontologia és podria definir com la representació o conceptualització del coneixement d'un domini concret totalment estructurat. Les ontologies juguen un paper central en el desenvolupament i desplegament de la *Semantic Web* (www.w3.org/2001/sw). Són usades per diferents propòsits com ara per estendre's a través de la localització de la informació, integració, consulta, presentació i navegació de coneixements. Per compartir la comprensió comuna de l'estructura de la informació entre la gent i entre agents del programari, per a fer assumpcions del domini explícites i també per permetre la reutilització del coneixement del domini per evitar "reinventar la roda" i per a introduir estàndards per a permetre la interoperabilitat.

Existeixen diversos llenguatges de representació d'ontologies, usarem un dels més nous, per tant, aquestes ontologies es representaran amb un nou llenguatge estàndard anomenat OWL (Ontology Web Language) [1]. OWL és un estàndard derivat de XML i creat per W3C per construir, representar i tractar ontologies, i és específic per representar ontologies especialment dissenyat per aportar coneixement semàntic als recursos Web.

Les ontologies amb les que treballarem són les que ens proporciona una eina que crear ontologies a partir de la Web i a les quals, a part de contenir tota la informació bàsica que tota ontologia ha de tenir, afegeix tota una sèrie de meta-informació molt interessant sobre els conceptes de l'ontologia que no es contempla a l'estàndard OWL, però també dades sobre la búsqueda que ha realitzat per desenvolupar la ontologia. Per tant, el

nostre visualitzador no podrà llegir qualsevol fitxer OWL perquè serà creat específicament per visualitzar les ontologies que ens proporciona l'eina de desenvolupament. Aquesta eina de desenvolupament d'ontologies és una nova versió del projecte **Sistema Multi-Agente para la construcción semiautomática de Ontologías a partir de la Web** [2] de Lorena Morales i Francisco Ruiz, creat l'any passat al GruSMA.

Actualment tenim aplicacions capaces de mostrar ontologies OWL gràficament i de manera interactiva però totes tenen els seus inconvenients.

Una de les aplicacions més utilitzades per visualitzar ontologies és Protégé [3]. Aquesta aplicació disponible gratuïtament a la xarxa és bastant útil però té algunes mancances i limitacions que ens interessaria resoldre. Per exemple, una d'aquestes limitacions és que Protégé representa la jerarquia de la ontologia però no representa cap tipus de dades d'aquesta. Protégé tampoc ens mostra de cap manera la meta-informació que tenim, com ara, les pàgines webs que s'han trobat sobre cada concepte del domini. A més és molt lent en temps de càrrega, i de seguida que l'ontologia és una mica gran (com potencialment poden ser els resultats de l'eina de creació d'ontologies), la visualització resulta caòtica i la interacció amb l'usuari molt lenta.

Totes aquestes limitacions són degudes a que Protégé no és una aplicació *ad-hoc*. Un instrument *ad-hoc* és una eina creada específicament per a una determinada ocasió o situació (www.wikipedia.com). Generalment s'usa per referir-se a alguna cosa que s'ha adequat només per a una determinada finalitat. Protégé és una aplicació de caràcter general per visualitzar qualsevol ontologia creada a base de plug-ins. Es va crear per visualitzar, editar i desenvolupar ontologies i per tant segueix un estàndard que és OWL.

Amb aquesta aplicació el que es pretén es mostrar el màxim d'informació útil amb aspectes visuals aconseguint així una aplicació amb una visualització rica. Així, a banda de mostrar els valors dels atributs també es pretén representa'ls visualment d'alguna manera perquè a primer cop d'ull l'usuari es pugui fer una idea de quins poden ser aquests valors. Per exemple, alguns dels recursos utilitzats són els diferents tipus de formes geomètriques, utilització de colors i també diferents tipus de lletres, inserció d'icones i símbols, etc. Tota la interfície implementada s'explicarà amb més profunditat

en posteriors apartat.

Per tant, aquest projecte el que es pretén es visualitzar de forma gràfica, còmoda i interactiva una ontologia generada a partir de la Web, és a dir, els resultat d'una cerca d'un coneixement d'un determinat domini totalment estructurat i de fàcil accés a totes pàgines webs per facilitar a l'usuari l'obtenció d'informació de la Web.

Finalment ens marquem uns objectius molt clars per complir. Aquest objectius són:

1. Llegir tota la informació dels fitxers OWL creats pel constructor d'ontologies, incloent tota la meta-informació.
2. Definir un disseny d'aplicació que permeti un bon temps de resposta quan treballem amb grans ontologies.
3. Utilitzar aquesta meta-informació per aconseguir una visualització rica.
4. Oferir flexibilitat amb la interacció amb l'usuari per controlar la visualització en tot moment.
5. Integrar el visualitzador en un entorn web per tal d'integrar-lo amb el projecte **Sistema Multi-Agente para la construcción semiautomática de Ontologías a partir de la Web.**

Organització de l'informe

En aquest document tractarem d'explicar com s'ha dut a terme l'aplicació creada així com els problemes que hem trobat i que s'han pogut resoldre o no i quin ha estat el resultat final, si s'ha aconseguit crear l'aplicació que se'ns ajusta més a les nostres necessitats i si es competitiu, i realment ha estat profitós la realització d'aquest projecte, amb les aplicacions que ja existeixen amb la mateixa finalitat.

L'estructura del document és la següent: al segon capítol s'explica detalladament la tecnologia utilitzada per a realitzar aquest projecte, els requisits que han de tenir les llibreries de visualització gràfica que usarem i una descripció i comparativa de les diferents alternatives que s'han trobat amb la que finalment s'ha triat. Finalment en

aquest capítol també hi ha un petit manual de la tecnologia triada per la representació gràfica així com de l'estàndard OWL per a la creació d'ontologies.

Al tercer capítol s'explica amb detall l'aplicació que s'ha creat, tota la lògica i les utilitats que aquesta presenta, com per exemple la possibilitat d'aplicar filtres en l'exploració o la possibilitat de fer un captura de l'estat del gràfic en un moment determinat entre d'altre.

Al quart capítol es presenta un exhaustiu joc de proves que demostrarà la real utilitat i funcionalitat d'aquest projecte, mostrant totes les opcions que podem realitzar amb el gràfic creat.

Al cinquè capítol tenim l'apartat de les conclusions, constituït per un resum, els avantatges i inconvenient, les aplicacions i els futurs treballs basat en l'aplicació creada.

2. DESCRIPCIO DE LA TECNOLOGIA

Per tal de crear el visualitzador explorarem la Web en busca d'una llibreria que ens faciliti la creació d'aquesta aplicació ja que seria molt costós crear l'aplicació des de zero mitjançant les primitives bàsiques de dibuix que ens proporciona els paquets estàndards de Java com ara la AWT o la SWING entre d'altres.

Bàsicament escollirem una llibreria gràfica escrita en Java que ens proporcionï funcions per pintar els diferents element que ens representen l'ontologia i funcions bàsiques com *Drag&Drop*, zooms, etc.

Aquests elements que pintarem són classes, instàncies i diferents tipus de fletxes per representar les relacions taxonòmiques i no taxonòmiques. Tots aquests conceptes s'expliquen a continuació.

En aquest apartat es profunditza en les tecnologies que hem trobat i la que hem triat, què tenen que no tinguin altres i perquè s'ha triat la escollida.

S'han utilitzat bàsicament dues tecnologies: una llibreria gràfica que ens ajudi en la representació gràfica de les ontologies i un llenguatge per llegir les ontologies abans de plasmar-les gràficament a la pantalla.

2.1 Ontologies

Una classe és un concepte del domini, per exemple *cancer* és una classe i *cancer de pulmó* també és una altra classe del mateix domini.

Les instàncies (Sánchez, 2005) de conceptes, són classe-entitat, és a dir, la concretització d'un concepte en una entitat pròpia (entitats, persones, companyies, esdeveniments, etc), com per exemple “Asociación Nacional De Lucha Contra El Cáncer” podria considerar-se una instància de “càncer”.

Les classe es poden relacionar taxonòmicament amb altres classes i no taxonòmicament amb altres classes o instàncies del mateix o de diferent dominis.

Les relacions taxonòmiques representaran la taxonomia de la ontologia. La taxonomia és la ciència de la classificació, estructurant un domini per taxons que descriuen jeràrquicament les relacions, de similitud, parentiu o qualsevol altre criteri, entre les entitats d'aquest domini.

Una relació no taxonòmica és aquella que relaciona dues classes entre si amb una etiqueta semàntica (típicament un verb), és a dir, ens diu de quina manera estan relacionades, que tenen que les lliga. Un exemple clar per entendre això és el següent: “*càncer rep radioteràpia*”. En aquest exemple les dues classes relacionades són *càncer* i *radioteràpia* i estan lligades pel verb *rep* indicant-nos que el càncer necessita radioteràpia per poder tractar-lo.

Una ontologia esta formada per diferents components. Aquests components són els conceptes, les propietat i atributs que poden tenir aquests conceptes, restriccions sobre aquestes propietats i atributs i individus.

En el següent quadre es pot veure un exemple de tots aquests components (fig. 2.1).

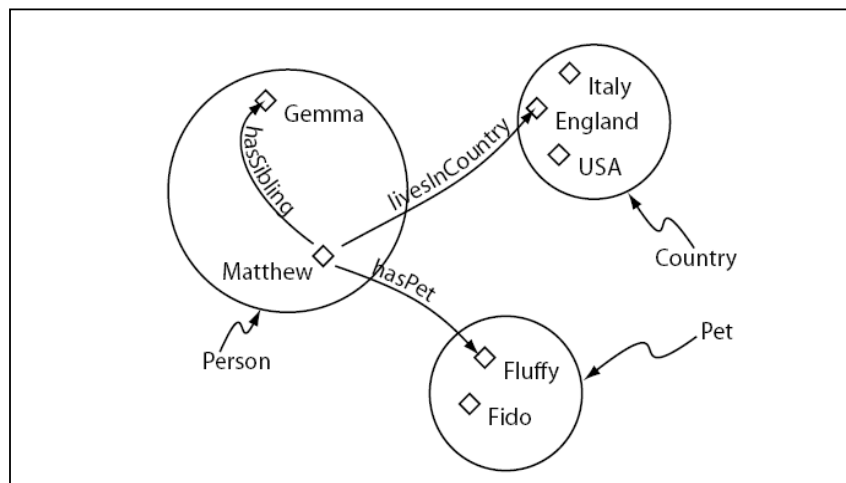


Fig. 2.1: Components de les Ontologies

Person, *Pet* i *Country* són conceptes. Cada concepte té alguns individus, en aquest cas, *Gemma* i *Matthew* són individus del concepte *Person*, *Italy*, *England*, *USA* són individus de *Country* i *Fluffy* i *Fido* ho són de *Pet*. Després també tenim propietats i atributs com ho són *hasPet*, *hasSibling* i *livesInCountry*, i per últim podríem tenir una restricció que fos: *Persons sempre* viuen en un *Country*.

2.2 Tecnologia OWL

Com hem comentat anteriorment, OWL (Web Ontology Language) és el resultat del treball del W3C Web Ontology (WebOnt) Working Group, que es va formar al novembre de 2001. Aquest llenguatge deriva de DAML+OIL i ho suplanta. Engloba la majoria de les característiques de DAML+OIL i renombra la majoria de les seves primitives. Com els llenguatges previs, OWL va dirigit a publicar i compartir ontologies a la Web. Al febrer de 2004 el W3C va anunciar l'aprovació de OWL, que es va convertir en una W3C Recommendation. Això va marcar l'aparició de *The Semantic Web* com una plataforma de grau comercial per a informació en la Web. El desplegament d'aquest estàndard en productes i serveis comercials marca la transició de la tecnologia de *The Semantic Web* del que va ser en gran part, un projecte d'investigació i desenvolupament avançat durant els últims cinc anys, a una tecnologia més pràctica desplegada en eines de mercat comú que permeten un accés més flexible a la informació estructurada de la Web.

A la figura 2.2 es mostra els diferents llenguatges per d'ontologies que existeixen:

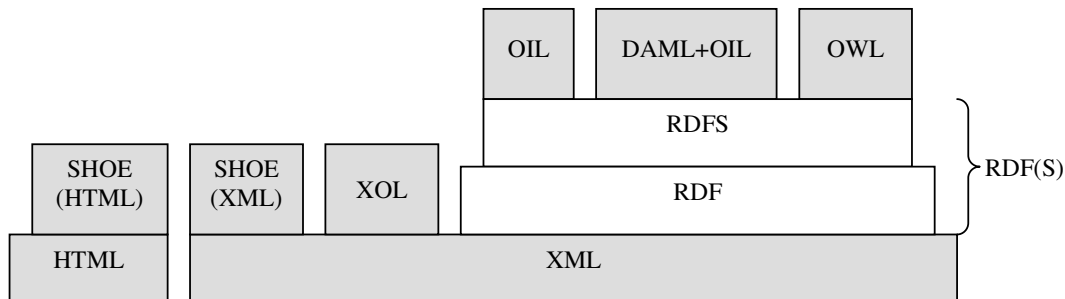


Fig. 2.2: Llenguatges d'etiquetes d'ontologies

OWL està dividit en capes: OWL Lite, OWL DL i OWL Full. OWL Lite estén RDF(S) i reuneix les característiques més comunes de OWL, per tant, està destinat a usuaris que només necessiten crear taxonomies de classes i restriccions simples. OWL DL inclou el vocabulari complet de OWL. Finalment, OWL Full proporciona més flexibilitat per a representar ontologies que OWL DL.

Igual que DAML+OIL, OWL està construït sobre RDF(S). Per tant, algunes primitives de RDF(S) són reutilitzades per OWL, i les ontologies de OWL estan escrites en XML o

amb la notació triple de RDF. Nosaltres usarem la sintaxi de XML per a tots els exemples posteriors.

Com OWL deriva de DAML+OIL, comparteix moltes característiques amb aquest llenguatge. Les principals diferències entre OWL i DAML+OIL són les següents:

- OWL no inclou restriccions numèriques qualificades (`daml:hasClassQ`, `daml:cardinalityQ`, `daml:maxCardinalityQ` i `daml:minCardinalityQ`).
- OWL permet definir propietats simètriques, que no van ser considerades en DAML+OIL, amb la primitiva '`OWL:SymmetricProperty`'.
- OWL no renombra les primitives de RDF(S) reutilitzades pel llenguatge, com passava amb DAML+OIL. Per exemple, `rdfs:subClassOf`, `rdfs:subPropertyOf`.
- En OWL han estat renombrades moltes primitives de DAML+OIL. Per exemple, la primitiva '`daml:toClass`' ha estat renombrada com '`OWL:allValuesFrom`'.
- OWL no inclou la primitiva '`daml:disjointUnionOf`', ja que es pot portar a terme combinant '`OWL:unionOf`' i '`OWL:disjointWith`'.

Són diversos els motius que ens han dut a utilitzar aquest llenguatge per a representar l'ontologia. En primer lloc la llibreria `taxoBuilder` (constructor d'ontologies) ens retorna el resultat de les seves taxonomies descrites en aquest llenguatge, així doncs, amb la condició de ser compatibles amb ella hem preferit no canviar de llenguatge. D'altra banda, la filosofia d'aquest llenguatge és ideal per al nostre propòsit, ja que aquesta pensat per a representar coneixement de recursos Web.

Així doncs, vam utilitzar aquest llenguatge per tractar els resultats de `taxoBuilder` i per a mostrar-los, tenint en compte tots els paràmetres que defineixen aquesta Ontologia (classes, subclasses, frases, instàncies, pàgines Web, etc.).

Tot i que a priori sembla un llenguatge ideal per al nostre sistema, té algunes limitacions derivades de la seva curta evolució i del seu disseny. El problema ve quan volem emmagatzemar dades i aquests contenen certs caràcters (`&`, `<`, `>`) entre les etiquetes, com per exemple les pàgines webs relacionades amb cada classe i relació. Això és perquè el llenguatge OWL no està pensat per a realitzar aquesta tasca.

2.2.1 Representació de coneixement

Una ontologia en OWL comença amb la declaració del node arrel de RDF. En aquest node hem d'incloure els *namespaces* per a les ontologies de Representació de Coneixement de RDF, RDFS i OWL. Si s'utilitzen tipus de dades de XML Schema, pot ser útil incloure un *namespace* per a XML Schema, que es prefixa normalment com 'xds' (i que apunta a la URL més recent de XML Schema, com en RDF(S)) (Prieto, 2004).

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">
```

Com en altres llenguatges, l'ordre en què apareixen les definicions en les ontologies de OWL no és rellevant, ja que es basa en RDF(S). No obstant això, les ontologies de OWL normalment defineixen primer l'encapçalament de l'ontologia i després els termes. L'encapçalament d'una ontologies de OWL pot incloure: la documentació de l'ontologia (rdfs:comment); la versió de l'ontologia (owl:versionInfo); i les ontologies importades (owl:imports). Nosaltres importarem ontologies sobre unitats i tipus de dades. No és necessari (ni recomanat) importar l'ontologia de Representació de Coneixement de OWL. L'encapçalament de l'ontologia també pot incloure informació sobre control de versió amb les primitives 'owl:backwardCompatibleWith', 'owl:incompatibleWith' i 'owl:priorVersion'. Dintre de l'ontologia també podem trobar definicions de classes i propietats *deprecated* amb les primitives 'owl:DeprecatedClass' i 'owl:DeprecatedProperty'.

```
<owl:Ontology rdf:about="">
  <owl:versionInfo>1.0</owl:versionInfo>
  <rdfs:comment>Sample ontology for travel agencies</rdfs:comment>
  <owl:imports rdf:resource="http://delicias.dia.fi.upm.es/owl/units"/>
  <owl:imports rdf:resource="http://www.w3.org/2001/XMLSchema"/>
</owl:Ontology>
```

Els conceptes són coneguts com a classes a OWL, i es creen amb la primitiva 'owl:Class'. A més del seu nom, una classe pot contenir la seva documentació (amb 'rdfs:comment') i qualsevol nombre d'expressions amb primitives.

Aquestes primitives són:

- 'rdf:subClassOf' conté expressions de classe. Permet definir les superclasses de la classe. Això ens permetrà crear les taxonomies.
- 'owl:disjointWith' afirma que la classe no pot compartir instàncies amb l'expressió de classe en aquesta primitiva.
- 'owl:equivalentClass' també conté expressions de classe. Aquesta primitiva defineix condicions necessàries i suficients per a la classe (és a dir, que s'utilitza per a redefinir conceptes ja definits).
- 'owl:oneOf' defineix una classe enumerant exhaustivament totes les seves instàncies. És una forma de definir una classe extensament.
- 'owl:intersectionOf', 'owl:unionOf' i 'owl:complementOf' defineixen una expressió de classe com una conjunció, una disjunció o una negació d'altres expressions de classe respectivament.

Les dues primeres primitives ('rdf:subClassOf' i 'owl:disjointWith') defineixen condicions necessàries per a la classe (es poden utilitzar en la definició de conceptes primitius), mentre que la resta de primitives defineixen condicions necessàries i suficients per a la classe (és a dir, que s'utilitzen per a definir conceptes definits). En OWL Lite, les úniques primitives que es poden utilitzar són: 'rdf:subClassOf', 'owl:equivalentClass' i 'owl:intersectionOf'. En tots els casos, només es poden utilitzar amb identificadors de classe i restriccions de propietat.

D'acord amb la terminologia de Lògica de Descripció (DL), OWL és un llenguatge SHOIN. Això significa que es poden construir expressions de classe amb els següents constructors:

- Conjunt (owl:intersectionOf), disjunció (owl:unionOf) i negació (owl:complementOf) d'expressions de classe.
- Col·leccions d'individuals (owl:oneOf).
- Restriccions de propietat (owl:Restriction). Contenen una referència a la propietat a la qual s'aplica la restricció amb la primitiva 'owl:onProperty' i un element per a expressar la restricció. Es poden aplicar les següents restriccions a les propietats: restricció de valor (owl:allValuesFrom), restricció existencial (owl:someValuesFrom), filtres de rols (owl:hasValue) i restricció de nombre

(owl:cardinality, owl:maxCardinality, owl:minCardinality).

Tot això ens permetrà crear les relacions no taxonòmiques.

- A més, les expressions de rol poden expressar rols inversos (owl:inverseOf) i jerarquies de rols (rdfs:subPropertyOf).

La gramàtica de OWL per a construir expressions de classes és molt similar a la de DAML+OIL. Les expressions de booleans i enumeracions es troben entre les etiquetes '`<owl:Class>..</owl:Class>`'.

```
class-name |
<owl:Class> boolean-expr </owl:Class> |
<owl:Class>
  <owl:oneOf rdf:parseType="Collection">
    instance-expr+
  </owl:oneOf>
</owl:Class> |
<owl:Restriction>
  <owl:onProperty> property-name </owl:onProperty>
  restriction-expr
</owl:Restriction>
```

'boolean-expr' se defineix así:

```
<owl:intersectionOf>
  class-expr class-expr+
</owl:intersectionOf> |
<owl:unionOf> class-expr class-expr+ </owl:unionOf> |
<owl:complementOf> class-expr </owl:complementOf>
```

'instance-expr' representa una instància d'una classe. Es defineix així:

```
<owl:Thing rdf:resource="#instanceName"/>
```

'restriction-expr' representa una restricció sobre una propietat quan es aplicada a la classe. Pot ser qualsevol de les següents:

```
<owl:allValuesFrom> class-expr </owl:allValuesFrom> |
<owl:someValuesFrom> class-expr </owl:someValuesFrom> |
<owl:hasValue> instance-name </owl:hasValue> |
<owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
  non-neg-integer
```

```

</owl:cardinality> |
<owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">
    non-neg-integer
</owl:maxCardinality> |
<owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
    non-neg-integer
</owl:minCardinality>.

```

En OWL Lite les expressions de classe només poden contenir noms de classe i restriccions de propietat. La primitiva ‘owl:hasValue’ no es pot utilitzar en restriccions de propietat. A més, les primitives ‘owl:allValuesFrom’ i ‘owl:someValuesFrom’ només contenen identificadors de classe o tipus de dades nomenades, i les restriccions de cardinalitat només prenen els valors 0 o 1. OWL DL no imposa cap d'aquestes restriccions.

Les funcions no són components del model de coneixement de OWL, encara que les funcions binàries es poden representar amb la primitiva ‘owl:FunctionalProperty’.

Els axiomes formals tampoc són components del model de coneixement de OWL.

Finalment, les instàncies es defineixen utilitzant només vocabulari de RDF. En OWL, hem d'utilitzar l'atribut ‘rdf:datatype’ per a expressar el tipus de valor de les propietats de tipus de dades.

2.2.2 Mecanismes de raonament

La semàntica de model teòric de OWL es descriu de dues formes diferents: com una extensió de la teoria de model RDF(S) i com una semàntica directa de model teòric de OWL. Ambdues tenen les mateixes conseqüències semàntiques a les ontologies de OWL, i estan basades en la semàntica de model teòric de DAML+OIL, tenint en compte les diferències entre ambdós llenguatges.

OWL permet incloure algunes declaracions addicionals (triples de RDF) en les seves ontologies, a part d'aquelles explícitament definides en el llenguatge. No obstant això, oculta les conseqüències semàntiques (o la seva manca) d'aquestes tripletes addicionals. S'han definit un conjunt de processos de prova, incloses proves d'implicació, proves de no implicació, proves de coherència, proves d'incoherència, etc., que il·lustren la correcta utilització de OWL i del significat formal dels seus constructors.

A causa de les seves similituds amb OIL i DAML+OIL, els motors d'inferència utilitzats per aquests llenguatges (FaCT, RACER, TRIPLE, etc.) es poden adaptar fàcilment per a raonar amb ell. Encara no hi ha molts motors d'inferència disponibles per a raonar amb OWL, però es preveu que els haurà aviat. Un motor d'inferència ja disponible és Euler (<http://agfa.com/3/euler/>).

Igual que amb altres llenguatges, aquests motors d'inferència permetran portar a terme classificacions automàtiques de conceptes d'ontologies de OWL, i detectar anomalies en taxonomies de conceptes de OWL.

A més, podem dir que l'herència múltiple està permesa en ontologies de OWL. En la semàntica de OWL, no obstant això, no hi ha explicació de com poden ser resolts els conflictes en l'herència múltiple. Es poden realitzar comprovacions de restriccions en els valors de propietats i les seves cardinalitats.

OWL assumeix raonament monotònic, encara que les definicions de classe i les definicions de propietat estan separades en diferents recursos Web. Això significa que els fets i les implicacions declarats explícitament o obtinguts amb motors d'inferència només poden ser afegits, mai eliminats, i que informació nova no pot negar informació prèvia.

Les eines capaces d'editar ontologies de RDF(S) també es poden utilitzar per a desenvolupar ontologies OWL a condició que el desenvolupador de l'ontologia usi les primitives de Representació de Coneixement de OWL. A més, es poden utilitzar motors de consulta de RDF(S), sistemes d'emmagatzematge i analitzadors per a manejar ontologies de OWL, ja que es poden serialitzar en RDF(S). Finalment, hem d'afegir que ja hi ha disponibles sistemes que transformen ontologies de DAML+OIL a ontologies OWL (<http://www.mindswap.org/2002/owl.html>).

2.3 Requisites

Pel que fa als aspectes visuals, els requeriments de la llibreria són els següent: la llibreria que busquem ha de permetre dibuixar diferents formes geomètriques amb diferents gruixos i utilitzar una gran gamma de colors per poder crear escales de colors.

La llibreria per representar els diferents tipus de relacions també ha de permetre dibuixar fletxes de diferents estils i formes i poder etiquetar-les. També s'ha de poder usar la tipografia que es desitgi amb l'estil que es vulgui (cursiva, negreta, subratllat).

Pel que fa als aspectes de la interacció amb l'usuari, la llibreria ha de proporcionar certes funcions bàsiques com per exemple la existència d'un punter (ratolí) i un controlador d'events per aquest. El punter ha de permetre la possibilitat de seleccionar un o múltiples objectes del graf i sobretot ha de implementar *Drag&Drop*. El *Drag&Drop* és importantíssim i consisteix en poder agafar i arrossegar objectes per la pantalla amb el ratolí. Aquesta operació és molt important perquè l'usuari pugui fer-se la millor distribució del graf un cop comenci l'exploració d'una ontologia.

La llibreria també ha de permetre eliminar i inserir nous objectes al graf de manera senzilla i sobretot molt ràpida. Això també és bàsic perquè s'usarà per expandir i col·lapsar un objecte (classe o instància) amb un simple *doubleclick* amb el punter.

La llibreria també estaria bé que ens permetés fer *zooms*, és a dir, ampliar o reduir el graf però això no és un requisit restrictiu. També estaria bé que tingués un sistema per poder fer i desfer els canvis que es produïssin al graf, encara que això tampoc seria un requisit indispensable per a la tria de la llibreria. Igualment també seria un funcionalitat molt beneficiosa per l'aplicació la possibilitat de guardar i recuperar l'estat del graf en un moment determinat, així com poder fer una captura completa del graf en algun format gràfic ja que normalment els grafs obtingut de les ontologies solen ser bastant extensos pel que fa al nombre de classes, instàncies i relacions no taxonòmiques.

Altres coses interessants que podria aportar la llibreria podria ser la possibilitat de posar *tooltips* al graf.

També s'han de poder crear menús, en concret usarem menús contextuais clickant amb el botó dret del ratolí, i els diàlegs per obrir ontologies i guardar les captures del graf, però això ho farem amb els paquets que ja ens proporciona el JDK.

També hem de mirar que la llibreria sigui totalment pública i de lliure distribució encara que no cal que sigui *open source* ja que si s'ajusta a tots els requisits esmentats no caldrà que la modifiquem però tot i això és un punt a tenir en compte. Seria molt útil que la llibreria tingués disponible una API Java per poder manegar-la més fàcilment.

2.4 Descripció de les alternatives

Les tècniques usades pels visualitzador genèrics de dades (per arbres i grafs) depenen de la naturalesa ontològica de les dades.

Hi ha dos sistemes per visualitzar grans estructures de dades. Un és el *Hyperbolic Tree* [4] (fig. 2.4) per a la navegació d'arbres i l'altre és *The Brain* [5] (fig. 2.3) per a la navegació de grafs.

Les aplicacions que usen aquestes visualitzacions representen estructures sintàctiques com estructures enllaçades.



Fig. 2.3: Web Brain

En el cas específic de les aplicacions que usen la visió hiperbòlica hem de dir que la visualització aconseguida és molt agradable per l'usuari. La idea d'aquest mecanisme de renderització d'estructures, com es pot veure a la figura 2.4, consisteix en què l'element (classe) principal es dibuixa al centre del diagrama i els elements més llunyans són dibuixats exponencialment petits quan major és la distància al centre del diagrama. Donat que el nombre d'elements en un arbre equilibrat augmenta exponencialment amb la profunditat d'aquest, les vistes hiperbòliques són una bona opció, aconseguint mostrar més nivells de l'arbre que una vista convencional.



Fig. 2.4: Hyperbolic tree

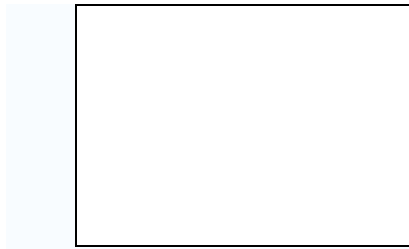
Aquestes són eines per visualitzar i facilitar la navegació de grans estructures, però després també hi ha visualitzador integrats en eines d'edició i desenvolupament d'ontologies com *Protégé* [3] o *OntoEdit* [6] que permeten visualitzar la semàntica sencera d'una part limitada d'un esquema del procés de desenvolupament de l'ontologia.

L'altre estil de visualització consisteix en usar llibreries basades en *clusters*.

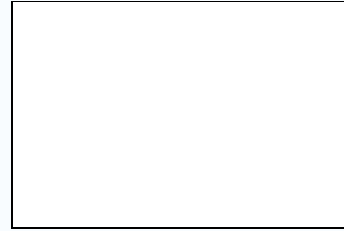
Fins ara totes les aplicacions esmentades tenien el gran defecte de l'escalabilitat que és un requeriment bastant important per ontologies amb un gran nombre de classes i sobretot instàncies ja que es preveuen més d'aquestes últimes que de classes com ja s'ha explicat. La solució a aquest problema es pot resoldre si s'utilitza la tècnica de visualització *Cluster Map*. Aquesta tècnica es basa en la utilització de clústers per agrupar les instàncies d'una classe, entenent com a clúster una geometria de farcell o grup. Serveix per visualitzar conjunts d'objectes classificats. Els propòsit és mostrar si i com aquests conjunts classificats es superposen, semblant als *Diagrames de Venn* (fig. 2.5) i als *Diagrames d' Euler* (fig. 2.6). Els *Diagrames de Venn* provenen de la branca de les matemàtiques coneguda com teoria de conjunts. Aquests diagrames són usats per a mostrar gràficament la relació entre diferents conjunts. Un *Diagrama de Venn* és un gràfic consistent en dos o més cercles, en el qual cadascun d'ells representa un conjunt. Un *Diagrama de Venn* mostra totes les relacions lògiques entre els conjunts d'interès.

Per exemple, quan els cercles se superposen, indiquen l'existència de subconjunts amb algunes característiques comunes. Per exemple, quan els cercles se superposen, indiquen l'existència de subconjunts amb algunes característiques comunes.

Els *Diagrames de Euler* són similars als *Diagrames de Venn*, però aquests no necessiten totes les possibles relacions.



2.5: Diagrama de Venn. Intersecció de dos conjunts.



2.6: Diagrama d'Euler.

Es veurà amb detall el seu funcionament en el punt 2.4.4 on veurem una eina implementada amb la llibreria Cluster Map [7] (fig. 2.7).

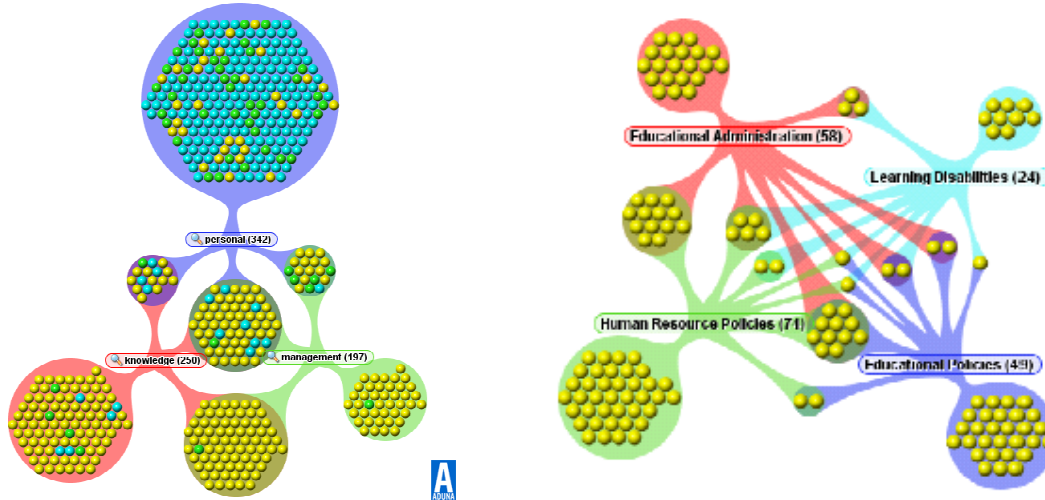


Fig. 2.7: Fotos de clusters

2.4.1 GraphViz (integrat)

Un eina de visualització gràfica d'ontologies molt satisfactòria per a la visualització de models de dades RDF és GraphViz [8], desenvolupat per AT&T. Les eines basades en GraphViz, com *IsaViz* [9] o *RDFViz* [10], són bàsicament empleades per entendre dades RDF.

Aquestes visualitzacions permeten la representació de qualsevol tipus de relacions entre classes i instàncies, sent així, un bona *suit* per mostrar ontologies complexes. La escalabilitat de la visualització és una mica pobre perquè és difícil d'entendre la visualització aconseguida tan punt es comencen a afegir moltes relacions.

Els programes que disposen de GraphViz [8] agafen descripcions de grafs en un llenguatge de text i creen diagrames en els formats més utilitzats com imatges i SVG (Scalable Vector Graphics) per pàgines webs, Postscript per incloure en PDF or altres documents; o mostrar en un explorador de grafs interactiu. GraphViz també suporta GXL i XML.

GraphViz té algunes característiques útils per diagrames concrets, tal com opcions de colors, fonts de lletres, estils de línies, hiperlinks i formes geomètriques.

Protégé [3] té dos plug-ins bàsics de visualització. En primer lloc, tenim el l'etiqueta *OntoViz* [11] que usa GraphViz [8] per visualitzar ontologies com es pot veure a la figura 2.8.

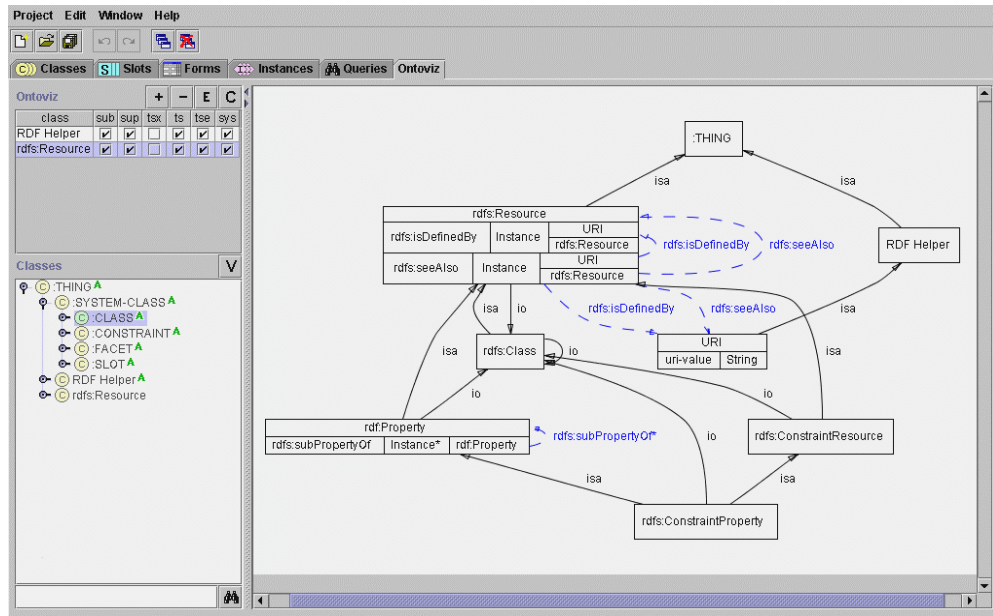


Fig. 2.8: Protégé OntoViz tab

La visualització és àmpliament configurable i inclou:

- Seleccionar un conjunt de classes o instàncies per visualitzar part de la ontologia.
- Mostrar propietats de les classes i instàncies i de les relacions.
- Especificació de color per als nodes(classes i instàncies) i per a les relacions.
- Quan es seleccionen només algunes classes o instàncies, es poden aplicar varies operacions de tancament (subclasses, superclasses, etc) per mostrar la seva proximitat.

L'etiqueta OntoViz [11] i els seu codi font (igual que Protégé) esta disponible gratuïtament sota la llicència pública de *Mozilla* (<http://www.mozilla.org/MPL/MPL-1.1.html>) de codi obert.

I en segon lloc tenim l'etiqueta *Jambalaya* [12] que permet a l'usuari buscar, explorar i interactuar amb la estructura. Aquesta visualització permet una vista més basta de la jerarquia de classes de la ontologia, usant dues metàfores de visualització diferents. Per una banda, és pot dibuixar el clàssic arbre d'exploració per la jerarquia de classes i

instàncies i per altra banda, la vista permet fer zooms graduals des de la vista més general fins a conceptes més específic.

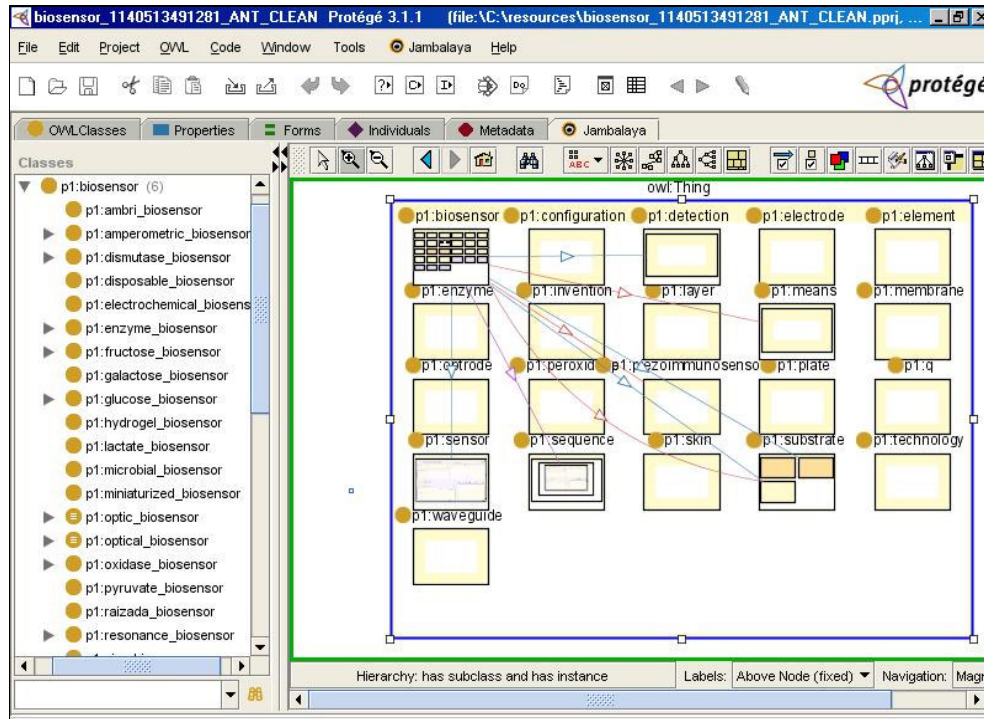


Fig. 2.9: Protégé Jambalaya tab

Aquesta llibreria no la triarem perquè té molt poca escalabilitat i és de molt baix nivell. Per altra banda, encara que es open source no hi ha cap API disponible que ens ajudi a entendre el funcionament d'aquesta.

2.4.2 OntoRama (tipus hyperbolic)

OntoRama [13] visualitza els arbres usant la visió hiperbòlica.

OntoRama, més aviat utilitza una projecció esfèrica com a representació geomètrica que la projecció hiperbòlica descrita per Lamping y Rao. Aquesta projecció esfèrica, que es basa en altres teories diferents, comparteix moltes característiques de la geometria hiperbòlica.

OntoRama [13] ha estat àmpliament usat per visualitzar ontologies expressades en RDF

però no es limita al RDF. En el moment de visualitzar ontologies expressades en RDF, OntoRama [13] fa una traducció de RDF a arbres amb vèrtexs etiquetat i fletxes etiquetades. Aquest arbre són visualitzats simultàniament en un estil d'exploració i en un diagrama d'estil hiperbòlic.

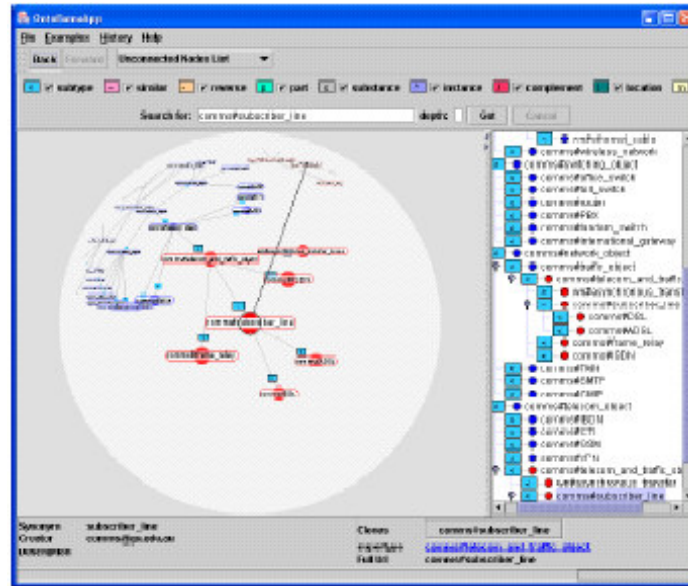


Fig. 2.10: OntoRama.

Aquestes dues vistes que es mostren a la fotografia (fig. 2.10), mostren el mateix fragment d'ontologia i estan enllaçades de manera que seleccionant un objecte en una de les vistes provoca que l'objecte es seleccioni en l'altre vista també. Totes dues vistes es complementen perquè quan un té propietats indesitjables l'altre no les té. Per exemple, si tenim el cas que un node té molt fills, en la vista hiperbòlica les etiquetes es superposen i la visió no es gaire entenedora. En aquest cas l'usuari pot reordenar a la vista d'arbre on les etiquetes no es superposen unes a les altres.

OntoRama [13] suporta el cas de la herència múltiple mitjançant la creació de clons diferenciant-los amb color de l'original i lligant-los a aquest mitjançant fletxes.

Selecció d'un vèrtex també es mostra informació d'aquest en la part inferior de la pantalla sent aquesta informació totalment configurable per l'usuari.

OntoRama [13] també disposa d'un sistema de cerca per a totes dues vistes mitjançant un camp on l'usuari pot escriure l'objectiu.

Pel que fa a la interoperabilitat a través de punter, quan l'usuari mou el ratolí passant per sobre d'un vèrtex els seus clons són identificats per línies dibuixades entre el vèrtex seleccionat i els seus clons. I quan arrosseguem un vèrtex tot el diagrama es mou d'acord la trajectòria del ratolí.

Per concloure podríem dir que OntoRama [13] va sorgir com a eina pràctica per visualitzar ontologies descrites en RDF usant el paquet Java Swing. El projecte també té un experiment implementat amb *Macromedia Flash*, anomenat ProntoRama, basat en una composició radial.

La llibreria OntoRama [13] no serà la triada per implementar el nostre visualitzador d'ontologies principalment perquè no es disposa cap API Java pública ni de cap codi disponible per al desenvolupament d'aplicacions a part de que no compleix la totalitat de tots els nostre requisits.

2.4.3 TouchGraph (tipus hiperbòlic)

TouchGraph [14] té una visualització molt simple, usant etiquetades com a nodes per les diferents classes. Les arestes que enllacen aquestes etiquetes són amb forma de triangle isòsceles i té la possibilitat de pintar-se en diferents colors.

El fet que els elements es representin en etiquetes rectangular de les puntes arrodonides no permet diferència a simple vista el tipus de classe o instància que pugui ser.

El que realment és interessant d'aquesta llibreria és la interoperabilitat, ja que ofereix un gran rendiment pel que fa a la distribució del graf en pantalla. No es produeixen superposicions d'etiquetes perquè sempre es manté un separació entre tots el nodes. Però les arestes si que se solapen encara que això és inevitable ja que d'arestes sempre n'hi ha moltes i aquestes es creuen per força.

Té aspectes gràfics interessants, com el fet que totes les arestes, de la classes seleccionada passant el punter per damunt, es ressaltin en un altre color poden veure així quines son les subclasses d'aquesta.

Representa relacions tant d'entrada com de sortida cap a altres classes, diferenciant-les únicament amb el color (unes vermelles i les altres blaves) i posant el costat curt de

l'aresta en forma de triàngle a la classe origen.

TouchGraph [14] permet expandir i col·lapsar un node. Això vol dir que per tots els fills de cada node aquests els podem amagar i mostrar en qualsevol moment així com també eliminar-lo del graf juntament amb tots els seus corresponents fills que no siguin compartits amb altres classes ja que TouchGraph [14] suporta herència múltiple.

La interacció amb l'usuari es fa a través d'un punter (ratolí). Amb el punter pots desplegar un petit menú amb les accions d'eliminar, col·lapsar i expandir una classe i també suporta Drag&Drop. El drag&drop implementat en aquesta llibreria és realment potent, ja que quan arrosseguem una node, provoca que la resta de nodes del graf es recol·loquin automàticament sense tenir la possibilitat que quedin superposats uns amb els altres. Però té l'inconvenient que els nodes no es poden traslladar a una altra posició del graf, sempre torna automàticament al punt original, això fa que el drag&drop d'aquesta llibreria encara que resulta agradable a la vista és una mica inútil. A la següent figura (fig. 2.11), es pot veure un exemple d'un graf visualitzar amb aquesta llibreria.

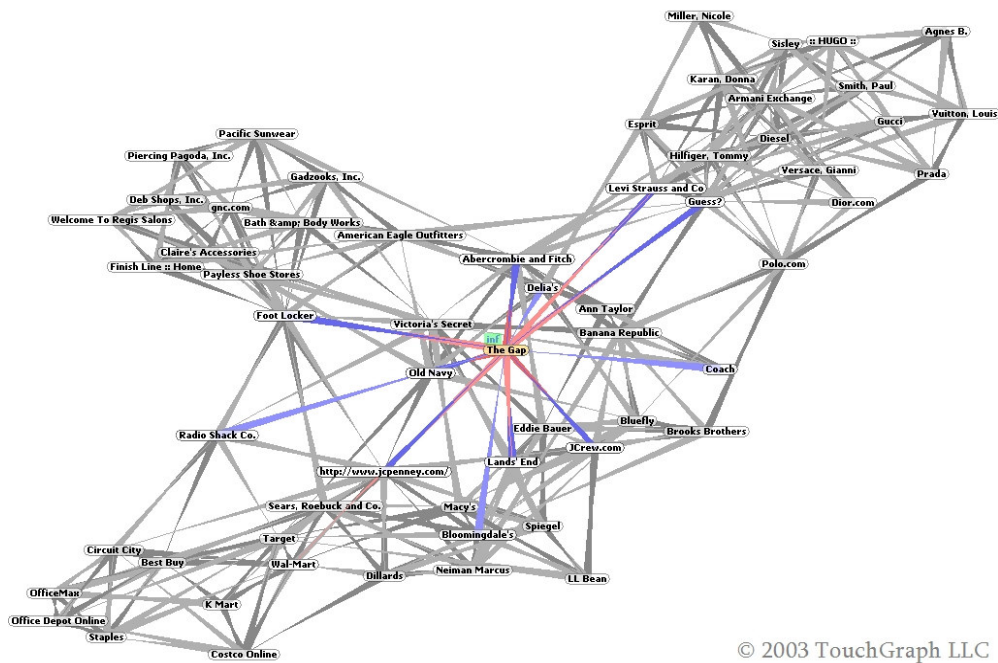


Fig. 2.11: Graf amb TouchGraph

Aquesta llibreria no està disponible per utilitzar-la en la creació d'aplicacions, així que queda descartada directament.

També té moltes mancances i moltes limitacions que fa que no compleixi els nostres requisits.

2.4.4 Aduna ClusterMap Library 2005.1 (tipus clustering)

La llibreria Cluster Map [7] conté funcionalitats per crear visualitzacions de col·leccions d'objectes classificats jeràrquicament, referides a vegades com taxonomies o jerarquies del concepte.

El cor d'aquesta llibreria està format per un conjunt de classe usades per la creació de la visualització. A més, aquestes classes proporcionen la construcció de un interfície d'usuari Java Swing-based, la qual fa possible la creació d'una visualització específica interactiva.

La llibreria també conté un visor del mapa de clusters, un eina d'escriptori simple feta d'aquests components que demostra la funcionalitat primària de la visualització i de la seva GUI (interfície gràfica d'usuari) i això es pot utilitzar per a una avaluació ràpida de la idoneïtat de la visualització del mapa de clústers per a un propòsit i tasca específics a través d'un format directe, fitxer de dades basat en XML.

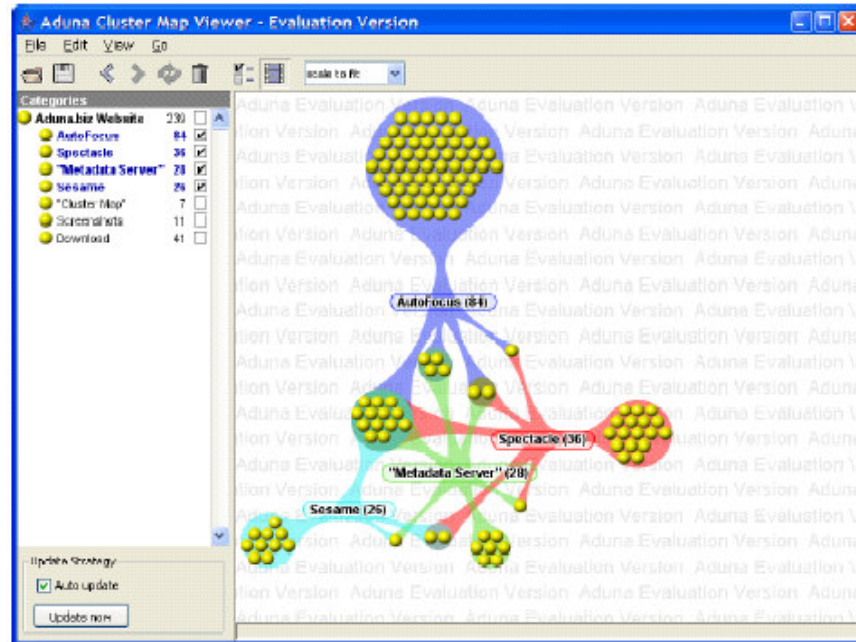


Fig. 2.12: Foto de Cluster Map Viewer

Descripció de la fotografia anterior (fig. 2.12).

Les ombres de colors representen conceptes, o sigui, les classes de l'ontologia i cada una té la seva etiqueta amb el seu nom i cardinalitat. Quan una relació de subclasse és la mateixa entre dues classes, aquestes subclasses son connectades por una aresta dirigida. Les esferes més petites de color groc representen instàncies del concepte, entenent per instància (en aquest cas) qualsevol document o pàgina web, és a dir, recursos.

Aduna ens proporciona una API Java de la llibreria. La imatge de sota (fig. 2.13) és un diagrama que mostra les classes i les seves relacions més importants d'aquesta API pública.

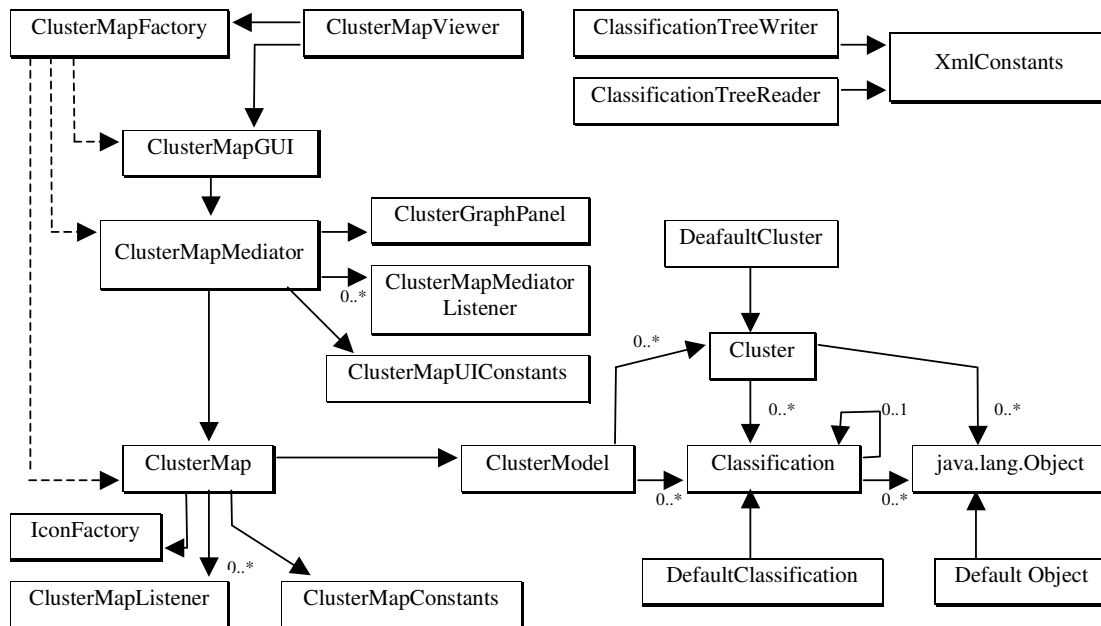


Fig. 2.13: Esquema de la llibreria

La classe **ClusterMap** és la classe central d'aquest paquet, produint la visualització del mapa de farcells. Un ClusterMap funciona sobre estructura de dades de **ClusterModel**, contenint un conjunt de farcells, anomenats **Clusters**. Cada Cluster representa una única combinació de **Classifications** i conté tots els **Objectes** (que representen per exemple documents o pàgines webs) que se contindran totes les classes que es representaran amb el Cluster però no de qualsevol de les altres classes presents al ClusterModel.

Per exemple, considerem que s'ha creat un ClusterModel que representa de las classes A, B i la C. Suposem que aquest model conté un Cluster que representa la combinació de las classes A i B. Aquest Cluster llavors contindrà tots els objectes que són membres de las classes A i B però no de la classe C.

Classifications i ClusterModel són instàncies i java.lang.Object s'usa per representar els Objects en els Classifications, és a dir, que s'usa per representar els objectes dins d'una classificació. Això proporciona gran flexibilitat per l'integrador, permeten així escollir

quines classes s'usaran per proporcionar aquesta funcionalitat. La Cluster Map proporciona implementacions per defecte per a tots estos casos que seran suficients per simples aplicacions. Molt notablement, **DefaultClusterModel** conté un algoritme eficient per calcular el conjunt de Clusters basat en un conjunt de Classifications poblats, de manera que els integradors de l'aplicació no han d'inventar la roda.

El ClusterMap mostra el contingut d'un ClusterModel com un graf. Pot visualitzar el graf usant els components GUI comentats a continuació o guardar-lo en un fitxer.

Un ClusterMap utilitza un **IconFactory** per crear els icones que s'utilitzen per mostrar Objectes i Classifications. L'usuari pot fixar la seva pròpia implementació d'aquesta interfície al ClusterMap, de manera que tens el control total de l'aparença d'aquests objectes.

També es pot afegir un **ClusterMapListener** a un ClusterMap per obtenir notifikacions desls canvis l'estat del ClusterMap, com per exemple quan el ClusterModel o la visualització del graf canvien o quan les configuracions són modificades.

Un **ClassificationTreeWriter** pot serialitzar un arbre de Classifications en un format basat en XML. El documento resultat es pot llegir amb un **ClassificationTreeReader**, que reconstrueix l'arbre de Classification. La deserialització de l'àrbre de Classification consistirà només de **DefaultClassifications** i de **DefaultObjects**, que són les implementacions per defecte de les classes Classification i Object.

El **ClusterMapMediator** és essencial per a la construcció d'un GUI al voltant d'un ClusterMap. A ClusterMapMediator envolta un ClusterMap i produeix diversos components GUI tal com un **ClusterGraphPanel** que visualitza el graf, un derivat de JTree tenint en compte la selecció de la classe, diverses accions, etc. Un ClusterMapMediator assegura la consistència entre tots els components que crea. Per exemple, quan una classes se selecciona per a la visualització a l'àrbre de classificació, aquesta s'afegeix automàticament al pannel que mostra el graf del mapa de Clusters. ClusterMapMediator crea dinàmicament el ClusterModels visualitzat al ClusterMap.

El ClusterMapMediator no és un component GUI; simplement crea i manega components amb els quals es pot construir un sofisticat GUI. Per a construir ràpidament un GUI, l'usuari pot utilitzar un **ClusterMapGUI**. ClusterMapGUI és una subclasse de JPanel que inclou un ClusterMapMediator, que permet crear un nombre de components UI. El Cluster Map Viewer (visor de ClusterMap) esmentat anteriorment ha estat creat

usant `ClusterMapGUI`.

Les instàncies de la classe `ClusterMap` i d'algunes més de les classes usades son generades per un **`ClusterMapFactory`**.

Com a desenvolupador es desitja un bon aspecte i comportament de totes les parts de la visualització. El `ClusterMap` i el `ClusterMapMediator` tenen un sistema `set/get` basats en tripletes simples del clau-tema-valor que es poden utilitzar per adaptar el comportament. A més, a l'exporta la visualització actual del `ClusterMap` a un mapa d'imatges HTML o quan obtens accions o `JComponents` d'un `ClusterMapMediator`, també s'utilitzen les claus o els pares de clau-valor.

Els paràmetres que se poden fixar en un `ClusterMap` bàsicament influeixen en la representació del graf, amb només algunes excepcions.

Només algunes de les característiques del `ClusterMapMediator` són actualment adaptables, influenciant el seu comportament interactiu. Els ajustament s'apliquen igual que al `ClusterMap`.

Per permetre la construcció dels components de UI (interfície d'usuari) com *toolbars*(barra d'eines), menús, etc., el `ClusterMapMediator` facilita la recuperació de la acció amb l'uso del mètode *getAction* amb les claus apropiades.

Aquesta llibreria encara que és molt agradable a la vista gràficament no ens proporciona les suficients eines per poder diferenciar tots els tipus de classes i instàncies que tindrem al nostre visualitzador. Però la raó principal per no escollir aquesta llibreria és perquè, tot i té una API Java disponible no és una llibreria lliure. Aduna (empresa propietària de la llibreria) distribueix una versió d'avaluació d'aquesta i per això tenim tant publicitat implícita de fons del pannel on es visualitza el graf, com explícita col·locant el logotip a la cantonada inferior dreta de la pantalla.

A més d'aquests inconvenient de llicència, la llibreria és molt còmoda de treballar perquè es programa a alt nivell però no es el suficient flexible per complir els requisits imposats.

2.4.5 JGraph (simple)

JGraph [15] proporciona un component completament estàndard del graf per la llibreria Java Swing UI que suporta extenses visualitzacions i opcions d'edició.

La intenció del projecte de JGraph [15] es proporcionar una implementació Swing d'un component de grafs disponible lliurement. Com component Swing per grafs, JGraph es basa en la teoria matemàtica de la xarxa, anomenada teoria de grafs, i en la llibreria d'interfície d'usuari Swing que defineix la arquitectura.

Les principals característiques de JGraph [15] són:

- Completament compatible amb Swing.
- Disseny net i eficient.
- 100% Java.
- Pot ser usat tant el la part client com en el costat servidor.
- Opera sobre diversos backends, fitxers XML a bases de dades o altres sistemes, tal com LDAP, JMS, J2EE.
- Proporciona operacions del tipus *zooming*, *folding*, fer, desfer i *Drag and Drop* entre altres.

La idea entre aquestes característiques està basada en l'experiència d'altres llibreries de grafs, en les que sovint són *frameworks* llargs i complexos que no segueixen els estàndards.

Swing és la llibreria d'interfície d'usuari de Java i proporciona els elements que es poden posar en una aplicació de finestres. Aquests elements s'anomenen components d'interfície d'usuari o simplement components. Alguns d'aquests components més comuns són botons, llistes, arbres i taules.

Swing, a la vegada, està basats en AWT (Abstract Windowing Toolkit). AWT és una altra llibreria d'interfícies d'usuari de Java que es pot usar independentment de Swing, però aquesta última és més sofisticada i incorpora moltes més propietats.

El patró de disseny MVC (Model Vista Control) de Swing s'utilitza per separar el model de la vista i estableix la relació: més d'una vista poden ser afegides al mateix model i totes aquestes vistes s'actualitzen automàticament quan el model canvia.

Seguint el patró MVC, JGraph [15] separa la part del model, la de la vista i la part de

control.

La següent figura (fig. 2.14) mostra un exemple d'un graf creat amb JGraph.

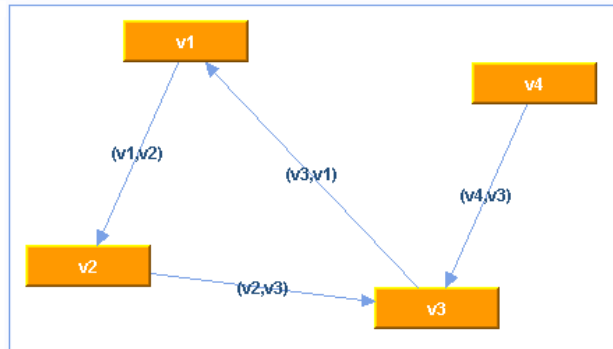


Fig. 2.14 Visualització d'un graf

La part del model descriu la interfícies bàsica del model del graf, la selecció del model i els elements que aquest conté; així com les classes usades per canviar el model del graf. El model, com ja s'ha dit, proporciona les dades pel graf, consistent en cel·les que poden ser vèrtexs, fletxes o ports, i la informació de la connectibitat la qual es defineix usant ports que es componen de orígens o destins de fletxes. Aquesta informació de connectibitat es refereix a la estructura del graf.

El model també proporciona accés a una altra estructura, anomenada estructura de grup. El canvis que es poden produir al model són atòmics i es poden emmagatzemar en un historial per a que més endavant es puguin desfer i tornar a refer en un moment determinat. Els possibles canvis que suporta el model són la inserció, eliminació i modificació.

La part de la vista estudia la representació interna de la visualització del graf i el pas i l'actualització entre el model i la vista.

A JGraph [15] la vista és una mica diferent d'altres classes en Swing perquè la visió de JGraph [15] té estat, és a dir, que conté informació que solament s'emmagatzema a la vista.

L'element de control explica el procés de representació, els passos a seguir en l'edició al moment i el manegament de cel·les, i els objectes involucrats en la transferència i en la selecció.

El control defineix el pas dels events que provoca l'usuari, als mètodes del graf i de la selecció del model i de la vista. Està implementat d'una manera depenent de la plataforma al *UI-delegate* (delegat de la interfície d'usuari), i bàsicament s'ocupa de la edició al moment, del manegament de les cel·les, i d'actualitzar la visualització. Com que l'únic objecte que s'intercanvia quan el *look and feel* canvi, el *UI-delegate* també especifica la part específic de *look and feel* de la representació.

A part dels escoltadors d'events per seguir l'entrada de l'usuari, el UI-delegate també s'utilitza per pintar el graf, i actualitzar la visualització quan el graf o la selecció canvia. Quan el graf es pintat, el delegat d'interfície d'usuari recorre totes les vistes associades i les pinta amb els seus respectius renderitzadors.

Una vista, generalment, mai contindrà les dades, i un model mai contindrà codi de dibuix, amb la excepció que a part de les dades del que el model proporciona, la vista pot contenir dades addicionals, que són independents del model.

En conclusió, el component JGraph [15] suporta grans visualitzacions i opcions d'edició. Jgprah [15] és molt configurable i en cas que les opcions de configuració que proporciona la llibreria no siguin suficients, el codi del component està disponible per modificar la implementació per defecte d'aquest.

JGraph [15] és una llibreria de baix nivell però això fa que sigui més flexible a l'hora que crear l'aplicació. A més com s'ha dit al principi és una llibreria totalment Java i és *open source*, de manera que si volem podem modificar-la. També tenim un API disponible que ens ajudarà molt a manegar aquesta llibreria gràfica tant útil.

Per tant, per tot això i perquè s'ajusta bastant als nostres requisits per poder desenvolupar aquest projecte, triarem JGraph [15] com la llibreria gràfica.

2.5 Manual JGraph

En aquest apartat explicarem la funcionalitat de la llibreria.

Amb la classe JGraph, es pot visualitzar objectes i relacions.

Un objecte JGraph no conté cap dada, simplement proporciona la vista d'una dada. Com

qualsevol component no trivial de Swing, JGraph [15] obté les dades a través de la consulta al seu model de dades.

JGraph [15] visualitza les dades dibuixant element individuals. Cada element dibuixat del graf correspon exactament a un element de les dades que s'anomena *cell*. Cada una d'aquestes cel·les pot ser un vèrtex, una aresta o un port. Els vèrtexs poden tenir veïns, ja siguin fills o pares, i cada aresta té un o cap vèrtex origen i destí.

Per crear un graf, s'ha d'instanciar la classe JGraph i col·locar-la dins d'un component Swing contenidor com per exemple un *JPanel*, *JScrollPane* o *JFrame*.

La classe JGraph té un constructor per defecte sense paràmetres però se li podria passar com a paràmetre un model de dades, la composició dels element sobre el graf o un manegador de selecció.

El constructor per defecte que utilitzarà una instància de *DefaultGraphModel*, crearà una instància de *GraphView* per aquest i afegirà un graf d'exemple al model (fig. 2.15).

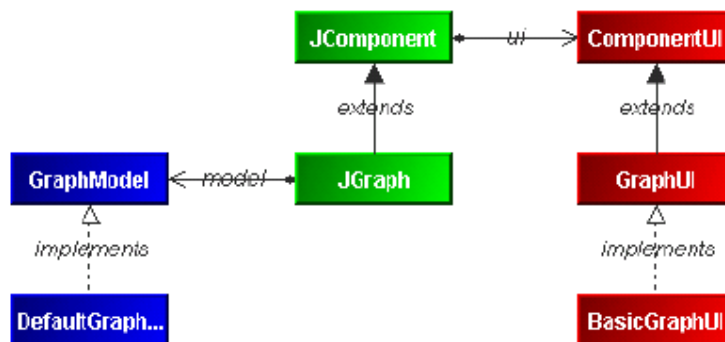


Fig. 2.15: Graf d'exemple de JGraph

La implementació per defecte de JGraph defineix el següent conjunt d'operacions amb el teclat:

- Alt-Click selecciona una cel·la.
- Shift- or Ctrl-Select afegeix a la selecció.
- Shift-Drag obliga la compensació en una direcció.
- Ctrl-Drag clona la selecció.
- Dobleclick or F2 edita la cel·la.

JGraph [15] ofereix les interaccions següents: edició al moment, moure, clonar, redimensionar, doblregar (afegir, esborrar i moure punts de l'aresta), establir i eliminar connexions. Totes aquestes interaccions poden ser deshabilitades usant el mètode *setEnabled(false)* a la vegada, però també es disposa d'un mètode per activar o desactivar cada una d'aquestes interaccions, com per exemple, *setMoveable* o *setSizeable* per activar i desactivar la possibilitat de moure i redimensionar la grandària de les cel·les respectivament.

A més, JGraph [15] també ens proporciona molt més mètodes com per exemple *setDragEnabled* per activar i desactivar l'ús del *drag&drop*.

Per veure tota la resta dels mètodes consultar la API Java disponible de la llibreria on es mostra els mètodes amb els paràmetres d'entrada (si en necessita) i de retorn, i una petita explicació de la funcionalitat d'aquest.

També podem capturar i respondre als events que ens provoqui el ratolí i del propi JGraph [15], ja sigui per canvis en el model, en la vista o en la selecció, o per l'editor *undoable*.

JGraph és compatible al *Undo-Support* de SWING. Cada vegada que l'usuari realitza una acció, el model envia un corrector que descriu el canvi. El corrector proporciona un mètode al *undo* per a desfer el canvi. Si *undo* és cridat al corrector, el model activa un *GraphModelEvent*, però no un *UndoableEditEvent*. Els últims solament es activen si JGraph desitja indicar que un corrector s'ha agregat a l'historial, que no és el cas per a un *undo*.

Per detectar els clicks que l'usuari pugui fer sobre un cel·la del graf hem d'usar un *MouseListener* junt amb el mètode *getFistCellForLocation*.

Si estem interessats en un sistema de notificacions hem d'implementar la interfície *GraphModelListener* i afegir una instància d'aquest al model amb *addGraphModelListerner*. Amb això aconseguirem que se'ns notifiqui quan una cel·la s'ha inserir o eliminat o quan l'etiqueta, l'origen i destí, pare o fill d'un objecte ha canviat.

El canvis realitzats sobre la vista també són notificats per la classe *GraphView*. Aquests canvis són de tipus canvi de dimensió, de color, de posició, etc. de la vista d'una cel·la en particular.

També es pot crear una classe que implementi la interfície *GraphSelectionListener* per saber els canvis que succeeixen en la selecció del graf.

JGraph [15] proporciona dos oïdors d'alt nivell que s'utilitzen per a controlar la funcionalitat del ratolí i del *datatransfer*. Eliminant aquests, pots guanyar el control complet de la selecció i de la capa de *datatransfer*, és a dir, com les cel·les s'importen i s'exporten via *drag&drop* i el portafolis.

També es pot personalitzar el manera de dibuixar els elements del graf de manera limitada. JGraph [15] defecte dibuixa els vèrtexs en forma de rectangles. Però a nosaltres ens interessa poder aplicar diferents formes geomètriques perquè tenim diferents tipus d'elements que són vèrtexs. Per exemple, nosaltres mostrarem classes i instàncies d'aquestes classes, però per JGraph [15] això són vèrtexs i per tant els representarà tots dos element (per nosaltres diferents) de la mateixa manera, amb un rectangle i amb el mateix tipus de lletra, colors, etc. Per solucionar això hem d'implementar, una subclasse d'un dels *renderers* que tenim per defecte, i sobreescrivre el seu mètode *paint*. Un *renderer* és una extensió d'un component que pinta cel·les basant-se en les seves propietats. Així, ni JGraph ni la seva específica implementació *look&feel* (mirar-i-sentir) conté realment el codi que pinta una cel·la. En lloc d'això, JGraph utilitza el codi de dibuix del *rendereritzador* de la cel·la. Un *renderer* nou es pot associar a una cel·la sobreescrivint el mètode *getRendererComponent* del *CellView* corresponent, o el mètode *getRenderer* per a les extensions de la classe *AbstractCellView*.

En el punt 3.4 s'explica més detalladament i en el 3.5 es veurà una aplicació d'això ja que l'hem usat per dibuixar de diferent manera les classes i les instàncies.

Si es desitja poder crear *tooltips* al graf s'ha d'implementar el mètode *getToolTipText* heretat de la classes *JComponent* i registrar al graf un *TooltipManager* de Swing per visualitzar aquestes etiquetes correctament.

En els grafs que tenen estructures complexes, és bastant comú oferir una finestra de diàleg en comptes d'una simple edició al moment, però en el nostre cas no usarem cap *dialog*.

Com ja sabem JGraph [15] separa el model i la vista. El model es defineix per la

interfície *GraphModel*, i conté els objectes que implementen la interfície *GraphCell*, mentre que la vista es representada per la classe *GraphView*, i conté els objectes que implementen la interfície *CellView*. El pas de les cel·les a les vistes es defineix amb la interfície *CellMap*. Un model pot tenir diverses vistes i per cada cel·la del model, existeix exactament una *CellView* per cada *GraphCell*. L'estat d'aquests objectes es representa per un mapa de parelles clau-valor.

Com hem dit l'estat d'una cel·la i igualment per una vista es representada pels seus atributs. En tots dos casos, s'utilitza la classe *GraphConstants* per canviar l'estat i es fa en dos passos: primer es construeix l'objecte que constitueix el canvi a realitzar mitjançant la creació de nous mapes usant el mètode *createMap* de la classe *GraphConstants*, i després executa el canvi al model o a la vista del graf.

L'estructura del graf es defineix amb els mètodes *getSource* i *getTarget*, els quals en retornen en tot moment el port origen i destí d'un objecte. Aquests ports a la vegada són fills d'un vèrtex i així podrem crear relacions entre vèrtexs; en el nostre cas entre classes i entre classes amb instàncies permetent també la multiplicitat de punts de connexió.

Per inserir vèrtexs del model, primer s'han de crear estenent de la classe *DefaultGraphCell*, a continuació afegirem dos ports a la cel·la i crearem un mapa d'atributs els quals, a la vegada, també seran mapes de parelles clau-valor, i que seran accessibles de manera segura a través de la classe *GraphConstants*. I finalment es cridarà al mètode *insert* que passant-li els paràmetres adients modificarà el model.

Per inserir arestes al graf el procediment és el mateix però usant la classe *DefaultEdge* i donant-li atributs adients que seran diferents del cas anterior de la inserció de la cel·la.

Per eliminar, tant una cel·la com una aresta del graf, simplement s'ha d'invocar el mètode *remove* del model i passant-li per paràmetre únicament les cel·les que es volen eliminar.

La classe utilitzada per crear un model de dades per al graf és *DefaultGraphModel*.

Si el *DefaultGraphModel* no compleix totes les teves necessitats, llavors haurem d'implementar un model personalitzat però que implementi la interfícies *GraphModel* igual que la classe *DefaultGraphModel* perquè adquireixi la qualitat de model. *GraphModel* admet qualsevol tipus d'objecte com a cel·les i no requereix que aquestes cel·les implementin la interfície *GraphCell* encara que en el nostre cas totes hereten de *DefaultGraphCell*.

3. ESPECIFICACIÓ DEL PROJECTE

3.1 Introducció

L'eina que ens proporcionarà els fitxers OWL, és un constructor semiautomàtic d'ontologies a partir de la Web.

Les ontologies creades, a banda de les classes i les relacions, també tenen informació relativa a la cerca, com per exemple, dades estadístiques i informació més rellevant com són les pàgines webs trobades per a cada concepte.

Aquesta meta-informació no està suportada per l'OWL estàndard però el nostre visualitzador volem que la mostri com a informació extra.

Tindrem meta-informació de dos tipus: meta-informació interessant per al desenvolupador i meta-informació per a l'usuari. La meta-informació per al desenvolupador són dades estadístiques sobre el procés d'aprenentatge, en canvi, la meta-informació per a l'usuari, són les planes webs categoritzades que cobreixen el tema.

Aquest sistema crea una única gran ontologia a partir de l'exploració recursiva de conceptes relacionats amb un determinat domini inicial i de la unió de varies ontologies de diferent dominis però que estan relacionades entre elles.

En la següent figura (fig. 3.1) es pot observar esquemàticament com es crea l'ontologia.

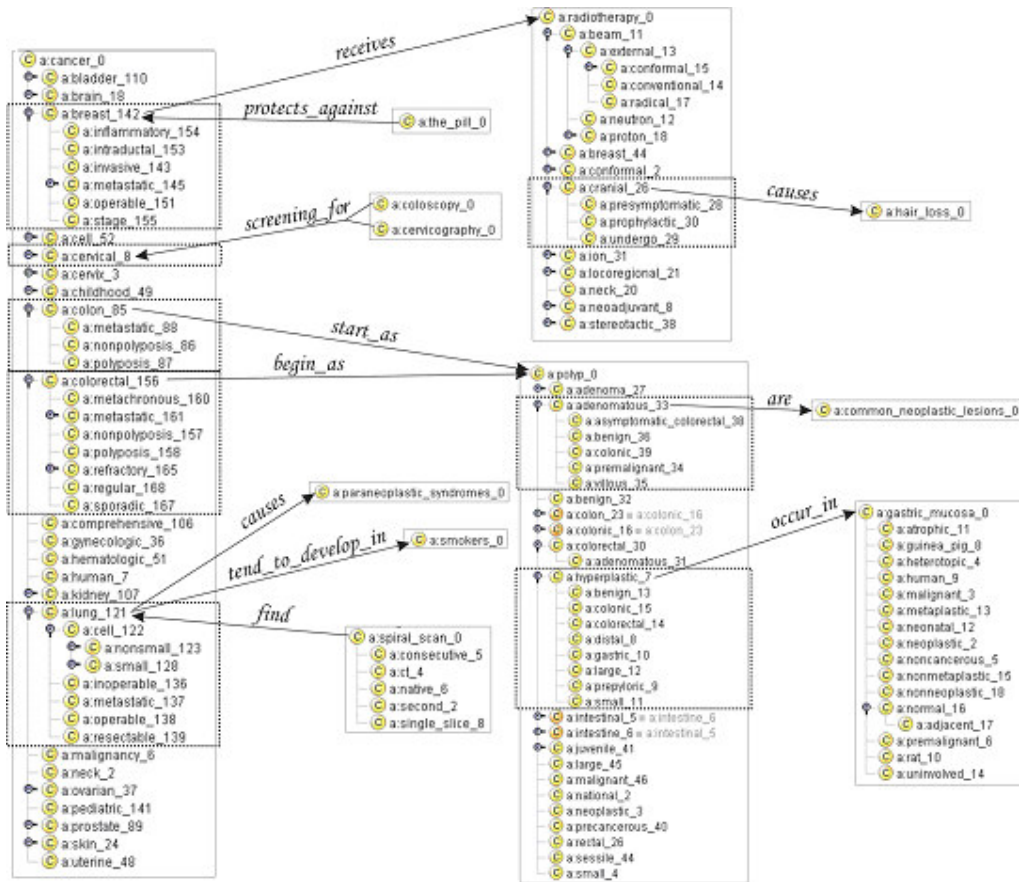


Fig. 3.1: Exemple de taxonomia generada pel constructor de taxonomies.

El sistema retorna dos fitxers amb la taxonomia desitjada, expressada com una Ontologia en llenguatge OWL. Un dels fitxers conté la taxonomia completa (fitxer full) juntament amb la meta-informació associada a cada elements de l'ontologia, és a dir, elements que aporten informació de la classe. En el següent exemple (fig. 3.2) es pot observar una classificació de les pàgines Web del domini “lung cancer”, en ell apareix la relació d'instàncies, la categorització de les URLs de “lung cancer” i les seves subclasses amb les URLs classificades.

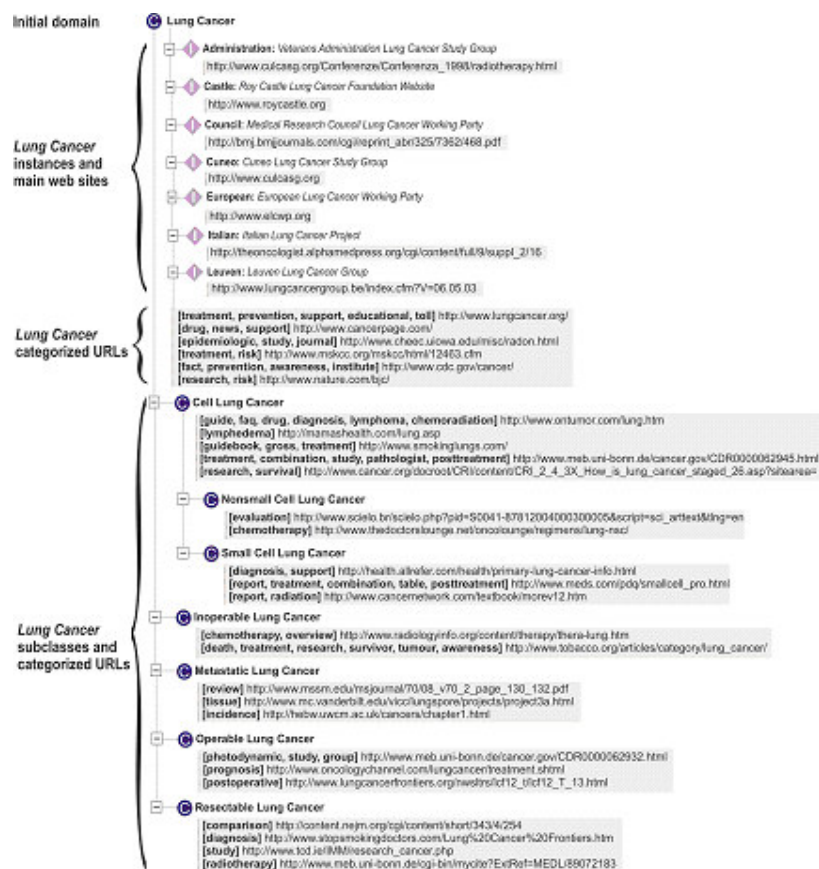


Fig. 3.2: Exemple de classes, instàncies i URLs trobades per al domini "lung cancer"

L'altre fitxer també conté la taxonomia però sense atributs (fitxer clean). És a dir, d'aquest fitxer solament podem obtenir estructura jeràrquica i en el cas de les instàncies no apareix el seu nom. Aquesta separació és degut al fet que com que OWL [1] no està pensat per a emmagatzemar dades, no podríem visualitzar el fitxer complet en programes com el Protégé [3]. D'aquesta forma donem l'opció a l'usuari a visualitzar la seva ontologia en aquests programes mitjançant el fitxer clean. En el cas d'aquest projecte es mostren ambdós fitxers segons decideixi l'usuari.

Per tant, visualitzarem dos tipus de fitxers OWL, els corresponents a ontologies completes amb meta-informació que s'utilitzarà per enriquir la visualització, sempre que aquesta estigui disponible, i les ontologies netes.

3.2 Especificació dels fitxers OWL

Les ontologies que visualitzarem contenen els següents elements:

- Relacions taxonòmiques: relacions de jerarquia del tipus pare-fill.

Una classe pare pot tenir cap o unes quantes classes filles sent aquestes subclasse de la primera (pare). I aquestes a la vegada també poden ser pare, de manera que poden tenir classes filles i així successivament.

Hem de considerar que una classe pot ser subclasse de més d'una classe, és a dir, que un classe pot ser filla de més d'un pare i aquest, per tant pot ser pare de diverses classes filles (herència múltiple).

Aquestes relacions, en OWL [1], es defineix mitjançant l'estructura *subClassOf*.

- Relacions no taxonòmiques: relacions de veïns.

Aquestes relacions són diferents de les anteriors ja que les classes relacionades són de dominis diferents que tenen alguna cosa en comú que les relaciona.

Per exemple, la relació entre la classe *càncer* i la classe *radioteràpia*. *Radioteràpia* no pertany al domini de *càncer* ja que no es cap tipus de càncer, però estan relacionades no taxonòmicament perquè el càncer es tracta amb radioteràpia.

Aquest altre tipus de relació, en OWL [1], es defineix mitjançant l'estructura *subClassOf*, *Restriction* i *onProperty*.

Una classe pot estar enllaçada taxonòmicament i no taxonòmicament al mateix temps, però el que no trobarem és una classe enllaçada amb ella mateixa.

- Classes: les classes poden ser de dos tipus encara que conceptualment són el mateix però es diferencien pel tipus de meta-informació que tenen ja que són d'origens diferents:
 - Classes de relacions taxonòmiques: són classes que representen conceptes i que tenen molts atributs de tipus taxonòmics. La meta-informació que ens interessa a nosaltres i que visualitzarem és:

- *ratio_google*: valor que assigna el constructor d'ontologies per triar o no una classe d'entre les candidates.
- *class_urls_totals*: URLs que retorna el cercador emprat per al concepte. Aquestes URLs no estan etiquetades.
- *class_main_urls*: URLs analitzades segons el domini d'aplicació i etiquetades.
- *num_pages*: número de pàgines web que s'han analitzat per a trobar el resultat.

Aquests tres atributs són bàsics i tenen les seves funcions dins del graf i per tant poden alterar l'aspecte d'aquest segons els seus valors.

En canvi, els atributs que es citen a continuació són atributs extres, purament estadístics que no afecten la visualització del graf.

- *learning_ratio_ant*: ratio d'aprenentatge d'aquesta paraula (tenint en compte els candidats trobats i les classes finalment seleccionades)
- *google_com*: numero de vegades que surt aquesta paraula concatenada amb la seva superclasse a Google.
- *google_ind*: numero de vegades que surt aquesta paraula en Google.
- *num_candidates_ant*: número de candidats que s'han avaluat per a treure les subclasses.

Tota relació no taxonòmica té un verb que relaciona les dues classes implicades en aquesta relació no taxonòmica. Aquest verb té atributs que no afecten a la visualització del graf, són informatius. Aquests són:

- *root_name*: arrel morfològica del verb.
- *levin_class*: tipus de verb segons una classificació estàndard.
- *other_verbs*: altres verbs que pertanyen a la mateixa categorització.
- *thematic_roles*: rol que juguen els elements de la relació expressada pel verb.

- Classes de relacions no taxonòmiques: igualment que les classes taxonòmiques representen conceptes d'altres dominis i tenen també atributs dels quals en depèn la visualització del graf i altres purament estadístics sense cap influència sobre el graf.

Atributs rellevats:

- *ratio_no_taxo*: indica el grau de relació de la classe en base lo inicial.
- *url_relation*: aquí van les URLs d'on s'ha tret aquesta relació. Aquestes URLs no estan etiquetades.

Atributs estadístics:

- *hits_individual_no_taxo*: número de vegada que surt el concepte tot sol.
- *hits_com_no_taxo*: número de vegada que surt el concepte acompanyat del concepte inicial.
- *similarity_no_taxo*: valor utilitzat com a mesura d'avaluació.

Al davant de cada valor d'aquest atributs, surt el concepte de la classe d'on surt la relació i l'etiqueta verbal. Això és per a poder distingir quins atributs corresponen a cada classe quan aquesta classe està relacionada amb diverses relacions no taxonòmiques.

Les classes, en OWL [1], es defineix mitjançant l'estructura *Class*.

- Instàncies: les instàncies ja s'han explicat al segon apartat (capítol 2.1) d'aquest informe que eren, i com les classes, també té la seva meta-informació.

Atributs rellevats:

- *full_name*: nom complet de la instància.
- *conf_instance*: confiança de la instància per triar-la o no.
- *urls_instance*: URLs on s'ha trobat aquesta instància. Aquestes URLs estan etiquetades amb els conceptes que apareixen en aquestes webs.

Atributs estadístics:

- *hits_instance*: número de vegades que s'ha trobat la instància.
- *hits_full_name*: número de vegades que s'ha trobat el *full_name* de la instància, si existís.

Les instàncies, en OWL [1] , amb l'etiqueta *type* s'indica que quina classe són.

3.3 Lectura de les ontologies.

Altres visualitzador com Protégé [3] estan contínuament llegint dades del fitxer OWL per visualitzar-les provocant així, que si l'ontologia és molt gran, el procés d'exploració sigui molt lent.

Per mirar de millorar aquest problema, nosaltres usarem la memòria del ordinador sobre la qual treballarem i així rebrem respostes quasi bé en temps real depenent de la màquina.

Per aconseguir això, primer llegirem el fitxer OWL i l'emmagatzemarem en memòria per no haver de llegir cada vegada el fitxer OWL i obtenir així un major rendiment i major eficiència en temps de càrrega i resposta. Després crearem la jerarquia (relacions taxonòmiques i no taxonòmiques) i emmagatzemarem tota la meta-informació per posteriorment visualitzar-la, guanyant així en temps de resposta a l'hora d'explorar la ontologia.

Com ja sabem tenim dos tipus de fitxers OWL, una versió completa amb meta-informació i un altra neta sense meta-informació, només amb la jerarquia del domini.

La manera de llegir aquest dos fitxer és exactament la mateixa.

Les llibreries de OWL [1] no funcionen quan el contingut dels atributs de les classes apareixen certs caràcter. Es típic que el nom de les URLs continguin caràcters que no són suportats pel charset d'OWL [1] i per això cal filtrar-los. Aquest caràcters són &, < i > sinó pertanyen a les etiquetes XML o formen fletxes(->) i “,” si aquestes es troba en una URL de les pàgines webs.

Per solucionar això hem hagut d'implementar el nostre propi parse d'OWL, capaç de

substituir aquests caràcters que en donen problemes per unes cadenes de text que ens assegurarem que no apareixen en cap fitxer OWL. Els canvis que realitzarem seran:

“-and-“, “-menor-“, “-major-“ i “-coma-“ respectivament. Posteriorment desfarem aquests canvis per poder fer la visualització completa i correctament.

Per llegir el fitxer OWL usarem la API de OWL [1] i una classe creada únicament per aquesta finalitat perquè la API de OWL [1] no proporciona mètodes per obtenir la meta-informació de l'ontologia.

Primerament obtenim una llista de totes les classes de la ontologia amb el mètode *getClasses* i iterarem aquesta llista i per cada classe, obtindrem els seus pares amb el mètode *getSuperClasses*. Aquest mètode ens retorna una llista amb *OWLDataAllRestrictionImpl*, que no usarem, *OWLObjectAllRestrictionImpl*, que representen restriccions sobre classes, que usarem per obtenir la informació dels verb de les relacions no taxonòmiques, i *OWLClassImpl*, que representen a altres classes de la ontologia, que seran els pares de la classe. Quan ens trobem un restricció buscarem la descripció que conté la meta-informació del verb amb el mètode *readDescription* que hem implementat especialment per buscar descripcions d'elements de l'ontologia. Un cop tenim el fragment de la descripció, mitjançant el mètode *readTaxonomicAnnotations*, creat per llegir l'atribut que se li passa per paràmetre a aquesta funció i en cas que a la descripció no existeixi la dada demanada es retorna *null*, obtindrem els valors de tots els atributs.

A mesura que anem descobrint informació, l'anem emmagatzemant en una primera estructura de dades que explicarem en el següent punt d'aquest mateix apartat.

A continuació obtindrem la meta-informació de la descripció de la classe i anirem llegint i emmagatzemant totes les dades taxonòmiques d'aquesta a la primera estructura de dades. Després farem els mateix amb la meta-informació no taxonòmica de la classe amb un altre mètode anomenat *readNoTaxonomicAnnotations*.

Finalment usarem el mètode *getUsage* que retorna una llista de *OWLIndividualImpl*, que representa una instància i *OWLObjectPropertyImpl*, que tampoc usarem. Si trobem una instància llegirem la seva meta-informació amb el mètode *readTaxonomicAnnotations*.

El mètode *readTaxonomicAnnotations* es fa servir per llegir la meta-informació taxonòmica de les classe i de les instàncies.

```

/**
 * @param desc : String amb la descripció de la OWLClass.
 * @param cl : nom complet de la classe.
 * @param Annotation : l'atribut en concret que volem llegir
 * @param clPad : nom de la classe a seques.
 * @return contenido : resultat de la consulta.
 */
public String readTaxonomicAnnotations(String desc ,String cl,
                                     String Annotation,String clPad)
{
    int k2 = -1;
    String contenido = null;
    quitarNumero qN =new quitarNumero();
    if ( desc.indexOf(cl) != -1 )
    {
        if (qN.quitarNum(cl).compareTo("#"+clPad) != 0)
        {
            contenido = null;
            k2 = desc.indexOf("<a:"+Annotation+">");
            if ( k2 >= 0 )
            {
                int k3 = desc.indexOf("</a:"+Annotation+">");
                if ( k3 >= 0 )
                {
                    int k4 = ("</a:"+Annotation+">").length()-1;
                    if ( k3 > k2 ) {
                        contenido = desc.substring(k2+k4,k3);
                    }
                }
            }
            else {
                //System.out.println(Annotation+" not found!");
                return null;
            }
        }
        else System.out.println("Descripció incorrecte!!");
    }
    return contenido;
}

```

El mètode busca dins d'un fragment de text, corresponent a la descripció de la classe o de la instància, les etiquetes de les dades que es desitgen. Els valors d'aquesta meta-informació la trobarem dins l'ontologia entre etiquetes de XML. Per exemple, per obtenir el *ratio_google* de la classe *biosensor* invocarem el mètode de la següent manera:

```

String description = rO.readDescription(file,"#+cl,clPad,false);
s=rO.readTaxonomicAnnotations(description,"#+cl,"ratio_google",clPad)

```

,on *rO* és una instància de la classe *ReadOWL*, que és la classe on s'han implementat aquests mètodes extres de lectura.

La crida per la lectura de dades no taxonòmiques és exactament igual, però invocant la funció pertinent.

```

/**
 * @param desc : String amb la descripcio de la OWLClass.
 * @param cl : nom complet de la classe.
 * @param Annotation : l'atribuit en concret que volem llegir
 * @param clPad : nom de la classe a seques.
 * @return contenido : Vector amb els resultats de la consulta.
 */
public Vector<String> readNoTaxonomicAnnotations(String desc ,String
                                                cl,String Annotation,String clPad)
{
    int k2 = -1;
    boolean ord = false;
    Vector<String> vect =new Vector<String>();
    //String contenido = null;
    quitarNumero qN=new quitarNumero();
    if ( desc.indexOf(cl) != -1 )
    {
        if (qN.quitarNum(cl).compareTo("#"+clPad) != 0)
        {
            //contenido = null;
            k2 = desc.indexOf("<a:"+Annotation+">");
            int k4 = -1, k3 = -1;
            if ( k2 >= 0 )
            {
                k3 = desc.indexOf("</a:"+Annotation+">");
                if ( k3 >= 0 ) {
                    k4 = ("</a:"+Annotation+">").length()-1;
                    if ( k3 > k2 ) vect.add(desc.substring(k2+k4,k3));
                }
            }
            }else return null;

            // busquem si hi ha mes atributs
            String Saux = desc.substring(k3+k4);
            k2 = Saux.indexOf("<a:"+Annotation+">");
            if ( k2 > 0 ) {
                ord = true;
                while ( k2 > 0 ) {
                    k3 = Saux.indexOf("</a:"+Annotation+">");
                    if ( k3 >= 0 ) {
                        k4 = ("</a:"+Annotation+">").length()-1;
                        if ( k3 > k2 )
                            vect.add(Saux.substring(k2+k4,k3));
                    }
                    Saux = Saux.substring(k3+k4);
                    k2 = Saux.indexOf("<a:"+Annotation+">");
                }
            }
        }
    }
}

```

```

        }else System.out.println("Descripció incorrecte!!");
    }
    if ( ord ) Collections.sort(vect);
    return vect;
}

```

Prèviament, però, s'ha d'invocar el mètode *readDescription* indicant-li si es busca la descripció d'una classe o d'una instància amb un booleà.

```

/**
 * @param s : String amb el fitxer OWL parsejat.
 * @param cl : nom complet de la classe.
 * @param clPad : nom de la classe a seques.
 * @param inst: indicador de si és busca un instància o no.
 * @return contenido : String amb el resultat de la consulta.
 */
public String readDescription(String s, String cl, String clPad,
                             boolean inst)
{
    String contenido = null;
    quitarNumero qN =new quitarNumero();
    if ( s.indexOf(cl) != -1 )
    {
        if (qN.quitarNum(cl).compareTo("#"+clPad) != 0)
        {
            contenido = null;
            String Saux = s.substring(s.indexOf(cl));
            String Saux2 = null;
            if ( Saux.indexOf("/rdf:Description") != -1 )
            {
                if ( !inst )
                    Saux2=Saux.substring(0,Saux.indexOf("/rdf:Description")+16);

                else Saux2 = Saux;
                String Saux3 = Saux2.substring(Saux2.indexOf(cl)+20);
                Saux3 = Saux3.substring(Saux3.indexOf("rdf:Description"));
                if ( Saux3.contains(cl+"\"") ) contenido = Saux3;

            }else
                // recursivitat
                contenido=readDescription(Saux.substring(20),cl,clPad,inst);
            }
        }
    }
    return contenido;
}

```

3.4 Estructura de dades

Per tal d'emmagatzemar tota la informació que ens interessa visualitzar de l'ontologia a memòria i poder consultar-la amb un bon rendiment, s'ha definit una estructura de dades apropiada.

Definim dos tipus d'estructures de dades: un que és el mapeig, de la ontologia del fitxer OWL que volem visualitzar, sobre memòria, i l'altre estructura de dades és la que defineix tota la lògica de l'aplicació que també tindrà una part per representa totes les classes, instàncies i tot tipus de relacions aprofitant la primera estructura de dades.

Un cop tenim tot el fitxer mapejat a memòria en una llista, crearem la segona i definitiva estructura de dades que serà amb la que treballarem en temps d'execució i que ens permetrà un ràpida interoperabilitat amb l'usuari. Aquesta consisteix en anar llegint la primera llista que conté les dades del fitxer OWL i crear nous elements que representaran les dades obtingudes anteriorment. Aquest elements que representaran classes i instàncies s'emmagatzemaran en un altra llista.

Totes dues estructures de dades són llistes. S'ha decidit emmagatzemar la informació en *LinkedList* perquè són col·leccions de dades que sofreixen poques alteracions, és a dir, poques insercions, eliminacions i modificacions. Però en canvi, si que es realitzaran moltes consultes i recorreguts sobre les col·leccions. Per tant, les llistes enllaçades són la millor opció en front a *ArrayList* o *Vector* de Java.

També tindrem la representació dels verbs de les relacions no taxonòmiques. Aquests s'emmagatzemaran en un mapa d'elements clau-valor, on la clau serà el nom del verb i el valor serà la classe Java implementada que els representa. S'ha decidit emmagatzemar-ho a part en una altra estructura de dades perquè moltes relacions no taxonòmiques usen sempre els mateixos verbs per relacionar classes i per tal de no repetir informació en cada classe o relació s'ha optat per aquesta solució. I s'ha fet en un *HashMap* perquè d'aquesta manera només indicant-li en nom del verb podem obtenir eficientment tota la informació d'aquest i perquè no accepta claus repetides de manera que no ens hem de preocupar de si ja existeix, a la estructura, el verb que volem d'inserir cada vegada que llegim una relació del fitxer OWL.

Una última estructura de dades genèrica que tenim és un conjunt de classes. Aquesta serveix per apuntar-nos quines classes en cada moment són classes principal en la visualització, és a dir, quines són aquelles classes visualitzen taxonomies sense estar relacionades amb la resta. S'implementarà en forma de *HashSet* ja que no permet tenir elements repetits i així aconseguim delegar el control de repetits a Java.

Aquestes són les estructures de dades principals de l'aplicació per emmagatzemar les dades que volem visualitzar, però també tenim altres elements que ens ajudaran a saber l'estat del graf en cada moment i que vol l'usuari. Aquests elements són simples flags que ens indicaran com vol explorar la jerarquia de classes creada, taxonòmicament o no taxonòmicament i si s'aplica algun tipus de filtre dels que es proporciona.

A continuació es mostra un esquema (fig. 3.3) de les classes implementades que representen cada concepte OWL que tenim per representar.

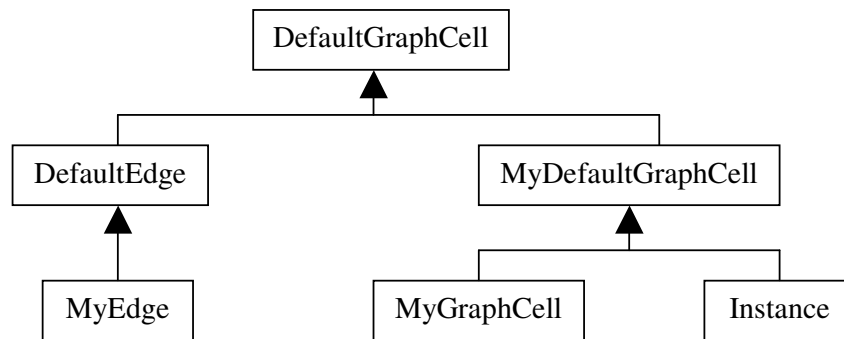


Fig. 3.3: Diagrama de les classes implementades

Les classes Java *DefaultEdge* i *MyEdge* representa les arestes de les relacions taxonòmiques i no taxonòmiques respectivament. La classe Java *MyGraphCell* representa les classes i *Instance* representa les instàncies de la ontologia.

Un cop tenim totes les dades emmagatzemades procedirem a crear la jerarquia i tot seguit a construir les relacions no taxonòmiques.

Aquesta jerarquia de classes es crea de la següent manera: a cada instància de la classe *MyGraphCell*, tenim una llista de fills i un camp que ens indica quina classe és el seu pare. Doncs bé, un cop sabem això el procediment a seguir és molt senzill perquè només hem hagut de recórrer la llista genèrica de classes on tenim totes les dades

emmagatzemades i mirant per cada classe de quina es filla i un cop sabem això només l'hem d'afegir a la llista de fills de la classe pare i procedir a crear la jerarquia d'aquesta classe filla recursivament de la mateixa manera, i així successivament fins a tenir tota la taxonomia creada. Això comporta moltes passades sobre la llista genèrica però com que estem treballant sobre memòria i no des de el fitxer OWL, el procediment és bastant ràpid. Per aconseguir aquesta rapidesa i per fer més eficient l'aplicació, la llista de fills contindrà punters a classes *MyGraphCell*, així no repetim informació i no ocupem espai innecessàriament. Igualment el camp que indica el pare de la classe, també serà un punter a dita classe pare.

Un cop ja tenim la taxonomia creada continuarem amb la creació de les relacions. Això es fa de la mateixa manera ja que també tenim, a la classe *MyGraphCell*, una llista corresponent als seus amics, entenent per amics les classes amb les que està relacionada, i en la classe *MyDefaultGraphCell*, un camp que ens indica amb quines classes està relacionada i amb quin verb. La col·lecció d'amics, lògicament també serà de punters a classes *MyGraphCell* i s'implementarà en una llista perquè es fan molt accessos als elements de la col·lecció i es fan poques modificacions, a la vegada que és l'ho suficient robusta.

Aquesta llista d'amics, és una *LinkedList* de *Property* que és la classe que representa les relacions no taxonòmiques, tenint com a atributs de classe una instància de *MyEdge* i dos punters de *MyGraphCell* (origen i destí de la aresta).

Com es veurà en el següent punt les classes i instàncies es visualitzen de diferent manera. Per tal d'aconseguir això s'ha d'implementar una vista per a cada concepte i una nova factoria de vistes. Així si l'objecte és una instància es crearà una vista determinada per a les instàncies i se l'associarà, i si és una classe es crearà una vista de classe i, de la mateixa manera, se li associarà a la classe.

Per tant, creem una nova *DefaultCellViewFactory* que farà aquesta feina. A més de crear la nova factoria de vistes, també hem d'implementar aquestes vistes.

La vista per a les classes serà la que tenim per defecte a la llibreria JGraph [15] però per a les instàncies crearem una de nova. Llavors, tindrem una classe *InstanceView* que estén de *VertexView* (implementació per defecte de JGraph). A aquesta classe se li ha de crear un *renderer*, que serà l'encarregat de fer el dibuix, per tant també implementem un *InstanceRender*.

El *renderer* serà una instància de la classe interna *InstanceRenderer*, ja que s'usarà exclusivament en la classe *InstanceView*, que estén de *VertexRenderer*. En aquesta classe interna sobreescrivem el mètode *paint*, on definirem la forma d'el·lipse que tindrà la instància, i el mètode *getPreferredSize* per obtenir la dimensió de la figura.

3.5 Visualització

Per a dur a terme la visualització de les ontologies, farem una representació de tipus arbre, en la que podrem expandir i col·lapsar classes, mostrant i amagant així els fills o les relacions no taxonòmiques que aquestes puguin tenir.

Estarà centrada en un element principal a partir del qual s'anirà expandint i creant tot el graf al seu voltant.

Per aconseguir una graf visualment ric, farem servir el màxim de meta-informació disponible de cada element.

Es pretén visualitzar el màxim d'informació de la ontologia en aspectes visuals, a part de mostrar aquests valors textualment.

Les classes s'han visualitzat en forma de rectangles amb el nom de la classe a l'interior d'aquest en minúscules i en negreta i la icona . Per altra banda, les instàncies es representen en forma d'el·lipses amb el nom a l'interior però en majúscules i en cursiva. Igual que les classe, les instàncies també tenen el seu icona: .

Tant les classes com les instàncies es coloren. El colors depèn del valor *ratio_google*, en el cas de les classes, i *conf_instance* en el cas de les instàncies. Les escales de colors són diferents: per a les classes s'utilitza una escala que va des del vermell cap al verd (fig. 3.4) i per les instàncies l'escala va del gris fosc al cian (fig. 3.5) . Això pretén mostrar visualment el grau de rellevància de les classes i instàncies per al domini.



Fig. 3.4: Escala de color de les classes.



Fig. 3.5: Escala de color de les instàncies.

Se suporta de forma transparent la presència o absència de qualsevol meta-dada, ja que tota meta-dada té un valor per defecte i conseqüentment una visualització. Per exemple, tant si el *ratio_google* com el *conf_instance*, no existeix o no té valor, la classe o instància es pinta de color gris clar, com es pot observar a la següent figura (fig. 3.6).

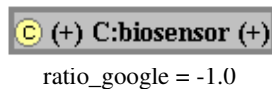


Fig. 3.6: Classe colorada per defecte.

Les classes també tenen un marc del mateix color amb el que s'ha pintat la classe però enfosquit (fig. 3.7). Aquest marc dóna una pista del valor de l'atribut *num_pages*. Això representa la quantitat d'anàlisi dedicat a analitzar les relacions d'una classe (nombre de webs analitzades).

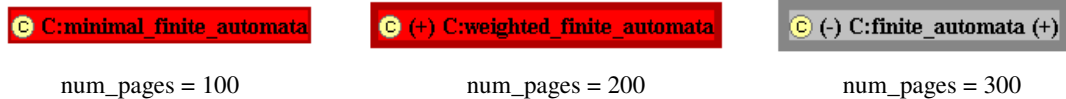


Fig. 3.7: Diferents gruixos de marc de classes.

Les classes també poden tenir dos símbols, un a cada costat del rectangle o del nom de la classe (fig. 3.8). Aquest símbol és (*), on * és o bé + o bé un -. El símbol + indica que la classe està col·lapsada i per tant que es pot expandir mostrant-se així els seus fills o amics. En canvi, el símbol - indica tot el contrari, és a dir, que la classe està expandida i per tant és pot col·lapsar amagant així els seus fills o amics. Si el símbol està a l'esquerra vol es refereix a la part taxonòmica i si el símbol es troba a la dreta es refereix a la part no taxonòmica.

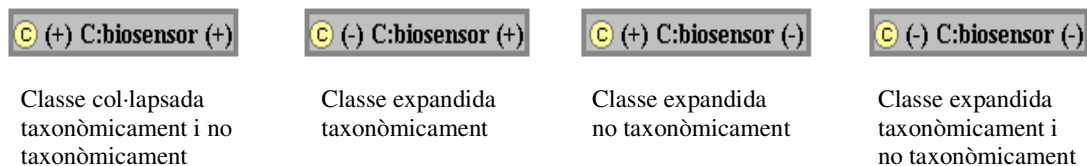


Fig. 3.8: Classe amb els diferents estats possibles.

Tots els valor de les classes i instàncies es mostren textualment en un pannel informatiu a la part baixa de la pantalla, quan el passa el punter per damunt d'aquestes, excepte el *full_name* de les instàncies que es mostra a la barra d'estat. En el cas de les classes, a aquesta barra es visualitza el nombre de relacions taxonòmiques, d'instàncies i de relacions no taxonòmiques que la classe té.

Per tal de diferenciar els diferents tipus de relacions, l'aresta que uneix la classe pare amb els seus fills, és un aresta normal sense cap punta i de colors diferents: si el fill és una classe, l'aresta serà de color blau i si el fill és una instància aquesta aresta serà de color cian. Veiem un exemple a continuació (fig. 3.9).

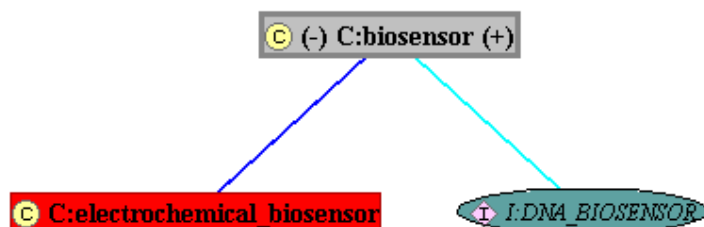


Fig. 3.9: Tipus d'arestes de relacions taxonòmiques.

Pel que fa a les relacions no taxonòmiques d'una classe, aquestes es visualitzen en forma de fletxes amb una punta plena a la classe destí. Aquestes fletxes són totes del mateix color, blau fosc, però són de diferents gruixos depenent del valor del *ratio_no_taxo*, representant així la rellevància de la relació que uneix les classes. Aquestes fletxes estan etiquetades amb el nom del verb, en minúscula i en negreta, que relaciona les dues classes. Veiem un exemple a la següent figura (fig. 3.10).

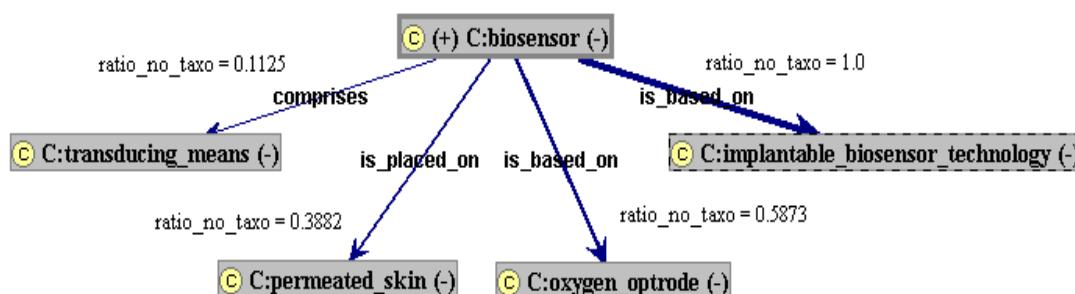


Fig. 3.10: Diferents gruixos de les relacions no taxonòmiques.

Es pot donar el cas que una classe tingui diverses relacions no taxonòmiques amb una classe, llavors es mostra les fletxes superposades però els verbs separat al llarg de les fletxes, de tal manera que es pot veure que hi ha diverses relacions. Llavors es pot separar les fletxes superposades amb el botó dret del ratolí aconseguint una forma corbada de la fletxa. Un cop separades es podem tornar a unir clickant amb el mateix botó dret del ratolí sobre el nous punts creat a la fletxa. Sempre que la classe es col·lapsa, quan la tornem a expandir les fletxes apareixeran superposades de nou. Val a dir que les etiquetes suporten drag&drop i per tant les podem col·locar on vulguem.

En el següent exemple (fig. 3.11) es pot veure com es pot passar d'un estat a l'altre com s'ha explicat.

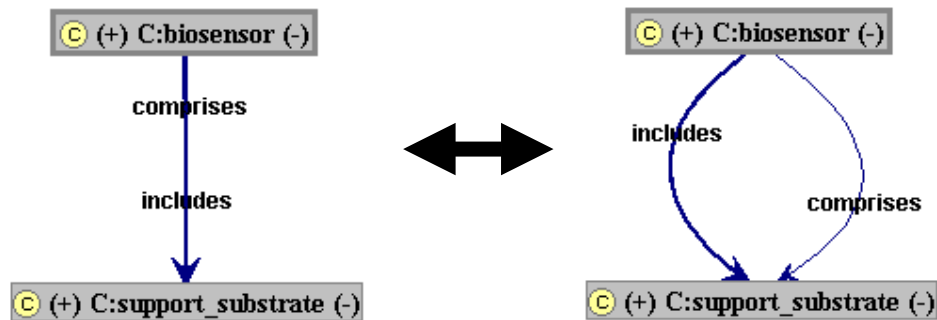


Fig. 3.11: Estats de les fletxes.

Si es passa el punter per damunt d'una fletxa aquesta visualitza tota la seva meta-informació al pannel informatiu que tenim a la part baixa de la pantalla i si la seleccionem ens mostra al segon pannel informatiu, situat a la l'esquerra del primer pannel, tota la meta-informació relativa al verb d'aquella relació no taxonòmica (fig.3.12).

· Ratio noTaxo: 0.14876033 · Similiary: -1.0 · Hits Com.: 1260 · Hits Ind: 8470 · Hits Com.: 1260	· VERB: includes · Other verbs: · Levin class: · Thematic role: · Root name: include
---	--

Fig. 3.12: Visualització de la meta-informació del verb.

De la mateixa manera si el passa el punter per damunt d'una classe (fig.3.13) o instància (fig. 3.14) també es mostra, en el pannell informatiu la meta-informació corresponent.

Ratio Google: -1.0 Candidates: 91 Pages: 140 Learning Ratio: 0.21978022 Google Ind.: 64000 Google Com.: -1		Confidence: 100.0 Hits Instances: 23 Hits FullName: 1	
---	--	---	--

Fig. 3.13: Meta-informació de classe.

Fig. 3.14: Meta-informació d'instància.

El punter per definició és la fletxa del mateix sistema operatiu de l'ordinador però quan es passa per damunt d'algun element gràfic del graf (classe, instància o relació no taxonòmica) aquest canvia i es transforma en una mà, com la usada per qualsevol navegador d'Internet quan es passa per un enllaç web. Aquesta mà ens serveix per indicar-nos que l'element en qüestió suporta algun tipus d'interacció, ja sigui *drag&drop* o l'edició de fletxes implementada.

Sempre que seleccionem un element de la visualització ja sigui una classe, instància o fletxa, es mostren totes les pàgines webs associades al element seleccionat en una llista situada a la dreta del primer pannell informatiu. I si se selecciona una d'aquestes pàgines webs, a la barra d'estat es mostra una sèrie d'etiquetes, si existeixen per a aquella pagina web, que ens indiquen quins temes tracten en aquella pàgina web (fig. 3.15) . I fent dobleclick obrim aquella pàgina web amb el explorador que tinguem instal·lat al nostre ordinador.

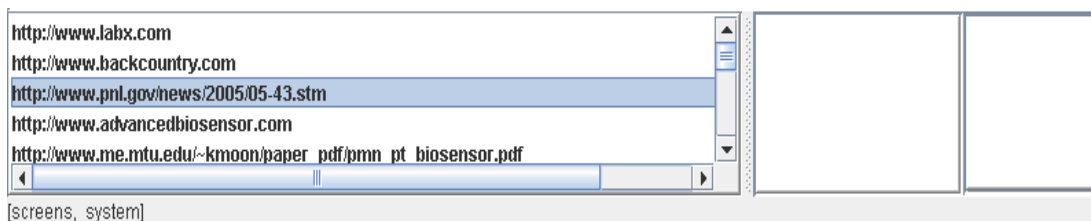


Fig. 3.15: Visualització de la llista de pàgines webs.

Donat que l'ontologia pot ser molt gran, s'ha implementat una sèrie de filtres perquè l'usuari pugui escollir les classes o instàncies que vol veure i així minimitzar la quantitat d'informació que es mostra.

Aquests filtres són bàsicament tres: el primer filtre consisteix en què només es visualitzaran les classes en el moment d'expandir la classe desitjada. El segon filtre visualitzarà només les instàncies de la classe en el moment d'expandir-la. I el tercer, s'aplica juntament amb un o els dos filtres anterior i consisteix en definir un llinar a partir del qual es visualitzaran les classe i/o instàncies que superin aquest llinar amb el valor de l'atribut *ratio_google* i *conf_instance* respectivament.

3.6 Tecnologia Web

El visualitzador s'ha implementat amb un applet de Java per tal de poder executar-lo a qualsevol màquina, independentment de la plataforma, que tingui una JVM i un explorador d'Internet. Un altre raó és perquè així es podrà integrar amb la resta del sistema constructor que està basat en una interfície web. També aconseguirem tenir un accés directe al navegador web a partir dels vincles webs inclosos com a meta-informació a l'ontologia.

Un applet és una petita aplicació, formada per una o varies classes escrites en Java, que està inserida en una pàgina web. D'aquesta manera no cal cap tipus d'instal·lació, simplement connectar-nos a la màquina oportuna ens descarreguem l'aplicació per executar-la localment a la nostra màquina.

Els applets es construeixen usant un marc de treball d'aplicació. S'hereta de JApplet i es sobreescrueixen els mètodes apropiats. Hi ha uns mètodes que controlen la creació i execució d'un applet en una pàgina web:

- *init*: s'invoca automàticament per a realitzar la primera inicialització de l'applet, incloent la disposició dels components.
- *start*: s'invoca cada vegada que es visualitza un applet al navegador per a permetre començar les operacions normals. També s'invoca després de l'*init*.

- stop: s'invoca cada vegada que un applet s'aparta de la vista d'un navegador web per permetre l'applet apagar operacions costoses. S'invoca automàticament abans del mètode destroy.
- destroy: s'invoca cada vegada que s'està descarregant un applet d'una pàgina par a dur a terme la alliberació final de recursos quan es deixa d'utilitzar l'applet.

El fet de treballar amb applets ens limita l'accessibilitat a la màquina local que l'executa per motius de seguretat i per resoldre això s'ha de firmar un certificat per tal que l'aplicació (applet) pugui llegir i escriure al sistema de fitxers de la màquina on s'executa.

Per tant tenim una pàgina web que conté un applet amb el nostre visualitzador.

A continuació, a la figura 3.16, es mostra l'aparença que tindrà el visualitzador a l'inici indicant tots els elements que tindrà l'aplicació.

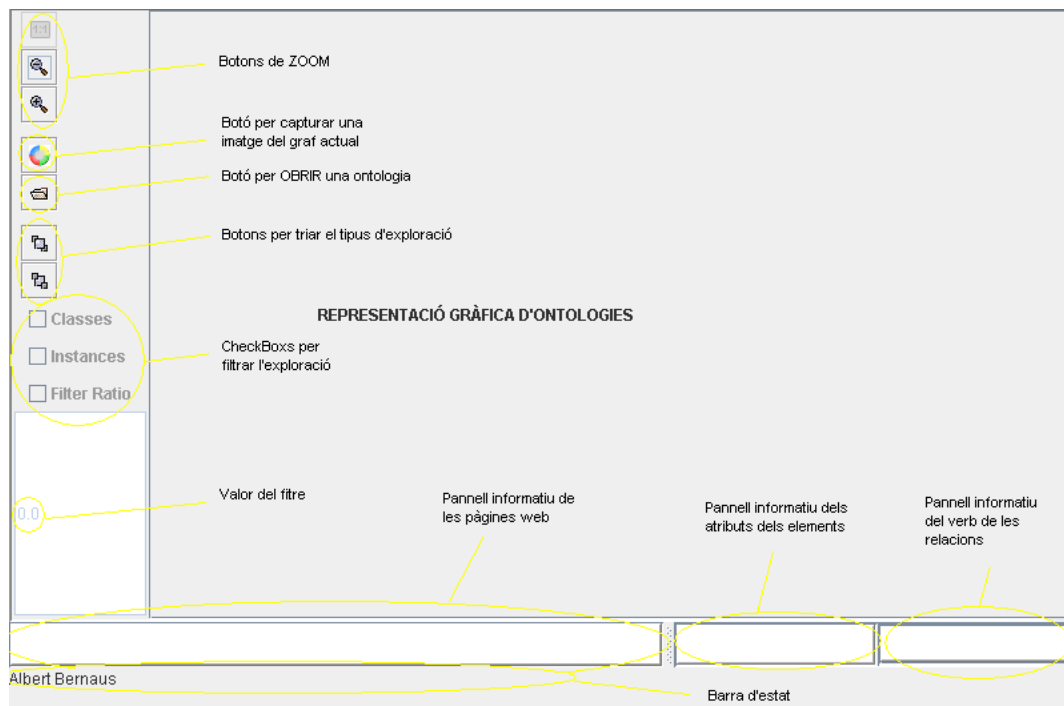


Fig. 3.16: Interfície gràfica del visualitzador.

4. JOC DE PROVES

S'ha realitzat un joc de proves per comprovar que es compleixen tots els objectius fixats a l'inici del projecte.

Usarem les següents ontologies amb els dos tipus de fitxers que tenim de cada una, la versió FULL i la versió CLEAN: *biosensor* i *context_free_grammar*.

Primer començarem avaluant la visualització del fitxers FULL, ja que són els que contenen tota la meta-informació i per tant la visualització és molt més rica que una visualització de la mateixa ontologia però amb la versió CLEAN.

4.1 Prova 1: Exploració taxonòmica.

Primer obrim el fitxer *biosensor_1140513491062_ANT.owl* per a comprovar l'exploració taxonòmica només de classes (fig. 4.1), només d'instàncies (fig. 4.2) i de classes i instàncies conjuntament (fig. 4.3) seleccionant els *checkboxs* que tenim a la *toolbar*.

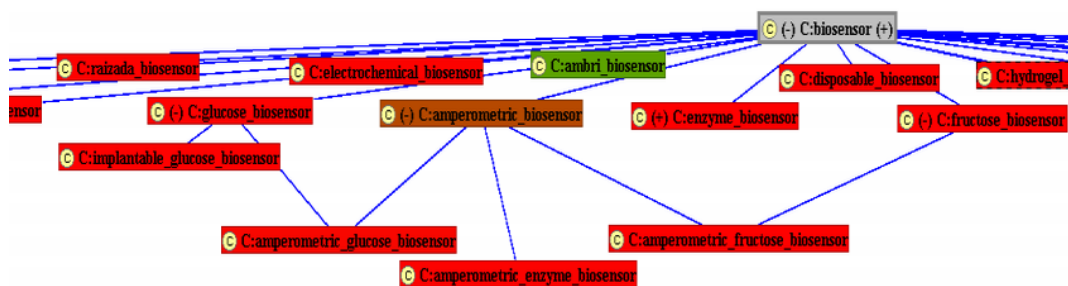


Fig. 4.1: Exploració taxonòmica de classes.

A la fig. 4.1 es pot veure diverses coses. Per començar es pot veure la jerarquia de classes de la ontologia *biosensor*. Es pot comprovar com les classes filles son de diferents colors depenent dels valors *ratio_google* de cada classe. Es pot veure una classe (*enzyme_biosensor*) que té el símbol (+) al davant indicant que està col·lapsada. Hi ha altres que tenen el símbol (-) que són les que veiem que estan expandides

(*glucose_biosensor*, *fructose_biosensor* i *amperometric_biosensor*). Hi també veiem altres classes que no tenen cap dels dos símbols, això vol dir que no té fills.

Si ens fixem en la classe *amperometric_biosensor*, veiem que està expandida i que té 3 fills: *amperometric_glucose_biosensor*, *amperometric_enzyme_biosensor* i *amperometric_fructose_biosensor*. Però el que es posa de manifest aquí és l'herència múltiple que suporta el visualitzador ja que *amperometric_glucose_biosensor* també és fill de la classe *fructose_biosensor* que a la vegada també té un altre fill que és *implantable_glucose_biosensor*. De la mateixa manera *amperometric_biosensor* també comparteix una altre classe amb *fructose_biosensor*.

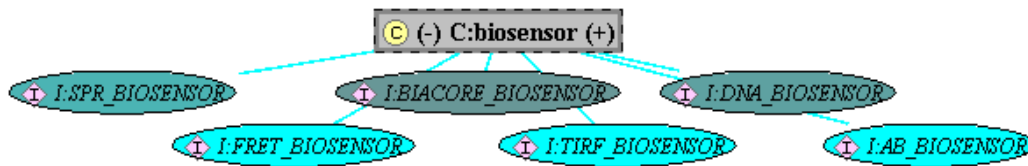


Fig. 4.2: Exploració taxonòmica d'instàncies.

A la fig. 4.2, es veu clarament la representació de les instàncies que té la classe *biosensor*. De la mateixa manera que les classes les instàncies també tenen diferents colors seguint l'escala establerta.

Igual que a la fig.1 es pot observar que la classe *biosensor* té el símbol (+) a la part dreta de la classe indicant que té relacions no taxonòmiques però que estan col·lapsades.

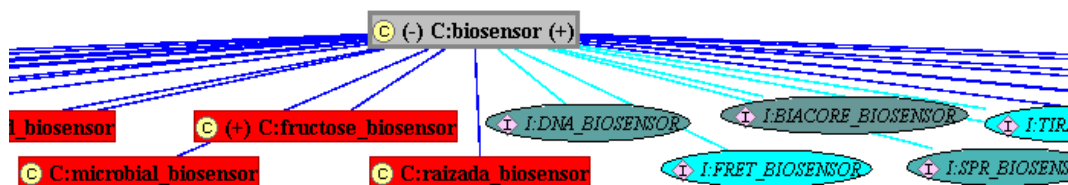


Fig. 4.3: Exploració taxonòmica completa

Finalment a la fig. 4.3 es veu com es pot visualitzar a la vegada les classes i instàncies d'una classe determinada.

4.2 Prova 2: Filtrar informació taxonòmica.

Continuant amb l'exploració taxonòmica aplicarem un filtre sobre l'atribut *ratio_google* de les classes i *conf_instance* de les instàncies.

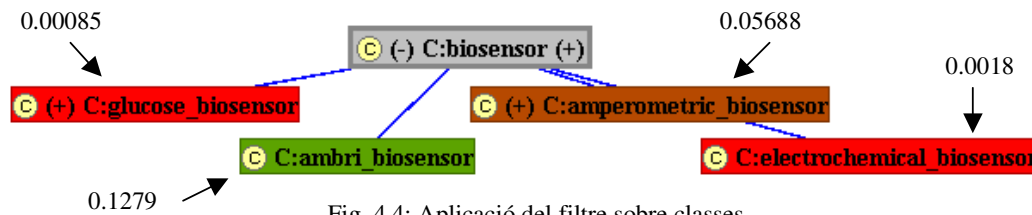


Fig. 4.4: Aplicació del filtre sobre classes

A la fig. 4.4 s'ha aplicat un filtre amb llindar 0.0008. Només les classes amb un valor de l'atribut *ratio_google* igual o major a aquest han estat visualitzades. I com es pot veure es mostra amb un requadre el valor de cadascuna de les classes per comprovar que compleixen la condició.

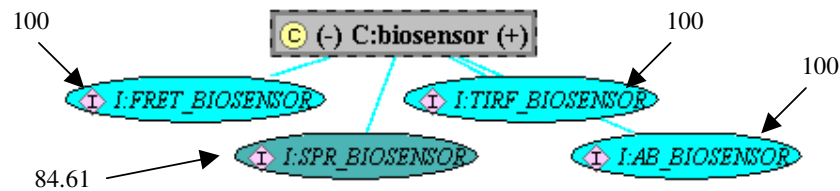


Fig. 4.5: Aplicació del filtre sobre instàncies

A la fig. 4.5 s'ha aplicat el filtre sobre les instàncies amb valor 82. totes les instàncies amb un valor de l'atribut *conf_instance* igual o major a 82 han estat visualitzades.

4.3 Prova 3: Exploració no taxonòmica.

Ara comprovarem l'exploració no taxonòmica de les classes seleccionant l'opció d'exploració no taxonòmica que tenim a la *toolbar*.

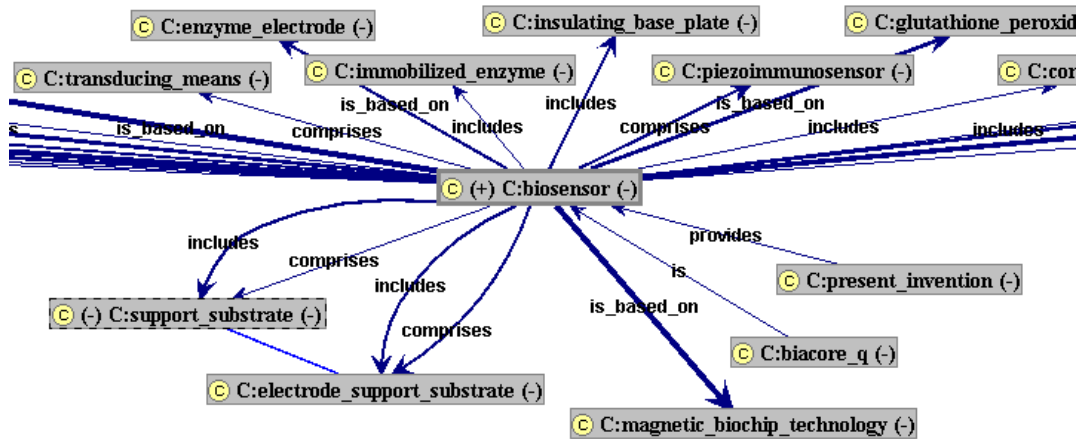


Fig. 4.6: Exploració no taxonòmica

A la fig. 4.6 es pot veure les relacions no taxonòmiques que té la classe *biosensor*. Com sabem aquestes relacions representen unint les classes amb fletxes i com podem observar aquestes fletxes són de diferents gruixos. Aquests gruixos depenen del valor de l'atribut *ratio_no_taxo* de la relació.

Com que hem expandit la classe *biosensor* no taxonòmicament, el símbol que hi ha a la dreta de la classe és (-).

Totes les classes són de color gris clar, això es perquè no tenen cap valor de *ratio_google* ja que, en aquest cas, totes les classes són no taxonòmiques o les que si ho són (*support_electrode*) no tenen atributs taxonòmics.

També podem observar que tenim relacions entrants, com és el cas de la classe *present_invention* i *biacore_q*, i sortints a les diferents classes indicades pel sentit de la fletxa.

Una altra detall que podem observar és que les fletxes es poden modelar al gust de l'usuari per aconseguir una visualització més clara del graf ja que normalment són bastant grans.

També podem veure que les classe també es podem relacionar entre elles com és el cas de la classe *electrode_support_substrate* està relacionada taxonòmicament amb *support_substrate* ja que és filla seva.

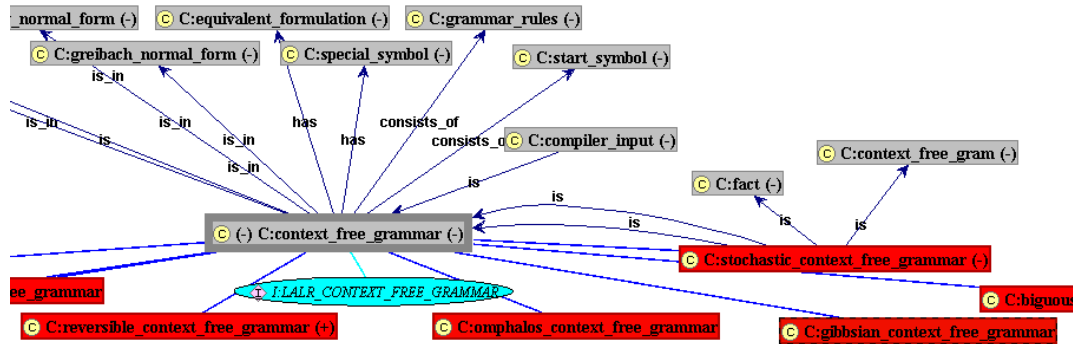


Fig. 4.7: Exploració completa d'una ontologia

A la fig. 4.7 es mostra l'exploració completa d'una ontologia, és a dir, l'exploració taxonòmica i no taxonòmica, en aquest cas, de la classe *context_free_grammar*.

Com que l'hem expandit taxonòmicament i no taxonòmicament, la classe té els símbol (-) a tots dos costats.

Context_free_grammar té varies relacions no taxonòmiques, tant entrants com sortints i fills que són classes i instàncies.

Totes les classes tenen un marc del mateix color que la classe però més fosc. El gruix d'aquests marc depèn del valor de l'atribut *num_pages*. A la classe *context_free_grammar* es pot veure que té un marc bastant gruixut ja que té 300 pàgines, en canvi la resta de les classe només tenen 100 pàgines i el marc és bastant més prim.

En aquesta imatge es pot veure com una classe es pot ser a la vegada taxonòmica i no taxonòmica. És el cas de la classe *stochastic_context_free_grammar*, que està relacionada no taxonòmicament amb *context_free_grammar*, però que a la vegada també es filla d'aquesta classe i que a la vegada també té relacions taxonòmiques sortints cap a les classes *fact* i *context_free_gram*.

S'ha implementat un algorisme especial per tal de controlar la correcta col·lapsació del element.

Si *stochastic_context_free_grammar* només estigués enllaçada taxonòmicament amb *context_free_grammar* i col·lapséssim aquesta, *stochastic_context_free_grammar* desapareixeria del graf. Per altra banda, si *stochastic_context_free_grammar* només estigués enllaçada no taxonòmicament amb *context_free_grammar* i col·lapsem aquesta no taxonòmicament també desapareixeria. Però en aquest cas tenim que *stochastic_context_free_grammar* està enllaçada tant taxonòmicament com no taxonòmicament, llavors veiem que si col·lapsem *context_free_grammar* taxonòmicament, la classe *stochastic_context_free_grammar* no hauria de desaparèixer perquè tenim la classe *context_free_grammar* expandida taxonòmicament i per tant enllaçada amb *stochastic_context_free_grammar*. En aquestos casos només les arestes o fletxes de les relacions no taxonòmiques es el que desapareix del graf.

Igualment passa alhora d'expandir un classe, que només es dibuixa l'aresta o la fletxa de la relació no taxonòmica si la classe ja està al graf per altres relacions taxonòmiques o no taxonòmiques amb altres classes de l'ontologia.

Aquest no és un problema gens trivial i per això hem hagut d'implementar un algorisme de col·lapsació especial.

4.4 Prova 4: Filtrar informació no taxonòmica.

Seguint amb l'exploració no taxonòmica aplicarem un filtre sobre l'atribut *ratio_no_taxo* de les relacions no taxonòmiques.

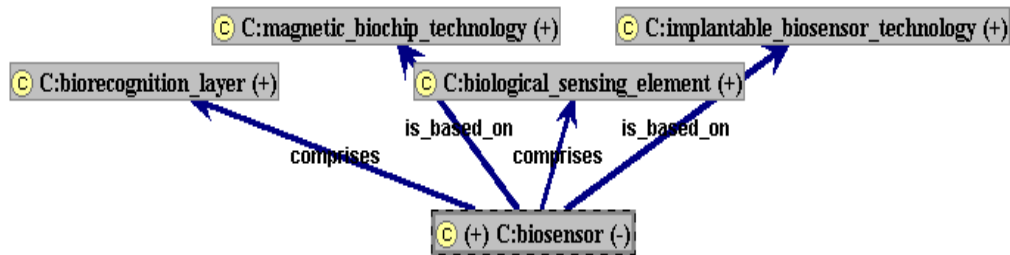


Fig. 4.8: Aplicació del filtre sobre les relacions no taxonòmiques.

En aquesta figura (fig. 4.8) es mostren només les relacions no taxonòmiques que igualen o sobrepassen el llindar introduït al filtre. En aquest cas s'ha establert un llindar de 0.75.

4.5 Prova 5: Classes mestres.

Si partint de l'inici, carreguem la ontologia *context_free_grammar*, l'expandim no taxonòmicament (fig. 4.9) i a continuació expandim i col·lapsem la classe *stochastic_context_free_grammar* (fig.4.10), ens trobem que tenim dos classes mestres (fig.4.12) i per tant dues ontologies separades que es podem explorar independentment una de l'altre.

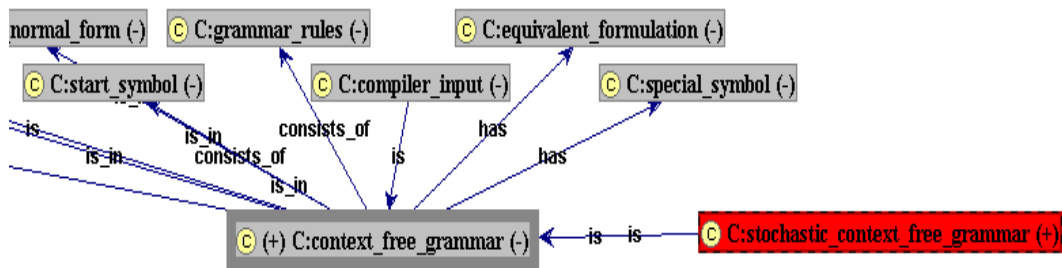


Fig. 4.9: Expansió no taxonòmica de la classe *context_free_grammar*.

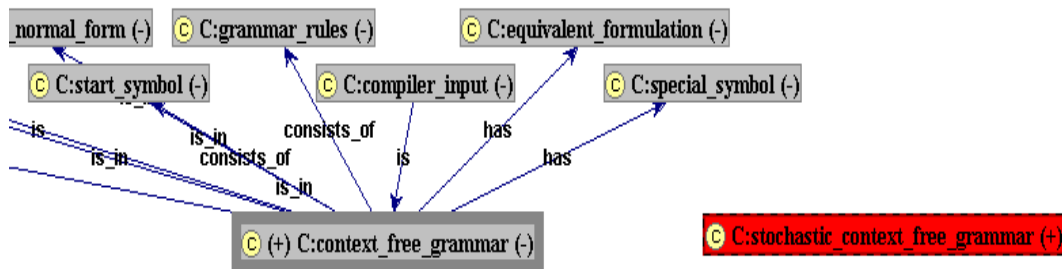


Fig. 4.10: Col·lapse de la classe *stochastic_context_free_grammar*.



Fig. 4.12: Col·lapse de la classe *context_free_grammar*.

4.6 Prova 6: Responent als events del ratolí.

Avaluarem les respostes que rebem quan interactuem sobre qualsevol dels elements que tenim al graf a través del ratolí.

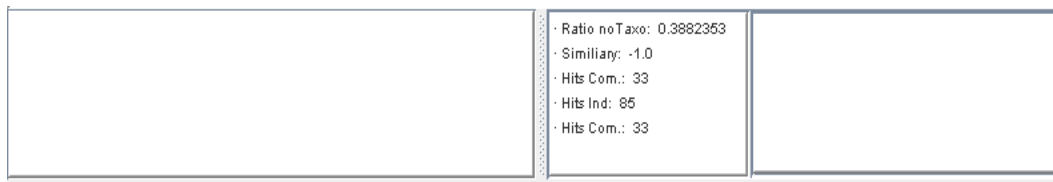


Fig. 4.13: Informació d'una relació no taxonòmica.

A la figura 4.13 podem veure com es mostra la informació que tenim de les relacions no taxonòmiques quan es passa amb el punter per damunt d'una d'aquestes relacions.

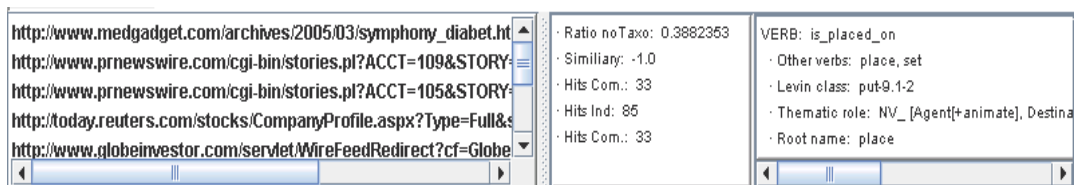


Fig. 4.14: Informació d'un ver d'una relació no taxonòmica.

I a la figura 4.14 es mostra la informació que es visualitza quan se selecciona una d'aquestes relacions no taxonòmiques. Es visualitza la informació de la relació, ja que quan seleccionem un element tenim el punter damunt d'aquest, i a l'esquerra una llista amb les pàgines webs de la relació i a la dreta es visualitza un altre panell informatiu amb la informació del verb que etiqueta la fletxa que representa la relació.



Fig. 4.15: Informació taxonòmica d'una classe

A la figura 4.15 es mostra la informació que rebem de l'aplicació al pannel informatiu central quan passem amb el punter per damunt d'una classe. Es pot observar que si la classe no té valors en algun dels atributs, és visualitzen els valors que se'ls dona per defecte: -1. Amb aquesta mateixa operació també es visualitza a la *statusbar* (part inferior) el nombre de relacions taxonòmiques, nombre d'instàncies i nombre de relacions no taxonòmiques que té la classe.

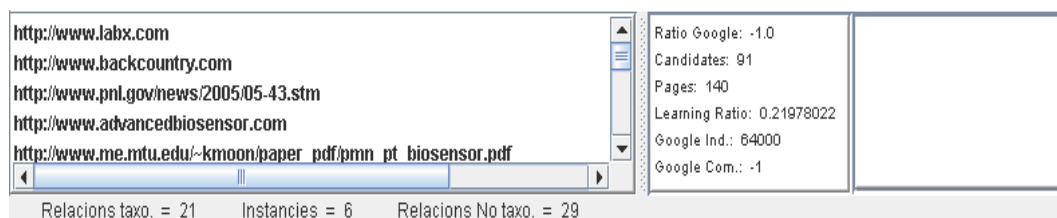


Fig. 4.16: Informació taxonòmica d'una classe.

La figura 4.16 és la mateixa que la figura 4.15 però en aquest cas s'ha seleccionat la classe sobre la que tenim situat el punter. Per tant, tindrem la mateixa informació més les pàgines webs de la classe visualitzades al pannel informatiu esquerra.

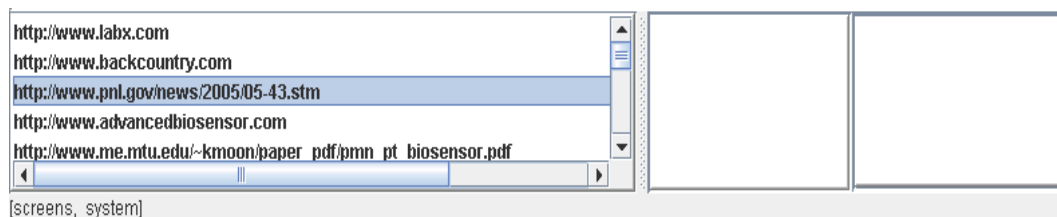


Fig. 4.17: Informació taxonòmica d'una classe.

La figura 4.17 és una imatge per mostrar que quan seleccionem una de les pàgines webs obtingudes en la selecció de la classe, a la *statusbar* es visualitzen les etiquetes, que

descriuen el contingut en relació al domini (concepte) al que pertanyen, associades que tenim a aquella pàgina web en concret.



Fig. 4.18: Informació d'una instància.

A la figura 4.18 es mostra la informació visualitzada quan passem el punter per damunt d'una instància. Veiem com al pannel informatiu central es visualitza la meta-informació relativa a la instància i a la *statusbar* es visualitza el nom complet (*full_name*) de la instància.

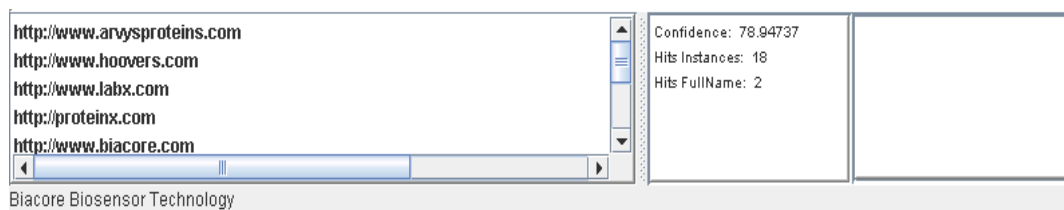


Fig. 4.19: Informació d'una instància.

La figura 4.19 és la mateixa que la figura 4.18 però amb la instància seleccionada, així doncs, com en els casos anterior, això provocarà que es visualitzin les pàgines webs de la instància.

Si premem el botó dret del ratolí quan estem amb el punter damunt d'una classe, apareix un menú *pop-up* amb les opcions d'expandir classe, col·lapsar classes i eliminar només les classes qui siguin masters com es pot veure a la figura 4.20.

Aquest menú només apareix si actuem sobre les classe, en les instàncies no rebrem cap tipus de resposta.

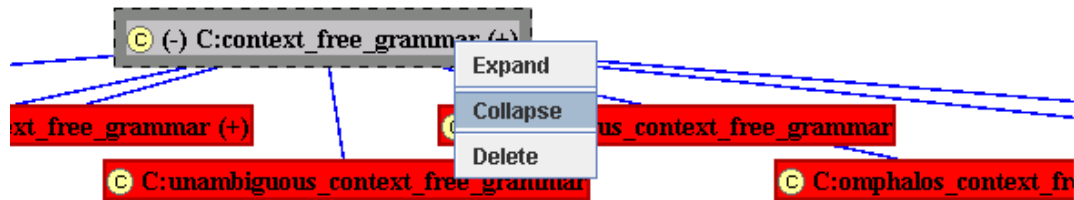


Fig. 4.20: Menú contextual sobre la classe *context_free_grammar*.

Si premem el botó dret del ratolí damunt d'una relació no taxonòmica no apareixerà cap menú sinó que aconseguirem modelar les fletxes creant punts intermedis de la fletxa com ja s'ha vist al punt 3.5 Visualització, concretament a la figura 3.11.

4.7 Prova 7: Zooms.

Comprovació del l'aparença del graf a l'aplicar zooms.

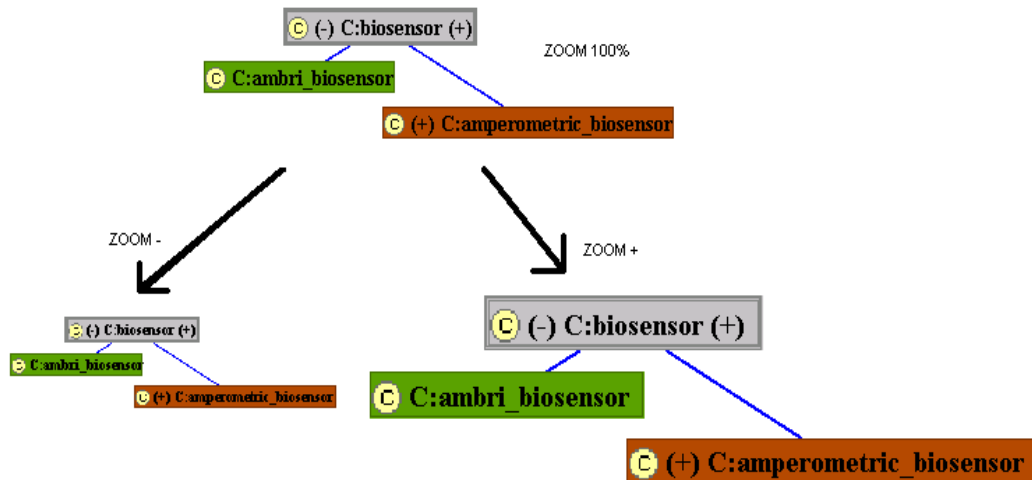


Fig. 4.21: Aplicació de zooms

A la figura 4.21 es mostra tres mides del graf: el primer graf és la mida original, escala 1:1, el graf de l'esquerra és el resultat d'aplicar un zoom reductor i el graf de la dreta és el resultat d'aplicar el zoom ampliador.

Per finalitzar, el joc de proves, avaluarem la visualització del fitxers CLEAN. Com que aquests fitxers no contenen meta-informació, la visualització és més pobre.

No tindrem cap dels aspectes gràfics que depenien de determinats atributs de les classes o instàncies, però el funcionament i la interoperabilitat serà la mateixa.

4.8 Prova 8: Exploració de fitxers CLEAN.

A la figura 4.22 veiem que no tindrem instàncies, només tindrem classes, però aquestes classes no tindran diferents colors, ni marc de diferents gruixos. Per tant, totes les classes seran de color gris clar i totes amb el mínim gruix de marc.

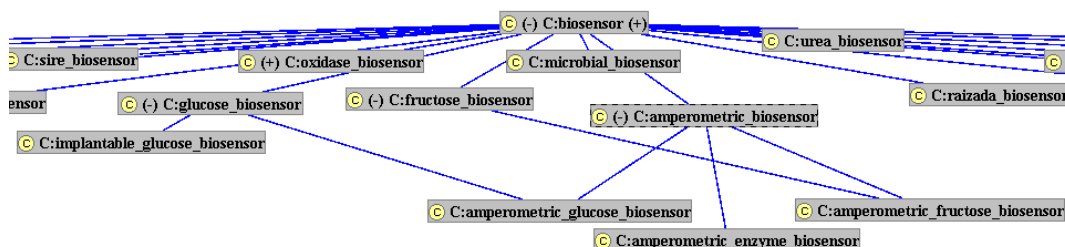


Fig. 4.22: Exploració taxonòmica.

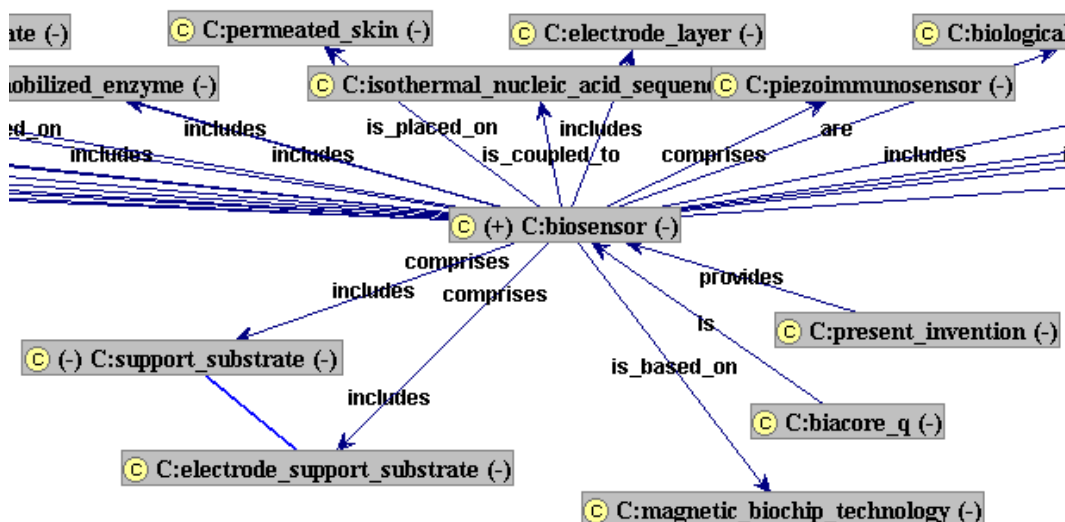


Fig. 4.23: Exploració no taxonòmica.

A la figura 4.23 veiem que les fletxes seran totes de la mateixa mida.

En totes dues figures es pot observar, si comparem amb les proves realitzades anteriorment amb la versió FULL dels fitxers OWL, que les relacions taxonòmiques i no taxonòmiques es mantenen però amb la manca dels aspectes gràfics per la falta de meta-informació.

4.9 Prova 9: Respostes del ratolí en fitxers CLEAN.

Es evident doncs, que si passen el punter per damunt d'una classe o d'una relació no taxonòmica, els atributs que apareixen al pannel informatiu central, sortirà amb els valors d'inicialització (per defecte 0), com es demostra amb les figures 4.24 i 4.25:

	Ratio Google: 0.0 Candidates: 0 Pages: 0 Learning Ratio: 0.0 Google Ind.: 0 Google Com.: 0	
--	---	--

Fig. 4.24. Informació d'una classe.

	Ratio noTaxo: 0.0 Similiary: 0.0 Hits Com.: 0 Hits Ind: 0 Hits Com.: 0	
--	--	--

Fig. 4.25. Informació d'una relació.

Si seleccionem un verb, el pannel informatiu del verb sortirà en blanc, igual que la llista de pàgines webs de les classes i relacions no taxonòmiques.

4.10 Prova 10: Captura del graf.

En la següent figura (fig. 4.26) es pot observar com, si premem el quart botó que tenim a la barra d'eines de l'esquerra: , ens apareix una finestra per guardar el nostre graf donant-li un nom i un lloc on guarda aquest. El format utilitzat serà BMP.

La imatge BMP obtinguda serà tan gran com el graf, és a dir, que tindrem una fotografia de la totalitat del graf actual i no només del tros de graf que es visualitza en aquest moment al visualitzador.

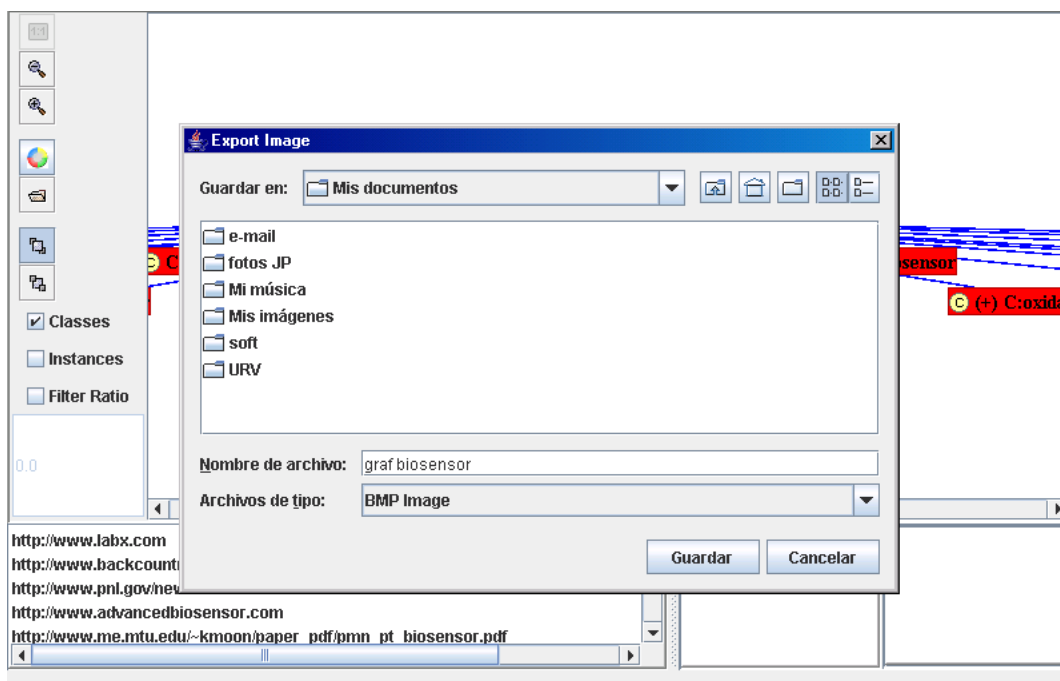


Fig. 4.26: Captura del graf actual

4.11 Prova 11: Integració web.

En aquesta prova veurem com el visualitzador s'ha integrat en l'explorador web de l'ordinador, i com gràcies a això, l'applet (visualitzador) pot obrir les pàgines webs que apareixen a la llista del pannel de planes webs quan seleccionem un element del graf. En la següent figura (fig. 4.27) s'ha seleccionat la classe *biosensor* i s'ha obert la primera pàgina web que apareix a la llista (www.labx.com).

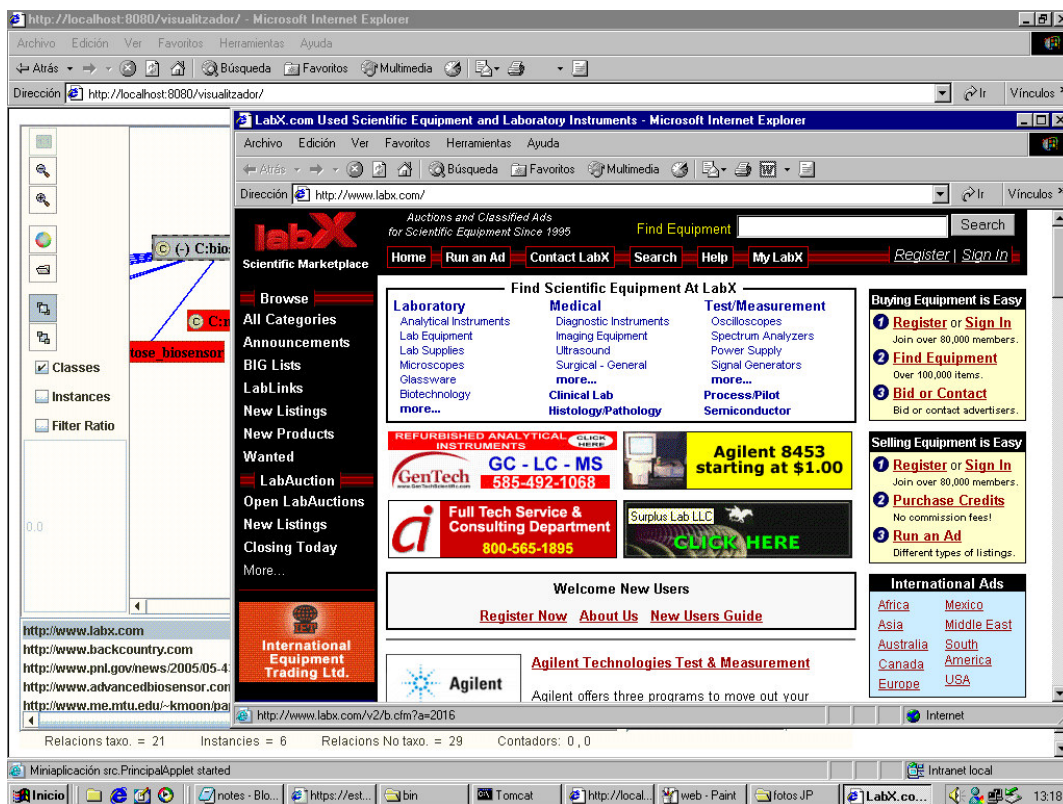


Fig. 4.27: Certificat del visualitzador.

Quan executem el visualitzador, el navegador descarrega l'applet del lloc web on es troba i el primer que veu l'usuari es un certificat que haurà d'acceptar per a que tot vagi bé. Acceptant aquest certificat li estem donat permisos al visualitzador per que pugui superar les restriccions de seguretat dels applets i poder així llegir i escriure sobre la maquina local que executa l'applet. A la fig. 4.28 veiem una imatge del certificat.

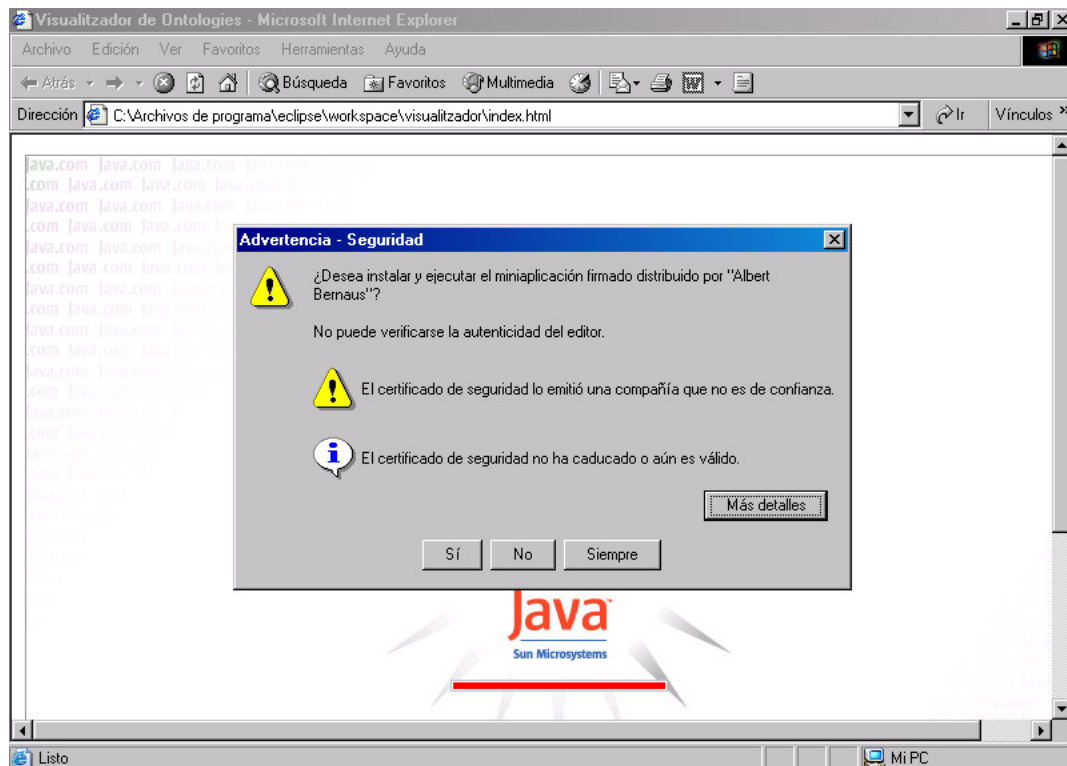


Fig. 4.28: Certificat del visualitzador.

5. CONCLUSIONS

5.1 Resum

Resumint hem creat una eina molt útil, tant a nivell d'usuari com a nivell de desenvolupador, ja que aquesta eina pot ser utilitzada per tot tipus d'usuari per la seva simplicitat.

Per un usuari no expert, el visualitzador, li permet extreure la informació de la Web de forma estructurada i li proporciona un fàcil accés a les pàgines webs existents sobre cada concepte d'un domini, classificades i categoritzades conceptualment.

Des del punt de vista d'un desenvolupador, el visualitzador també és una bona eina perquè proporciona tota un sèrie de meta-informació molt útil que li permet veure la ontologia creada en tot moment.

Per tant, el visualitzador creat, és una eina simple però complexa a la vegada per les prestacions que ofereix.

Un tema relacionat amb aquest projecte, és el camp de la visualització científica, que pretén visualitzar el màxim d'informació, tota de cop mitjançant aspectes gràfics. Això és precisament el que es pretenia mitjançant la meta-informació disponible.

Amb el joc de proves realitzat, es pot dir comprovar que s'han complert tots els objectius proposats a l'inici del projecte.

El visualitzador es capaç de llegir els fitxers OWL amb meta-informació, crear una interfície gràfica rica integrada en una plana web, i oferir flexibilitat i escalabilitat a l'usuari sobre la informació visualitzada en tot moment.

Tot això de forma ràpida i eficaç, ja que ha diferència d'altres eines existents, el temps de carrega i de resposta que hem aconseguit amb el nostre visualitzador és molt bo.

Tenint en compte que es visualitzador s'ha creat per visualitzar els fitxers OWL proporcionats per una nova versió del projecte "Sistema Multi-Agente para la construcción semiautomática de Ontologías a partir de la Web" de Lorena Morales i Francisco Ruiz, creat l'any passat al GruSMA, el treball futur més immediat d'aquest

projecte podria ser, integrar el visualitzador dintre del sistema de construcció d'ontologies, de manera que tindríem l'eina de visualització d'ontologies dins de l'eina encarregada de crear-les, aconseguint així un sistema encara més potent que es complementa.

De fet, aquesta integració es farà amb el projecte actual de Angel Rodríguez, que és aquesta nova versió del projecte de Lorena Morales i Francisco Ruiz.

Un altre treball futur podria ser la millora gràfica del visualitzador incorporant més aspectes visuals o més funcionalitats com ara guardar i recuperar l'estat d'una visualització o implementar un sistema de cerca del graf. I seguint en aquesta línia, com a treball més avançat, es podria incorporar al visualitzador eines d'edició, de manera que es pogués modificar l'ontologia.

Una altra treball futur que es podria fer, és modificar el visualitzador perquè pugui suportar tota la resta d'estructures d'OWL que s'han utilitzat al constructor però que si poden estar presents en un fitxer OWL

La realització d'aquest projecte ha estat molt satisfactòria a nivell personal.

Realitzar el projecte crec que és molt beneficiós per l'alumne perquè és el més semblant al que et pots trobar quan surtis al món laboral, on algú et demana que fàcil alguna cosa sense importar-li com la facis, però que funcioni. Sense donar-te cap eina ni ajuda, només els requisits que ha de complir la feina a realitzar en qüestió i com ha de quedar. A partir d'aquí tu t'has d'espavilar a investigar i trobar eines que t'ajudin a la teva tasca i has de ser tu qui decideixis el disseny més apropiat.

Jo personalment, també he après a planificar-me la feina ja que es tractava de fer una aplicació a llarg termini degut a la complexitat d'aquesta, a diferència de les pràctiques que realitzem al llarg del curs que generalment no solen ser molt llargues.

També es posa de manifest la capacitat de col·laboració i de reunió, en aquest cas amb els directors de projecte, per a la presa de decisions i resolució de problemes que van sortint al llarg del desenvolupament del projecte.

Tot això fa que al final, un cop el projecte està acabat i tot a sortit bé, et sentis orgullós de la feina realitzada al llarg del curs.

6. REFERENCIES

- [1]. OWL. WebSite: <http://www.w3.org/2004/OWL>
API OWL. WebSite: <http://owl.man.ac.uk/api.shtml>
- [2]. Lorena Morales i Francisco Ruiz.
Projecte final de carrera, 2005.
Sistema Multi-Agente para la construcción semiautomática de Ontologías a partir de la Web.
- [3]. Protege. WebSite: <http://www.protege.stanford.edu>
- [4]. Hyperbolic Tree. WebSite: <http://www.inxight.com>
- [5]. The Brain. WebSite: <http://www.thebrain.com>
- [6]. OntoEdit. WebSite: [http:// www.ontoknowledge.org/tools/ontoedit.shtml](http://www.ontoknowledge.org/tools/ontoedit.shtml)
- [7]. Aduna ClusterMap Library 2005.1.
WebSite: <http://http://aduna.biz/index.html>
- [8]. GraphViz. WebSite: <http://www.graphviz.org/>
- [9]. IsaViz. WebSite: <http://www.w3org/2001/11/IsaViz>
- [10]. RDFViz.
WebSite: <http://www.ilrt.bris.ac.uk/discovery/rdf-dev/rudolf/rdfviz>
- [11]. Ontoviz. Plug-in Protégé.
WebSite: <http://http://protege.cim3.net/cgi-bin/wiki.pl?OntoViz>
- [12]. Jambalaya. Plug-in Protégé.
WebSite: <http://www.cs.uvic.ca/~chisel/projects/jambalaya/jambalaya.html>
- [13]. OntoRama. WebSite: <http://www.ontorama.com>
- [14]. TouchGraph. WebSite: <http://www.touchgraph.com>
- [15]. JGraph. WebSite: <http://www.jgraph.com>

- Bruce Eckel.
Piensa en Java, Segona edició.
WebSite: <http://www.programacion.net>
- WebSite: <http://es.wikipedia.org/wiki/Wikipedia>