UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

JDataMotion: unha ferramenta para a visualización dinámica de diagramas de dispersión

Autor:
Pablo Pérez Romaní

Codirectores:
Paulo Félix Lamas
David González Márquez

Grao en Enxeñaría Informática

Xullo 2015

Traballo de Fin de Grao presentado na Escola Técnica Superior de Enxeñaría da Universidade de Santiago de Compostela para a obtención do Grao en Enxeñaría Informática

D. Paulo Félix Lamas, Profesor do Departamento de Electrónica e Computación da Universidade de Santiago de Compostela, e **D. David González Márquez**, bolseiro de FPU do Centro de Investigación en Tecnoloxías da Información da USC,

INFORMAN:

Que a presente memoria, titulada *JDataMotion: unha ferramenta para a visualización dinámica de diagramas de dispersión*, presentada por **D. Pablo Pérez Romaní** para superar os créditos correspondentes ao Traballo de Fin de Grao da titulación de Grao en Enxeñaría Informática, realizouse baixo nosa dirección no CiTIUS da Universidade de Santiago de Compostela.

E para que así conste aos efectos oportunos, expiden o presente informe en Santiago de Compostela, a 10/07/2015:

Os codirectores, O alumno,

Paulo Félix Lamas David González Márquez Pablo Pérez Romaní

Índice xeral

Índice de figuras

Capítulo 1

Introdución

Na actual sociedade da información, onde a cantidade de datos que se manexan aumenta día a día de xeito exponencial, a minería de datos convértese nunha ferramenta fundamental para poder explotalos de maneira eficaz, co fin último de xerar coñecemento a partir dos mesmos.

Para visualizar estes datos unha das técnicas máis utilizadas son os diagramas de dispersión ou scatterplots. Estes permítennos analizar os datos e atopar con facilidade relacións entre as distintas variables, como a correlación entre elas, a distribución dos puntos no plano, a tendencia dos datos recollidos ou outras características que sería complicado extraer a partir dun simple listado, posiblemente desordenado, de vectores de datos. Non obstante, os diagramas de dispersión restrínxennos a unha perspectiva estática do problema. En moitos deses problemas imos encontrar datos cunha compoñente que os sitúa no tempo. Con este proxecto pretendemos dotar a esta representación da súa perspectiva dinámica, para amosar os datos engadindo outro punto de vista que enriqueza a información extraida.

Deséxase desenvolver unha ferramenta para etiquetar cada punto dun diagrama de dispersión cun valor de significado temporal, de tal xeito que este puidese ser empregado como índice nunha visualización dinámica. Este valor numérico podería referenciar dende o momento de captación da tupla que a contén, ata unha ordenación dos datos atendendo á súa prioridade ou relevancia.

1.1. Obxectivos xerais

A motivación principal deste proxecto é o desenvolvemento dunha ferramenta capaz de visualizar a evolución dun conxunto de datos ao longo dunha

magnitude como sería o tempo, ademais de permitir o preprocesado ou manipulación deses datos. Sendo máis específicos, este proxecto busca a realización da análise, deseño e implementación dunha aplicación que consiga:

- Facilitarlle ao usuario o procesado de volumes de datos dun tamaño significativo.
- Posibilitar o traballo con formatos de arquivo CSV ou ARFF.
- Dispor das funcionalidades necesarias para manipular os datos.
- Ser capaz de amosar os datos en forma de diagramas de dispersión, con funcións de reprodución básicas. Tamén se debe posibilitar a configuración desta reprodución por parte do usuario.
- Aplicar filtros nos datos cos que se traballa, de xeito que se poidan eliminar datos fora dun rango, normalizar os seus valores, etc.
- Interaccionar co usuario por medio dunha interface simple e amigable.
- Aplicar nun caso real a ferramenta JDataMotion, para apreciar a súa utilidade.
- Finalizar o desenvolvemento do proxecto antes do día 10 de Xullo de 2015.

1.2. Relación da documentación

Esta memoria plasma o proceso de desenvolvemento do proxecto JDataMotion, que persegue os obxectivos citados no apartado anterior.

Os distintos capítulos repártense do modo que segue:

Capítulo 1. Introdución:
composta por obxectivos xerais, relación da documentación que conforma a memoria, descrición do sistema (métodos, técnicas ou arquitecturas utilizadas e xustificación da súa elección).

Capítulo 2. Planificación e presupostos:
inclúe a estimación dos recursos necesarios para desenvolver este proxecto, xunto co custo (presuposto) e planificación temporal do mesmo, así como a súa división en fases e tarefas.

Capítulo 3. Especificación de requisitos:
inclúe a especificación do sistema, xunto coa información que este debe almacenar e as interfaces con outros sistemas, sexan hardware ou software, e outros requisitos (rendemento, seguridade, etc).

Capítulo 4. Deseño:
rexistra como se realiza o sistema, a división deste en diferentes compoñentes e a comunicación entre eles. Así mesmo, neste apartado determínase o equipamento hardware e software necesario.

Capítulo 5. Exemplos:
Avaliación do grao de cumprimento dos requisitos e tests os verifican.

Capítulo 6. Conclusións e posibles ampliacións.

Apéndice A. Manuais técnicos:
incluirase toda a información precisa para aquelas persoas que se vaian a encargar do desenvolvemento e/ou modificación do sistema.

Apéndice B. Manuais de usuario:
incluirán toda a información precisa para aquelas persoas que utilicen o sistema: instalación, utilización, configuración, mensaxes de erro, etc.

Apéndice C. Licenza.

Bibliografía

Capítulo 2

Xestión do proxecto

Neste capítulo comentaremos distintos aspectos relacionados coa planificación de como se vai xestionar este proxecto. Falaremos, por exemplo, da xestión de riscos que conleva o desenvolvemento do software, así coma os métodos de continxencia, prevención ou minimización que seguiremos en caso da incidencia dos mesmos. Cos riscos expostos, abordaremos a metodoloxía de desenvolvemento máis axeitada para o proxecto, de acordo tamén cos obxectivos anteriormente plasmados. Seguiremos coa planificación temporal do proxecto e finalizaremos coa estimación de custo e prazos, así como a xestión da configuración.

2.1. Xestión de riscos

Na fase de planificación dun proxecto hai que sopesar os distintos riscos aos que estará exposto o seu desenvolvemento, cuantificalos e deseñar estratexias para a súa aparición. Algúns dos riscos nun Traballo de Fin de Grao poden ter graves consecuencias na liña base do proxecto debido á inexperiencia do seu autor, polo que fronte á falta de experiencia hai que esforzarse en mellorar a planificación.

Na análise de riscos valoraremos a probabilidade de aparición e a súa gravidade, para a continuación deseñar unha medida de continxencia, prevención ou minimización. A escala de valoración da probabilidade e da gravidade vai ser:

- Moi baixa
- Baixa
- Media
- Alta
- Moi alta

Os riscos considerados son os seguintes:

- **Risco 01**

  Nome:
  Cambios no alcance durante o desenvolvemento

  Descrición:
  A lista inicial de requisitos funcionais que se captará nas primeiras reunións cos titores vai sufrir modificacións, incluso co proxecto en etapas avanzadas de desenvolvemento. A súa probabilidade duplícase pola existencia de dous clientes no Traballo de Fin de Grao.

  Probabilidade:
  Moi alta

  Gravidade:
  Alta

  Medidas de minimización:
  Botaremos man da folgura temporal do proxecto (marxe de tempo dispoñible para eventualidades). Os cambios razoaranse cos titores, presentando a lista de requisitos actual e valorando a parte da folgura que consumirían ditos cambios. Tamén se tratará de ter reunións de avaliación cos titores cunha alta frecuencia, para así detectar o antes posible calquera cambio nos requisitos, se ben pode acontecer que se propoñan cambios sobre as primeiras etapas cando o proxecto se atopa en etapas avanzadas.

- **Risco 02**

  Nome:
  Imprecisión á hora de fixar entregables

  Descrición:
  A inexperiencia do alumno manifestarase xa nas primeiras entregas programadas. Ao non ter traballado previamente en proxectos desta índole, resultará complicado estimar os prazos de entrega nas primeiras fases do proxecto, tanto por exceso como por defecto.

  Probabilidade:
  Alta

  Gravidade:
  Media

  Medidas de minimización:
  Intentaremos especificar entregas dun contido menor e máis frecuentes,

sobre todo nas primeiras fases, para que sexa máis doado comezar a estimar correctamente os prazos de entrega.

- **Risco 03**

  Nome:
  Imposibilidade de reunirse cun dos titores

  Descrición:
  Un dos titores non pode acudir a algunha reunión proposta, nin estará nos seguintes 5 días.

  Probabilidade:
  Alta

  Gravidade:
  Baixa

  Medidas de minimización:
  Desenvolverase a reunión co titor dispoñible, sendo mester informar das conclusións sacadas ao outro titor por medio do correo electrónico en canto remate a reunión.

- **Risco 04**

  Nome:
  Descoñecemento ou inexperiencia coas solucións

  Descrición:
  O alumno non coñece as posibilidades que teñen as ferramentas das que dispón (librarías, módulos, solucións, etc.).

  Probabilidade:
  Alta

  Gravidade:
  Media

  Medidas de prevención:
  Adicarase un tempo prudencial, nas primeiras fases, a revisar as APIs e a documentación en xeral das librerías, proxectos de terceiros e demais ferramentas que se van empregar, para ser conscientes de como poden solucionar as nosas necesidades.

- **Risco 05**

  Nome:
  Imposibilidade de finalizar o proxecto en tempo

Descrición:
A folgura está esgotada, e o cumprimento do prazo de entrega vese ameazado.

Probabilidade:
Media

Gravidade:
Moi alta

Medidas de prevención:
Evitaremos na medida do posible recorrer á folgura, e trataremos de seguir a planificación do xeito máis estrito que podamos.

Medidas de minimización:
Poremos en coñecemento aos titores do estado do proxecto e dos seus prazos, para discutir a modificación ou eliminación dalgúns ítems da especificación.

- **Risco 05**

  Nome:
  Limitación das librarías gráficas

  Descrición:
  As APIs e librarías gráficas non dan solución todas as nosas necesidades

  Probabilidade:
  Media

  Gravidade:
  Alta

  Medidas de prevención:
  Estudiar a especificación de cada solución e as funcionalidades que ofrece.

  Medidas de minimización:
  Implementarase de xeito manual a solución que se necesita, podendo partir ou botar man do código da librería.

2.2. Metodoloxía de desenvolvemento

A elección da metodoloxía de traballo é un paso importante na planificación de calquera proxecto, xa que a posteriori influirá en varios aspectos deste: a

xestión dos seus riscos, a súa tolerancia a cambios externos, a confianza na súa validez, etc. Hai dous enfoques fundamentais: as metodoloxías estritas e as metodoloxías áxiles. As primeiras esixen unha planificación estrita, practicamente inmutable e necesariamente realista de todo o plan de traballo, e son boas cando o conxunto de requisitos é fixo e moi concreto. As segundas, pola contra, son flexibles e adáptanse ben a cada situación, pois nelas asúmese que se van producir variacións nos requisitos.

Sopesando as circunstancias nas que se desenvolve un Traballo de Fin de Grao, onde a experiencia do alumno é practicamente nula no que respecta á xestión de proxectos, semella que deberíamos adoptar unha metodoloxía de traballo que se adapte ás necesidades de cambios que vaian xurdindo, e que consiga en cada entrega recibir certa retroalimentación por parte dos titores, de forma que tras cada iteración podamos ter a seguridade da correspondencia entre o proxecto e o modelo mental de quen o especificou. É dicir, necesitamos unha metodoloxía áxil.

Dentro do compendio de metodoloxías áxiles existentes, decantarémonos pola metodoloxía Scrum [7], pois enfoca todas as súas avaliacións sobre entregas parciais, pero funcionais, para facilitarlle ao receptor do proxecto a valoración do mesmo. Necesítase, polo tanto, unha gran implicación do cliente no proxecto, algo que se pode conseguir dada a dualidade da titoría (é máis doado que haxa un titor dispoñible para realizar a reunión). Por outra banda, os requisitos que constitúen as distintas entregas deben estar priorizados para que o proxecto poida avanzar cun carácter incremental, e ditas entregas deben resultar usables para o cliente.

Scrum define unha serie de ferramentas e de regras, idóneas para levar a cabo o desenvolvemento de proxectos que buscan unha metodoloxía áxil. Os preceptos básicos desta metodoloxía son:

- Adoptar unha estratexia de desenvolvemento incremental, no canto da planificación e execución completa do produto (é dicir, no canto dunha metodoloxía estrita).
- Basear a calidade do resultado máis no coñecemento das persoas que o especificaron ca na calidade dos procesos empregados.
- Solapamento das fases de desenvolvemento (análise, deseño, implementación e probas) no canto da sucesión secuencial que nos ofrecen metodoloxías como a fervenza.

A metodoloxía Scrum comeza coa adquisición de requisitos en reunión co cliente, da cal se extrae un Backlog, é dicir, unha lista ordenada por prioridade de requisitos funcionais (RFs). Ademais, Scrum define o sprint como unidade elemental de tempo de traballo. Un sprint dura entre 1 e 4 semanas, aínda que nós

trataremos de manter a súa duración en 2 ou incluso 1 semanas para maximizar a supervisión e xestionar ben os riscos, sobre todo nas etapas iniciais.

Ao término de cada reunión co cliente, revísase o Backlog e incorpórase un certo número de RFs a un novo sprint, o cal dará comezo en canto remate a reunión. Para o remate dese novo sprint (dentro dunha semana no noso caso) terase programada a seguinte reunión, na que se valorará o sprint finalizado e se accederá ao Backlog para acordar o seguinte sprint, e así sucesivamente. A valoración do sprint en cada reunión realízase presentando a lista de RFs de dito sprint, e demostrando ante o receptor do proxecto que cada un dos ítems ou tarefas do sprint funciona correctamente.

Para regular o desenvolvemento desa metodoloxía pódese botar man de diversas ferramentas, de entre as cales nós escollemos Acunote [8] para o noso proxecto. Acunote é unha aplicación web especialmente deseñada para a xestión da metodoloxía Scrum. Ten varios plans de prezos, pero nós empregaremos o gratuíto porque as nosas necesidades restrínxense a un equipo de persoal pequeno (o alumno e os dous titores). Entre as prestacións desta ferramenta, sacarémoslle maior proveito ás seguintes:

Wiki:
Empregarémola para engadir contido visible ao resto de membros do grupo.

Lista de sprints:
Amosa os sprints en 3 grupos: sprints pasados, sprints presentes e sprints futuros. Ao abrir un sprint visualízanse os ítems ou tarefas (requisitos funcionais no noso caso) que o compoñen. Accederemos a este apartado na maioría dos casos para crear novos sprints.

Sprint actual:
Visualiza os RFs do sprint actual. A medida que se vaian completando requisitos funcionais, accederase a esta lapela para cambiar o estado do requisito en cuestión. Os estados posibles son:

- Non comezado (por defecto)
- En progreso
- Reaberto
- Bloqueado
- Completado
- Verificado
- Duplicado
- Non se vai realizar

Backlog:
Contén todos os ítems (requisitos funcionais) pendentes de ser asignados a un sprint. Este lista cumprimentarase ao principio, cos requisitos funcionais captados e ordenados por prioridade, e logo accederase a ela á hora de asignar RFs aos novos sprints. Na extracción de RFs débese respectar a orde dos mesmos dentro da lista, collendo sempre un número de ítems determinado da parte superior. Estes ítems desaparecerán do Backlog en canto sexan asignados.

Tarefas:
Mostra todos os ítems especificados, independentemente de que fosen asignados a un sprint ou non.

2.3. Planificación temporal

A metodoloxía Scrum caracterízase como ben dixemos polo solapamento das fases que nun modelo en fervenza estarían ben separadas. As primeiras semanas de traballo estarán adicadas á análise para a captación inicial de requisitos, pero nas sucesivas iteracións ou sprints poderán realizarse en paralelo análise, deseño, implementación e probas. Esta é unha das licencias que outorga o emprego das metodoloxías áxiles. De todos xeitos, aínda dentro da variabilidade destas metodoloxías, podemos dividir a vida do proxecto en unha serie de fases fundamentais:

Inicio:
Constitúe o primeiro sprint (Sprint 00) da planificación, e durará dúas semanas. Nesta fase programaranse reunións cos titores para realizar a captación de requisitos funcionais (análise de requisitos), e ordenaranse estes por prioridade, dando lugar ao Backlog. Tamén se definirá a especificación de cada requisito e se deseñarán as probas que os verifiquen.

Desenvolvemento:
Abrangue dende o Sprint 01 ata o Sprint 11, ambos inclusive (20 semanas en total). Nesta fase elaborarase o produto de acordo cos requisitos.

Documentación:
Abrangue 5 semanas de traballo. Nesta fase recompilarase toda a documentación xerada nas fases anteriores, e confeccionarase a memoria e máis a presentación, que constituirán os entregables do Traballo de Fin de Grao.

En total, o proxecto traballarase durante un período de 27 semanas (189 días, algo máis de 6 meses), co cal, para realizar as 401,25 horas de traballo necesarias teremos que levar un ritmo de traballo aproximado de 15,28 horas semanais

(unha media de 2 horas e cuarto diarias). Non nos convén asumir un ritmo de traballo maior, pois durante ese período de tempo o alumno deberá repartir a súa axenda entre este proxecto, o resto de materias, as prácticas en empresa, etc.

2.4. Xestión da configuración

Todo proxecto ten elementos de interese para incluír na xestión da configuración. Estes elementos caracterízanse porque son candidatos a sufrir cambios que poden ameazar o correcto desenvolvemento do proxecto. A xestión da configuración trata de manter a integridade do proxecto perante a estes cambios. O noso deber é identificar que obxectos do proxecto (sexan entregables ou resultados parciais do mesmo) merecen a súa inclusión na xestión da configuración, e por outra parte, temos que especificar que ferramentas empregaremos para dar soporte a esta característica.

Para este caso consideraremos ao código fonte do proxecto (e máis das súas probas) e á documentación como elementos de configuración. O código fonte é o sustento do noso proxecto, e os cambios no seu contido veranse directamente reflexados no produto a entregar, polo que é mester incluír este elemento na xestión da configuración. Tamén incluiremos o código fonte das probas porque debemos respectar a integridade entre estas e o propio proxecto. Por outra parte, a documentación sufrirá cambios de xeito paralelo ao código, e evolucionará da man deste ao longo da vida do proxecto (rexistrará a súa especificación de requisitos, o seu deseño, etc.), así que tamén debe ser un elemento a considerar en aras de preservar a integridade do proxecto. En resumo, faremos seguimento de cambios dos tres directorios ('src', 'test' e 'doc'), e para iso botaremos man do software GitHub [9].

Github é unha plataforma para darlle aloxamento a distintos tipos de proxectos, por medio do sistema de control de versións Git. Para aloxar o noso proxecto crearemos un repositorio local e outro remoto, chamando a ambos 'JDataMotion', e outorgándolle ao remoto permisos de lectura e escritura para o alumno e permisos de lectura (ou de escritura tamén, opcionalmente) para os titores. Deste xeito estes poderán descargar a última versión do proxecto en calquera momento, mentres que o alumno poderá ir subindo as modificacións cos cambios implementados cada certo tempo.

O proxecto conterá moitos máis elementos, pero non podemos consideralos a todos aptos para a xestión de configuración por diversos motivos: as librarías empregadas non cambian (e no caso de querer actualizar algunha, asúmese que non deben xurdir problemas de integridade grazas á compatibilidade entre versións), os arquivos de configuración persoal non deben ser almacenados no repositorio, e

os ficheiros de código obxecto e de distribución dependen directamente do código fonte (que xa é un elemento de configuración), pois xéranse como resultado da súa compilación.

2.5. Análise de custos

A estimación dos custos de desenvolvemento do proxecto amósase no Cadro 2.1. Para ela, consideramos a adquisición dun novo equipo informático. As horas de traballo neste caso non van ter un valor económico asociado, como consecuencia de que este proxecto pertenza a un Traballo de Fin de Grao, pois as horas do traballo do alumno correspóndense coas que este debe cumprimentar para a obtención do título. Para o consumo eléctrico tivemos en conta o prezo do kWh en España [5] e fixemos unha estimación [6] do consumo eléctrico dun equipo informático, que a plena potencia pode traballar a 120 W. Considerando que o desenvolvemento do traballo durará 401,25 horas, necesitaremos 120 W * 401,25 h = 48150 Wh = 48,15 kWh. Ademais, sabemos que o salario medio anual dun desenvolvedor de software en España é de 26740 /ano. Se un ano ten 249 días laborables, o custo por hora deste traballador sería de (26740 €/ano) / (249 días laborables/ano) / (8 horas/día laboral) = 13,4237 €/hora. Os custos inclúen o IVE, pero trataremos de diferencialos na seguinte táboa.

Activo	**Cantidade**	**C.U. sen IVE**	**IVE**	**Custo total**
Ordenador portátil	1	570,00 €	21 %	689,70 €
Horas de traballo	401,25 horas	11,094 €/hora	21 %	5386,26 €
Consumo eléctrico	48,15 kWh	0,0663 €/kWh	21 %	3,86 €
			Total	6079,82 €

Cadro 2.1: Custos

A anterior estimación é válida só en caso de que o produto a desenvolver non teña unha finalidade comercial. No caso de que este se pretenda comercializar, certas ferramentas que imos utilizar poderían esixir o pago dunha licencia específica.

Capítulo 3

Análise

Na fase de análise do proxecto intentaremos discernir cales van ser as metas ou obxectivos do mesmo. Ao seu remate deberemos ter claro cales son os obxectivos que debe alcanzar o proxecto, así como ter definidos os pasos ou tarefas para logralos. Tras isto, nesta etapa tamén incorporaremos un estudo e valoración das tecnoloxías das que botaremos man.

3.1. Análise de requisitos

A extracción dos requisitos dun proxecto é unha fase fundamental na realización de calquera proxecto, pois inflúe non só nas propias tarefas a desenvolver para a súa implementación, se non tamén na valoración do produto final e da súa calidade. O proceso desta etapa pódese revisar na norma IEEE-STD-830-1998 [12]. A obtención de requisitos adóitase facer durante ou tras unha reunión cos clientes.

Durante a reunión cos clientes foron xurdindo requisitos ou condicións necesarias para o produto. Estes requisitos foron rexistrados para posteriormente seren ordenados e clasificados segundo certos criterios que se amosan a continuación.

Os casos de uso empréganse para modelar e representar cómo se vai realizar a interacción entre o sistema e os usuarios del, tamén coñecidos como actores. Os casos de uso constitúen as posibilidades das que dispón cada actor. Esta análise resulta especialmente útil en entornas orientadas a usuarios con distinta prioridade (un administrador, un usuario invitado, un usuario rexistrado, un usuario prémium, etc.) nas que cada un deses actores ten acceso a uns casos de uso específicos (por exemplo, moitas aplicacións web). Por tanto, a riqueza dos diagramas de casos de uso radica na variedade de tipos de usuario (actores). A nosa

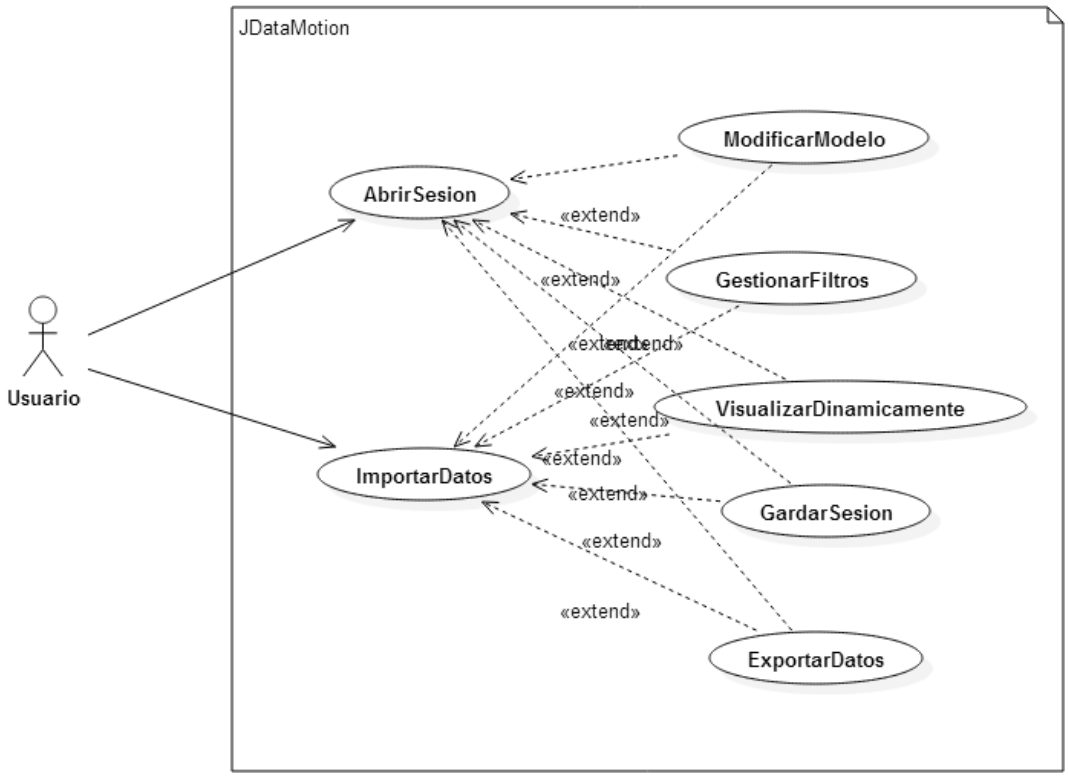


Figura 3.1: Diagrama de casos de uso

aplicación non necesita facer distinción algunha entre os tipos de usuario que poden facer uso dela. Todos van dispor das mesmas funcionalidades. De todas formas, o diagrama de casos de uso pódese apreciar na figura 3.1

Un escenario de caso de uso podería comezar cun traballador do ámbito médico-sanitario que dispón nun csv (exportado por outra aplicación, por exemplo) de certas medicións relacionadas cun paciente seu ao longo do seu seguimento. O traballador (usuario do sistema) podería importar o arquivo dentro do JDataMotion e obter unha táboa editable cos datos contidos, aplicar filtros para eliminar os datos atípicos das medicións do paciente e normalizar algunhas variables. Cos datos xa preprocesados, poderá reproducilos para ver o seu comportamento ao longo do tempo e a evolución do paciente. Finalmente, gardará os datos filtrados baixo un novo csv.

3.1.1. Requisitos funcionais

RF01

Título
Importar arquivos con datos para o experimento

Descrición
A aplicación debe permitir cargar do sistema de arquivos un ficheiro que conteña unha secuencia de datos (nun formato axeitado segundo o RNF01) para ser utilizados no experimento.

Importancia
Esencial

RF02

Título
Exportar datos

Descrición
A aplicación debe permitir almacenar nun arquivo o conxunto de datos do experimento actual (tendo en conta filtrados, modificacións, datos engadidos ou eliminados...). Os arquivos de saída deberán respectar o RNF01 en canto a formato de almacenamento.

Importancia
Esencial

RF03

Título
Gardar sesión

Descrición
A aplicación debe permitir gardar en disco a sesión (ou experimento) actual tal e como está no momento de executar esta acción.

Importancia
Esencial

RF04

Título
Abrir sesión

Descrición
A aplicación debe permitir restaurar unha sesión (ou experimento) gardada anteriormente, de xeito que se atope exactamente igual ca no momento en que se gardou.

Importancia
Esencial

RF05

Título
Representar os datos en forma de táboa

Descrición
A aplicación debe ser capaz de amosar os datos segundo unha táboa na que figuren cabeceiras, tipos, valores, etc.

Importancia
Esencial

RF06

Título
Insertar datos no experimento actual

Descrición
A aplicación debe permitir a inserción dinámica de datos no experimento actual.

Importancia
Esencial

RF07

Título
Modificar datos no experimento actual

Descrición
A aplicación debe permitir a modificación dinámica de datos no experimento actual.

Importancia
Esencial

RF08

Título
Eliminar datos no experimento actual

Descrición
A aplicación debe permitir a eliminación dinámica de datos no experimento actual.

Importancia
Esencial

RF09

Título
Asignar tipos aos atributos dun arquivo importado

Descrición
A aplicación debe permitir especificar os tipos de atributos presentes no arquivo importado. Por exemplo, os datos cuantitativos poderían ser enteiros ou reais, mentres que os cualitativos serían algo distinto (mesmamente strings).

Importancia
Esencial

RF10

Título
Sinalar identificación temporal

Descrición
A aplicación debe permitir sinalar unha columna que exprese o orde ou a temporalidade dunha tupla, ou ben definir esta columna manualmente.

Importancia
Esencial

RF11

Título
Representar os datos graficamente mediante diagrama de dispersión

Descrición
A aplicación debe ser capaz de representar graficamente (mediante diagrama de dispersión) o conxunto de parámetros de entrada. Concretamente, débense poder representar ata 3 parámetros por cada diagrama de dispersión (ordeadas, abscisas e cor e forma dos puntos). Todos os diagramas de dispersión estarán englobados dentro do "menú de visualización", que cumprirá co RNF04.

Importancia
Esencial

RF12

Título
Engadir diagramas de dispersión ao menú de visualización

Descrición
A aplicación debe permitir engadir dinámicamente novos diagramas de dispersión dentro do menú de visualización.

Importancia
Esencial

RF13

Título
Eliminar un diagrama de dispersión do menú de visualización

Descrición
A aplicación debe permitir eliminar un diagrama de dispersión do menú de visualización.

Importancia
Esencial

RF14

Título
Configurar diagramas de dispersión do menú de visualización

Descrición
A aplicación debe permitir especificar para os diagramas de dispersión do menú de visualización a súa configuración, respecto a que parámetros se representarán en cada un dos eixos ou si as cores e formas dos puntos se desexan usar para representar algún atributo nominal.

Importancia
Esencial

RF15

Título
Detallar punto seleccionado dentro do diagrama de dispersión

Descrición
Cada punto dos diagramas de dispersión pode ser seleccionado para ver nun apartado os seus detalles (todos os seus atributos).

Importancia
Esencial

RF16

Título
Resaltar punto en diagramas de dispersión

Descrición
Cada punto seleccionalo dentro dun diagrama de dispersión resaltarase tanto nel coma en todos os demais diagramas de dispersión (que plasmarán outras proxeccións do mesmo punto).

Importancia
Esencial

RF17

Título
Desprazar a ventá de visualización por arrastre de cada diagrama de dispersión

Descrición
Para cada diagrama de dispersión poderemos usar unha ferramenta “man” para desprazar a ventá polo diagrama de dispersión.

Importancia
Esencial

RF18

Título
Escalar a ventá de visualización de cada diagrama de dispersión

Descrición
Para cada diagrama de dispersión poderemos usar unha ferramenta de escalado da ventá para facer zoom no diagrama de dispersión.

Importancia
Esencial

RF19

Título
Escalar e reposicionar dinamicamente

Descrición
Para cada diagrama de dispersión permitirase que a ventá de visualización que o enfoca se adapte dinamicamente ao conxunto de datos representados (movéndose, afastándose e aproximándose para englobar todos os datos).

Importancia
Esencial

RF20

Título
Reproducir a secuencia de datos

Descrición
A aplicación debe de permitir que a visualización dos diagramas de dispersión poida basearse na variable temporal (ou de orde) para reproducir a secuencia de datos, amosando os datos de cada diagrama de dispersión baixo unha secuencia de vídeo. Nesta secuencia engadiríase á visualización en cada instante a tupla de atributos asociada a esa marca temporal.

Importancia
Esencial

RF21

Título
Representar estela

Descrición
A aplicación debe de permitir que cada novo punto pintado se ligue ao último representado no diagrama de dispersión por medio dunha liña recta.

Importancia
Esencial

RF22

Título
Difuminar estela ao longo da reprodución

Descrición
A aplicación debe permitir difuminar as estelas xa representadas a través do avance temporal.

Importancia
Esencial

RF23

Título
Configurar a reprodución da secuencia de datos

Descrición
A aplicación debe de permitir que a visualización dos diagramas de dispersión sexa configurable en canto a tempo transcorrido entre marcas temporais. Para a reprodución usando marcas temporais ponderadas, este tempo representará a separación entre as dúas marcas temporais mais próximas (tempo mínimo). Ademáis débese poder especificar o número de marcas temporais que durará o difuminado dos puntos que se ploteen, de xeito que durante ese intervalo cada punto se vaia difuminando ata desaparecer. Pode ser igual a 0 para que os puntos non se difuminen.

Importancia
Esencial

RF24

Título
Pausar a reprodución

Descrición
A aplicación debe permitir parar a reprodución na marca de tempo na que se atope ao executar esta acción, mantendo as visualizacións para ese momento.

Importancia
Esencial

RF25

Título
Ir a un determinado instante dentro do intervalo temporal da reprodución

Descrición
A aplicación debe permitir situarse directamente sobre un instante de tempo, mantendo a reprodución pausada sobre esa marca temporal, e visualizando os diagramas de dispersión tal e como deben estar nese momento.

Importancia
Esencial

RF26

Título
Insertar filtros para os datos do experimento

Descrición
A aplicación debe permitir engadir unha serie de filtros que se aplicarán de xeito secuencial sobre a secuencia de datos coa que se esté a traballar. Chamarémoslle "secuencia de filtros" a esta secuencia.

Importancia
Esencial

RF27

Título
Eliminar un filtro para os datos do experimento

Descrición
A aplicación debe permitir eliminar un determinado filtro dentro da secuencia de filtros.

Importancia
Esencial

RF28

Título
Configurar filtros para os datos do experimento

Descrición
A aplicación debe permitir seleccionar un determinado filtro dentro da secuencia de filtros para modificar a regla de filtrado implícita.

Importancia
Esencial

RF29

Título
Gardar unha secuencia de filtros do experimento

Descrición
A aplicación debe permitir gardar unha secuencia de filtros, non necesariamente correlativos, dentro dos que se estean aplicando sobre o experimento. Esta secuencia pode comprender tanto un só filtro como a secuencia de filtros enteira.

Importancia
Esencial

RF30

Título
Cargar unha secuencia de filtros para o experimento

Descrición
A aplicación debe permitir cargar do sistema de arquivos unha secuencia de filtros que se engadirá á cabeza da secuencia de filtros (a cal pode estar baleira). Esta secuencia tamén pode estar composta por un só filtro.

Importancia
Esencial

RF31

Título
Mover os filtros dentro da secuencia de filtros

Descrición
A aplicación debe permitir desprazar un filtro dentro da secuencia de filtros do experimento, de xeito que o orde de aplicación dos filtros varíe. O desprazamento realizarase inserindo o filtro en cuestión nunha nova posición.

Importancia
Esencial

RF32

Título
Configurar o menú de visualización

Descrición
A aplicación debe permitir cambiar os parámetros de visualización dos diagramas de dispersión que compoñen o menú de visualización, por exemplo, a cor das etiquetas e lendas, do fondo, dos eixos... ou a fonte, tamaño de letra...

Importancia
Optativa

3.1.2. Requisitos de calidade

RC01

Título
Latencia mínima para o procesamento

Descrición
A aplicación debe responder nun tempo razoable ás operacións executadas polo usuario, e intentar que esa latencia escale de xeito controlado ao aumentar a talla dos parámetros.

Importancia
Esencial

3.1.3. Requisitos de deseño

RD01

Título
Modularidade no deseño dos filtros

Descrición
A aplicación debe facilitar unha interface para a inclusión e uso de filtros personalizados por parte de calquera desenvolvedor de software que a implemente dentro do proxecto.

Importancia
Esencial

3.1.4. Requisitos non funcionais

RNF01

Título
Formatos de arquivo admitidos ao importar e exportar arquivos

Descrición
A aplicación debe estar preparada para importar e exportar arquivos en distintos formatos, como son o CSV e ARFF.

Importancia
Esencial

RNF02

Título
Relación programa-sesión

Descrición
Cada instancia do programa debe traballar cunha única sesión (experimento).

Importancia
Esencial

RNF03

Título
Implementación en Java

Descrición
O software tense que desenvolver na linguaxe de programación Java.

Importancia
Esencial

RNF04

Título
Representación matricial dos diagramas de dispersión

Descrición
Os diagramas de dispersión represéntanse de xeito matricial, facendo que cada parámetro dentro dun eixo sexa enfrontado a cada un dos demais do outro eixo, e en cada punto desa dupla se sitúe o diagrama de dispersión que compara ambos parámetros. Deste xeito, os diagramas de dispersión non son acumulables: se temos un que representa X (abscisas) fronte a Y (ordenadas), non podemos engadir outro que represente X (abscisas) fronte a Y (ordenadas), pois ocuparían ambos a mesma cela dentro da matriz de diagramas de dispersión.

Importancia
Esencial

RNF05

Título
Entrega dentro de prazo

Descrición
Débese entregar unha versión funcional e documentada antes do día 10 de Xullo de 2015, ás 14:00 horas, pois é o momento no que remata o prazo de entrega.

Importancia
Esencial

3.1.5. RFs dos sprints

Imos a detallar a asignación de requisitos funcionais (RFs) aos distintos sprints ao longo da fase de Desenvolvemento, asignando uns prazos aproximados de traballo sobre uns conxunto de RFs relacionados entre si.

Sprint 01

Nome:
Interacción co sistema de ficheiros

Fase:
Desenvolvemento

Comezo:
17/02/2014

Finalización:
24/02/2014

RFs a implementar:
RF01, RF02, RF03, RF04

Sprint 02

Nome:
Manipulación de datos

Fase:
Desenvolvemento

Comezo:
24/02/2014

Finalización:
10/03/2014

RFs a implementar:
RF05, RF06, RF07, RF08

Sprint 03

Nome:
Preprocesado

Fase:
Desenvolvemento

Comezo:
10/03/2014

Finalización:
24/03/2014

RFs a implementar:
RF09, RF10

Sprint 04

Nome:
Visualización dos datos

Fase:
Desenvolvemento

Comezo:
24/03/2014

Finalización:
14/04/2014

RFs a implementar:
RF11, RF12, RF13, RF14

Sprint 05

Nome:
Ferramentas de visualización

Fase:
Desenvolvemento

Comezo:
14/04/2014

Finalización:
28/04/2014

RFs a implementar:
RF15, RF16, RF17, RF18, RF19

Sprint 06

Nome:
Reprodución

Fase:
Desenvolvemento

Comezo:
28/04/2014

Finalización:
05/05/2014

RFs a implementar:
RF20

Sprint 07

Nome:
Configuración da reprodución

Fase:
Desenvolvemento

Comezo:
05/05/2014

Finalización:
19/05/2014

RFs a implementar:
RF21, RF22, RF23

Sprint 08

Nome:
Funcións de reprodución

Fase:
Desenvolvemento

Comezo:
19/05/2014

Finalización:
02/06/2014

RFs a implementar:
RF24, RF25

Sprint 09

Nome:
Filtros

Fase:
Desenvolvemento

Comezo:
02/06/2014

Finalización:
16/06/2014

RFs a implementar:
RF26, RF27, RF28

Sprint 10

Nome:
Xestionar filtros

Fase:
Desenvolvemento

Comezo:
16/06/2014

Finalización:
30/06/2014

RFs a implementar:
RF29, RF30, RF31

Sprint 11

Nome:
Outras funcións de visualización

Fase:
Desenvolvemento

Comezo:
30/06/2014

Finalización:
07/07/2014

RFs a implementar:
RF32

A partir da planificación temporal de grupos de RFs en común podemos estimar con certa confianza a duración total do proxecto, polo menos na súa fase de desenvolvemento. Faltarían dúas fases máis por considerar:

- Fase de inicio, previa á execución dos sprints, duraría un prazo de 2 semanas e constaría das seguintes tarefas:
  - Lectura da especificación e documentación
  - Reunión cos directores do proxecto
  - Redacción do anteproxecto
- Fase de documentación, posterior á execución dos sprints, duraría un prazo de 5 semanas e constaría das seguintes tarefas:
  - Revisión e documentación do código
  - Compilación de documentos cos sprints
  - Redacción da memoria
  - Redacción dun manual de usuario
  - Reunión co director para revisión

Agora que temos as estimacións da fase de inicio, da fase de documentación e da fase de desenvolvemento (cos seus sprints estimados) podemos realizar un diagrama de Gantt do proxecto (figura 3.2) para establecer a liña base do mesmo:

Considerando que coñecemos as tarefas da fase de inicio e da fase de documentación, así como os sprints da fase de desenvolvemento e incluso os RFs a desenvolver dentro de cada un deles, podemos plasmar todas estas tarefas nun Esquema de Descomposición do Traballo ou EDT (figura 3.3). Isto implica que os RFs (ou máis ben o seu desenvolvemento) van ter a consideración de tarefas a partir de agora no noso proxecto.

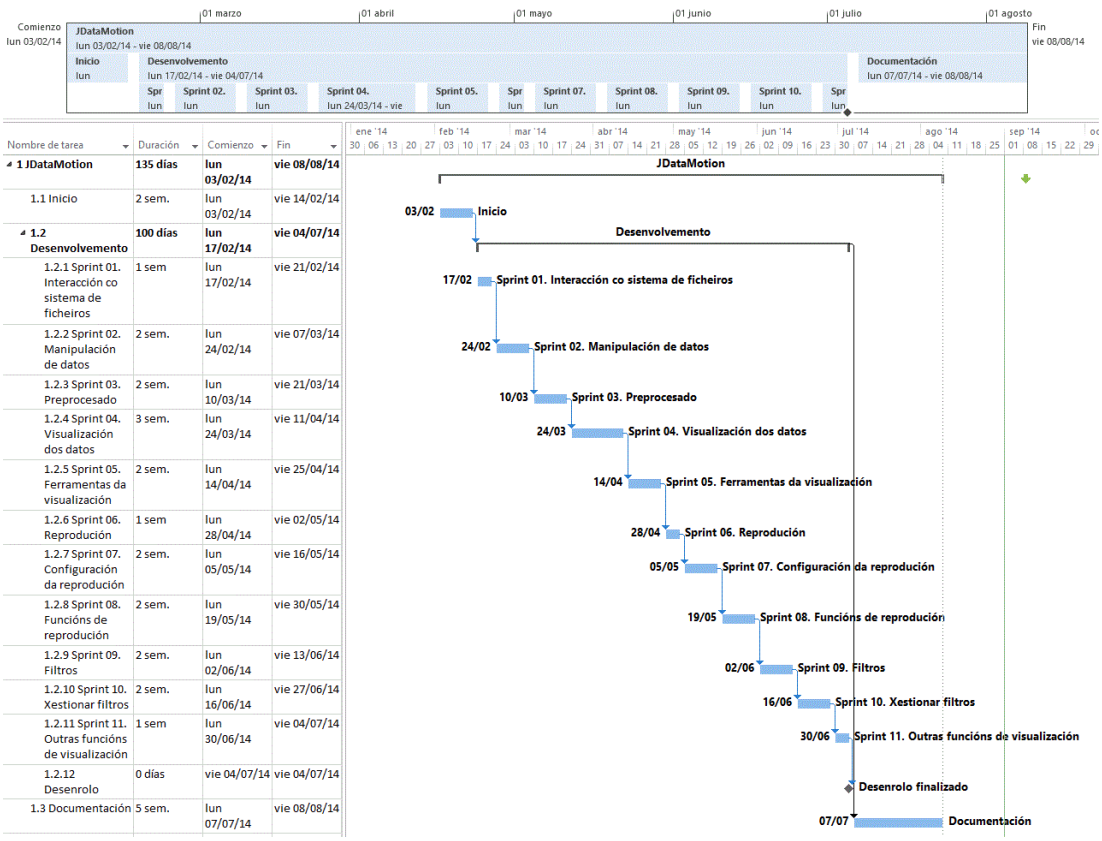


Figura 3.2: Diagrama de Gantt

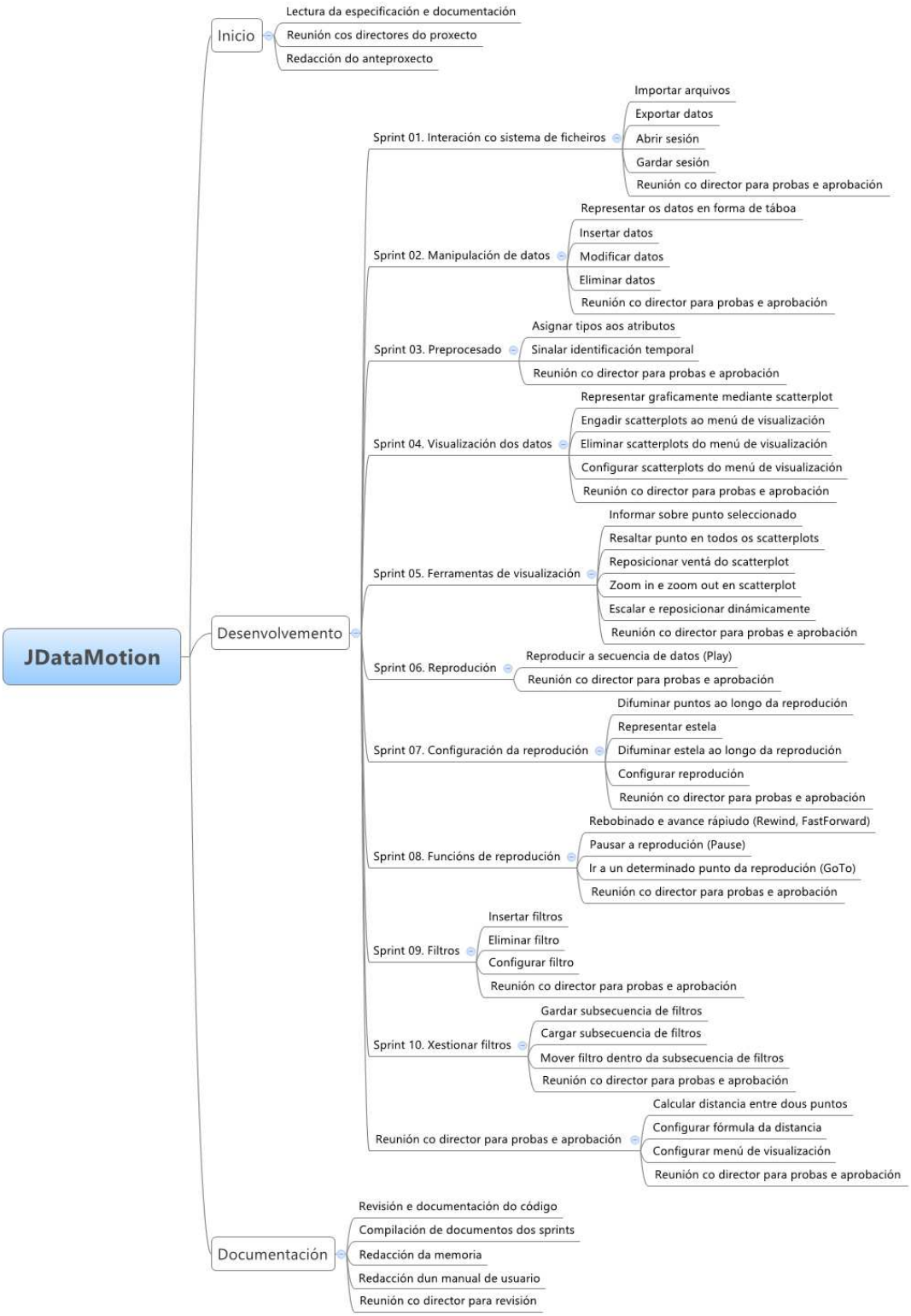


Figura 3.3: Esquema de Descomposición do Traballo (EDT)

3.2. Análise de tecnoloxías

Chegados a este punto, temos un conxunto de tarefas especificadas e delimitadas no tempo que se van realizar, así que agora abordaremos as tecnoloxías (ferramentas e técnicas) que imos utilizar para levalas a cabo.

3.2.1. Arquitectura

Nesta sección imos comentar a arquitectura sobre a que estará cimentada o noso proxecto. É importante considerar á hora de elixila que a nosa aplicación non só debe funcionar correctamente con ela, se non que ademais debe facilitar a súa mellora e evolución ao longo do tempo, ou o que é o mesmo, debe permitir a súa escalabilidade.

Na especificación deste proxecto e na captura de requisitos falouse do JDataMotion como unha ferramenta que permite visualizar dinamicamente conxuntos de datos que o usuario aporta. Esta premisa xa nos permite entrever que o modelo de datos non ten necesidade de ser extraido da máquina do cliente. Pódese traballar sobre ela de xeito local e obter resultados (visuais, ou un novo ficheiro de información procesada) sen necesidade de interactuar con sistemas externos (o que sería a computación distribuída). A arquitectura elixida será, segundo estas premisas, a dunha aplicación de escritorio.

3.2.2. Tecnoloxías

A partir do paradigma de computación que escollemos hai que elixir un conxunto de tecnoloxías que non só sexan compatibles con ela, se non que ademais permitan desenvolver satisfactoriamente os RFs pactados.

3.2.2.1. Ferramentas de deseño

UML

UML (Unified Modeling Language) é un estándar de modelado perfectamente aplicable ao un entorno de desenvolvemento de software. Baséase na especificación e documentación dos compoñentes do sistema a diferentes niveis, dentro da fase de deseño do sistema. Nós empregaremos este estándar mediante 3 tipos de diagramas:

Diagramas de casos de uso:
Na figura 3.1 xa presentamos o diagrama de casos de uso do sistema.

Diagrama de clases:
Este tipo de diagrama presenta as clases que subxacen baixo o sistema, xunto cos seus métodos, atributos e interrelacións.

Diagramas de secuencia:
Estes diagramas plasman a interacción entre os compoñentes (ou obxectos) do sistema durante escenarios concretos da execución da aplicación. Neles amósase o intercambio de información e o fluxo de control entre compoñentes.

3.2.2.2. Ferramentas de desenvolvemento

A única imposición a nivel de desenvolvemento dada pola especificación do anteproxecto é perfectamente válida coas premisas tomadas ata agora: o sistema debe ser implementado na linguaxe de programación Java.

Java

Java é unha linguaxe de programación de propósito xeral, é dicir, contén librarías e soporte para diversos propósitos: acceso a base de datos, comunicación entre máquinas, cálculo matemático, procesamento de imaxes, etc. Tamén é unha linguaxe concorrente, pois contén fornecemento para fíos de execución que van dan lugar, entre outras moitas vantaxes, á posibilidade de crear unha interface gráfica como a que imos necesitar.

Outras vantaxes que Java nos ofrece implicitamente son as seguintes:

- Java é multiplataforma, é dicir, o código que se programa compílase unha soa vez para ser empregado en calquera sistema, sen importar a súa especificación (sistema operativo, arquitectura do computador, etc.).
- O colector de lixo, que nos permite abstraernos do proceso de liberación de recursos na memoria. Este sistema localiza e libera automaticamente na memoria aqueles obxectos que deixaron de ser referenciados (e polo tanto xa non vai volver a empregar o programa).

Java é unha das linguaxes máis populares para practicar a programación orientada a obxectos. Este paradigma, que traballa con clases e obxectos para modelar entidades reais como van ser as táboas, os filtros e por suposto os diagramas de dispersión, está particularmente ben adaptado para traballar coas interfaces gráficas. Outras vantaxes da programación orientada a obxectos das que nos imos beneficiar son a herdanza, o encapsulamento e o polimorfismo.

Para o noso proxecto, a versión de Java que imos empregar vai ser a 8, en calquera das súas actualizacións (1.8.X, sendo X calquer valor). Para a execu-

ción do JDataMotion deberemos ter instalada unha versión igual ou superior da máquina virtual de Java, que podemos descargar dende o sitio web oficial [4].

JFreeChart

Na especificación do proxecto tamén se deixa caer unha decisión que afectará significativamente ao desenvolvemento do proxecto cerca das últimas tarefas deste (sprints asociados a visualización dos datos) e é o de elixir unha libraría gráfica axeitada para as nosas necesidades á hora de representar os diagramas de dispersión.

Facendo un balance das principais solucións que existen actualmente na web, obtivemos:

- JFreeChart
- JZY3D
- Project Waterloo
- GRAL
- JChart2D
- JenSoft Java Chart API
- JRobin

Deste conxunto saleu como mellor posicionado o proxecto JFreeChart [2] de acordo ás seguintes características:

- É a solución que conta cun soporte máis activo. JFreeChart parecía ser o único proxecto no momento de tomar esta decisión que seguía sacando novas versións, actualizacións e parches. A solución JZY3D, sendo a segunda máis recente, non sacou unha nova versión dende o 2013, e no caso de JRobin a última versión é do 2011. Tendo en conta que Java 8 foi liberado en Marzo do 2014, parece ser que só JFreeChart se fixo eco da aparición da nova versión da máquina virtual, e polo tanto é a única librería que lle estará sacando partido a estas melloras.
- Non só é unha solución de software libre, senón que ademais é de código aberto. Isto será determinante á hora de acceder aos métodos e clases da librería, pois deberemos sobrescribir algúns para que se adecúen ao noso proxecto. De todos xeitos, as demais solucións presentadas tamén debían reunir esta característica á hora de ser elixidas.
- A documentación de JFreeChart é moi completa, os Javadocs da súa interface de programación de aplicacións (API) explican perfectamente a funcionalidade da librería, co que se reducen os custes temporais de aprendizaxe.

- O seu deseño favorece a herdanza de clases en beneficio das necesidades do noso proxecto.
- Incorpora as funcións de configuración gráfica necesarias no que atinxe aos diagramas de dispersión. Por medio de menús contextuais, cada diagrama de dispersión pode fixar a cor de fondo, títulos e eixos, ou a fonte e tamaño de letra das etiquetas por exemplo. Tamén pode exportar en distintos formatos de imaxe o diagrama en cuestión, copialo ao portapapeis.
- Permite a adición dinámica de puntos ao diagrama de dispersión. Isto será clave no proceso de reprodución dos datos.
- Incorpora as funcións de axuste da ventá de visualización para os diagramas de dispersión. Cada vez que creemos un diagrama de dispersión, este xa contará coa posibilidade de ser desprazado ou escalado dentro da súa ventá de visualización. Ademais tamén mellora a inserción dinámica de puntos no diagrama, pois cada vez que un punto se engade ao diagrama, a ventá reposiciónase para seguir abarcando todos os puntos.

JCommon

JCommon é unha librería da cal depende JFreeChart. Temos que acoplala ao proxecto, pero non imos ter necesidade de utilizala directamente.

Formatos de arquivo

No RNF01 fálase de dous tipos de arquivo de entrada e saída: “csv” e “arff”.

O formato csv (comma-separated values) de arquivos almacena os datos dun xeito tabular simple. A primeira liña dun arquivo csv contén separados por N-1 comas os N nomes dos atributos ou columnas da táboa (se segue o estándar). A continuación, cada nova liña conterá tamén N-1 comas para separar os N valores que toma esa entrada para cada atributo. Un valor nulo represéntase deixando baleiro o oco correspondente.

O formato arff

é un formato de arquivo que contén a mesma información ca o csv, pero a maiores especifica o tipo de valores que se admiten para cada atributo, e tamén o nome da relación. A estrutura desta información é a seguinte:

@RELATION <nome da relación>
só unha vez en cada arquivo, sinala o nome da relación contida no ficheiro.

@ATTRIBUTE <nome dun atributo><tipo do atributo>
incluirase unha entrada coma esta por cada atributo da relacion. O tipo do atributo pode ser un dos seguintes:

- numeric (atributo numérico)

- <valor nominal 1, valor nominal 2, valor nominal 3, ...>(atributo nominal, é dicir, que só pode tomar un dos valores nominais especificados na lista)
- string (calquera cadea de caracteres)
- date [<formato de data>] (data no formato dado)

@DATA \n <entrada 1 \n entrada 2 \n entrada 3 \n ...>
despois de poñer a cláusula @DATA, engadirase unha nova liña para cada entrada. En cada entrada figurarán separados por comas os valores que esta toma para cada atributo (igual ca nos csv). Cada un desos valores asóciase ao atributo declarado na mesma posición.

% <calquera liña de texto >
os comentarios comezan co símbolo '%' e abranguen a liña enteira. Poden ir intercalados en calquera zona do ficheiro.

Weka

A raíz da documentación atopada sobre o formato arff vimos mencionado varias veces o término Weka. Weka (Waikato Environment for Knowledge Analysis) é unha plataforma de software libre para o aprendizaxe automático e a minería de datos, desenvolvido en Java pola Universidade de Waikato. Esta universidade precisamente creou o formato arff para Weka.

O software de Weka pódese obterse a través do seu sitio web [1]. A apariencia do programa tras importar un arquivo arff pódese observar na figura 3.4, e na figura 3.5 podemos observar como Weka plasma os datos do arquivo baixo unha matriz de diagramas de dispersión. Como se pode ver, Weka xa realiza as funcións de importación e visualización dos datos.

Podemos extraer de Weka non só ideas para a interface do JDataMotion e das súas funcionalidades, se non que ademais, gracias a que Weka é un proxecto de código aberto, poderemos reutilizar o proxecto como unha libraría mais, utilizando os métodos e clases públicas que ofrece a súa interface de programación (API) para implementar facilmente a importación dos datos desde arquivo. A visualización dos datos tamén podería ser reutilizada, pero os diagramas de dispersión de Weka son estáticos, e a súa solución de visualización non vai servir para o JDataMotion. Weka non implementa a reprodución da visualización que JDataMotion busca.

Swing

Swing é unha libraría gráfica expresamente deseñada para Java. Inclúe unha serie de elementos (coñecidos como "widgets") para facilitar a confección da interface gráfica a calquera desenvolvedor. Ao igual que Java, Swing é independente

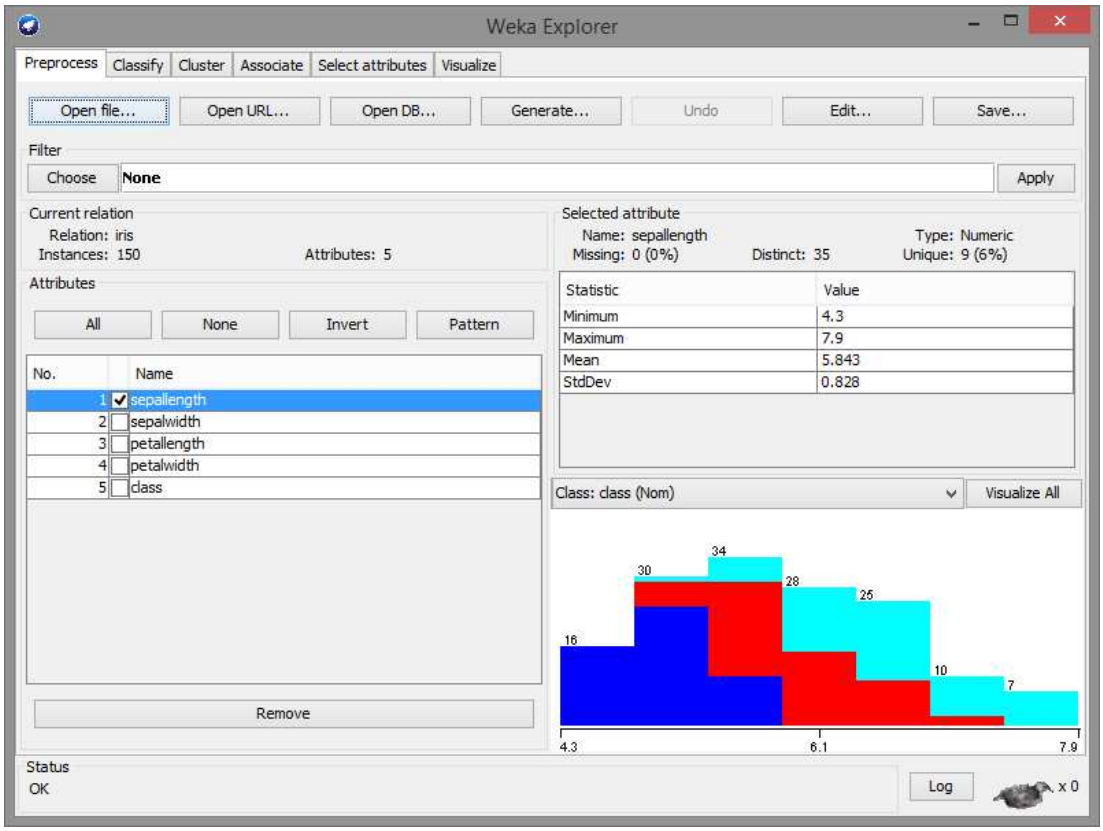


Figura 3.4: Software Weka

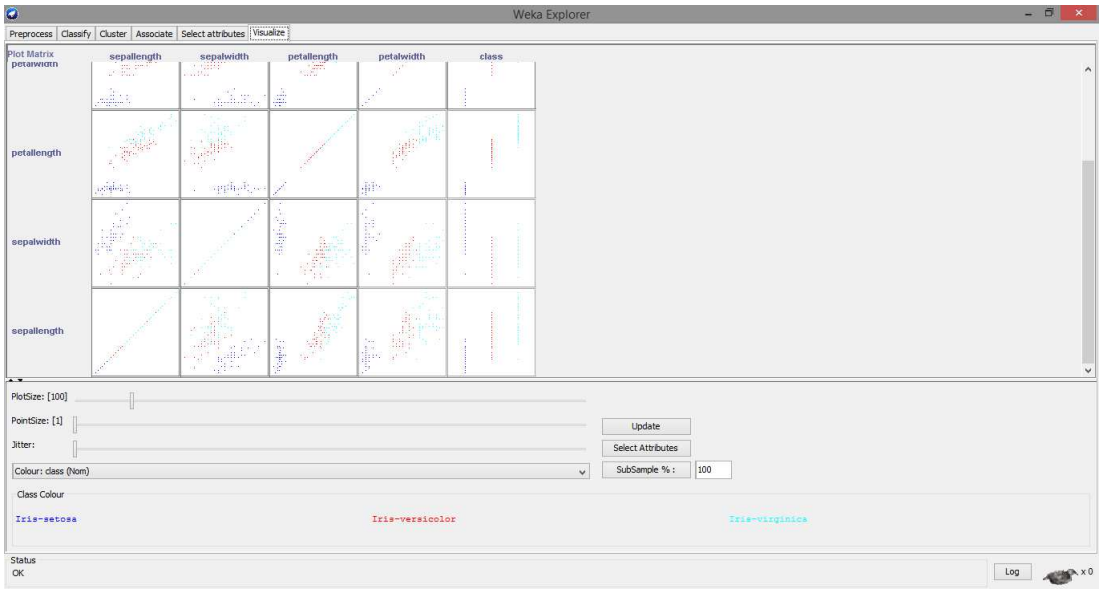


Figura 3.5: Diagramas de dispersión en Weka

da plataforma, e ademáis posibilita en gran medida e extensión de clases, ou widgets, para adaptalos ás necesidades de calquera interface gráfica.

Lumzy

Lumzy [11] é unha ferramenta online para diseñar Mockups e prototipos de aplicacións. Facilita a maioría de gadgets e elementos (botóns, etiquetas, lapelas, marcos, etc.) que empregan as interfaces gráficas, de xeito que se poda traballar con eles de xeito áxil sobre un lenzo, e apreciar mellor o resultado a nivel visual.

NetBeans IDE

NetBeans IDE é un entorno de desenvolvemento integrado, especialmente deseñado para programar sistemas en linguaxe Java, aínda que tamén da soporte a moitos outros linguaxes de programación. En NetBeans poderemos programar con facilidade as clases que logo o IDE compilará e executará. Tamén facilitará as labores de adxuntar librarías e dependencias, e dará fornecemento ao deseño da interface por medio dun lenzo e unha paleta de "widgets" propios de Swing, o cal reducirá o tempo que tardemos en aprender a manexar a libraría Swing.

3.2.2.3. Ferramentas de validación

A validación é un proceso que permite contrastar os resultados da implementación do sistema fronte aos requisitos (funcionais, non funcionais, de deseño, de calidade, etc) que inicialmente se esperaban dela.

JUnit

JUnit [10] é un compendio de librarías para aplicacións Java destinadas a realizar probas unitarias do sistema no que se inclúen. Contén un conxunto de clases que actúan sobre as clases e métodos do sistema orixinal, validando a súa resposta ante certos parámetros de entrada.

Capítulo 4

Deseño e implementación

Neste capítulo documentarase o proceso de deseño, e algúns trazos da consecuente implementación, que se irá formando de xeito incremental nas sucesivas iteracións da metodoloxía Scrum. Comentarase a arquitectura do modelo de datos que vai manexar a aplicación, así como a distribución deses datos na aplicación, o que nos obrigará a falar da interface gráfica de usuario. Despois disto, mergullarémonos no que é o deseño e máis a implementación dos compoñentes da aplicación.

4.1. Deseño global

O proxecto que imos traballar podería terse desenvolvido, atendendo á súa especificación, baixo un único módulo de traballo. É unha aplicación de escritorio, é dicir, non existe a necesidade, por exemplo, de facer un módulo cliente e un módulo servidor, e polo tanto poderíamos crear un único proxecto de nome JDataMotion que contivese toda a lóxica necesaria para ler, interpretar e procesar datos externos, fornecidos polo usuario.

Sen embargo, e a pesar de todo isto, debemos considerar o requisito de deseño RD01, que nos obriga a definir e facilitar ao usuario unha interface de programación, a cal lle permitirá programar e ensamblar os seus propios filtros no proxecto. Ademais da interface, teremos que publicar as clases das que esta depende. Desta forma, decidimos que o máis práctico era crear un segundo proxecto, ao que mencionaremos a partir de aquí como JDataMotion.common.

E seguindo nesta liña, crearase un terceiro módulo que se usará para probar e exemplificar a efectividade da relación entre os outros dous. Este proxecto chamarase JDataMotion.filters.sample e conterá algunhas clases que implementarán a interface de filtro publicada en JDataMotion.common.

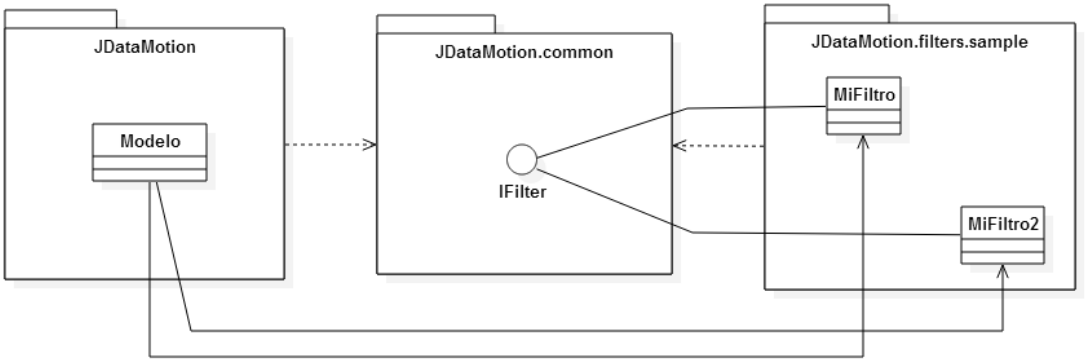


Figura 4.1: Deseño global dos 3 módulos que se incluirán dentro do proxecto

O diagrama que ilustra as relacións entre os 3 módulos sería o que se expón na figura 4.1. O módulo (ou subproxecto) JDataMotion.common contén a interface IFilter (además doutras clases das que depende a interface). Tanto o JDataMotion (a aplicación) como o JDataMotion.filters.sample dependen do módulo común, e en concreto o JDataMotion.filters.sample deberá implementar a interface naquelas clases que vaian ser susceptibles de ser importadas en forma de filtro polo JDataMotion.

A continuación abordaremos o deseño de cada un dos tres módulos implicados neste proxecto.

4.2. Deseño de JDataMotion

Os datos son o punto de partida de todas as funcionalidades deste proxecto. Todo xira arredor do arquivo con información que o usuario importa tras abrir por primeira vez o JDataMotion. O dato debe ser almacenado de xeito adecuado, e accedido só a través dos métodos e clases necesarios, para manter o fluxo de información controlado. A clase que captará, almacenará e distribuirá os datos de cara ás demais clases da aplicación recibirá o nome de Modelo, posto que realmente alberga o modelo da aplicación.

JDataMotion vai ser un programa moi dependente da súa interface de usuario. Os diagramas de dispersión non poden ser visualizados a través dunha simple terminal, e o constante fluxo de datos entre o usuario e o sistema non se pode activar a través dun menú de opcións en termos de usabilidade. Si traballamos co JDataMotion a través dunha interface gráfica multifío como as que Java permite deseñar, a aplicación pode procesar a información ao mesmo tempo que o usuario realiza outras interaccións (ver o modelo, configurar un filtro ou mesmo cancelar a visualización dinámica). Por isto, a clase Vista será un dos artefactos que máis

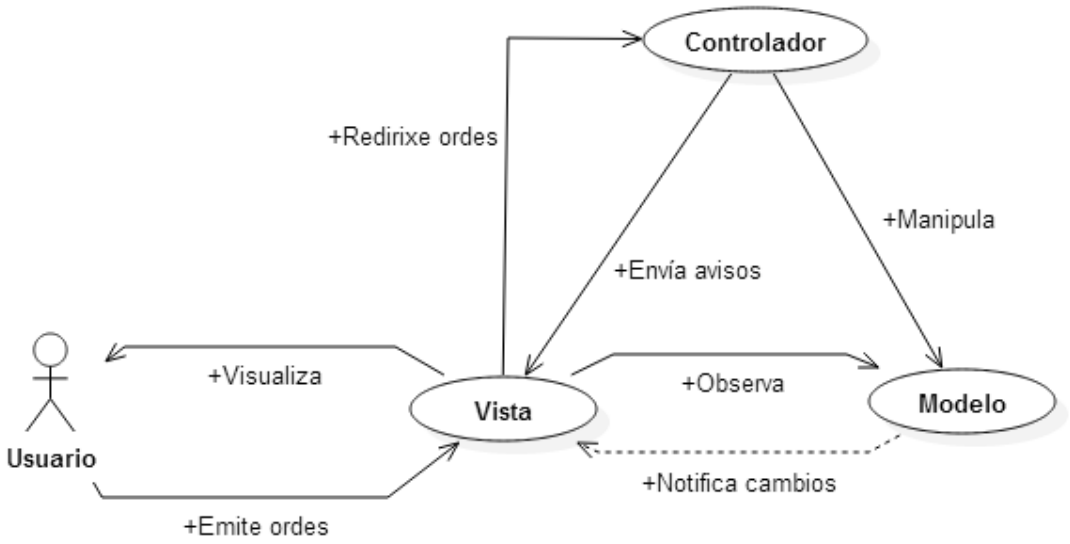


Figura 4.2: Modelo-Vista-Controlador para JDataMotion

tempo adicaremos a implementar. Vista conterá os "widgets" da libraría Swing necesarios para presentar a información ao usuario, e recibir del as novas ordes. Será unha clase complexa e de bastante peso que delegará certas funcións en clases internas ou asociadas.

Só falta unha clase que reciba da Vista as funcionalidades activadas polo usuario e cree un comando que se encargue de realizar o traballo. Esta clase deberá non só disparar o comando, se non que tamén terá que xestionalo, e reaccionar dun xeito específico en caso de que o comando chegue a unha situación de erro. A maioría de comandos influirá directa ou indirectamente sobre o Modelo. A esta clase chamarémoslle Controlador porque a súa responsabilidade será esencialmente esa, a de recibir eventos da Vista e xestionar comandos que actúen de cara ao Modelo. Deste xeito a Vista utilizará a API do Controlador, e quedará exenta de responsabilidade no caso de que se detecten fallas no Modelo.

Acabamos de definir dun xeito explícito cal é o patrón de deseño sobre o que vai xirar toda esta fase do proxecto: o Modelo-Vista-Controlador (MVC). Os patróns de deseño son solucións prácticas a problemas de deseño comúns. En concreto, o Modelo-Vista-Controlador divide o sistema en tres compoñentes coas responsabilidades definidas que xa comentamos anteriormente. A mellor forma de expoñer a aplicación exacta do patrón MVC no noso proxecto é o diagrama da figura 4.2.

As relacións marcadas cunha liña sólida son asociacións directas (unha clase que actúa de xeito explícito sobre os atributos ou métodos doutra), mentres que a relación marcada cunha liña descontinua representa unha relación indirecta (por

exemplo, unha clase que resulta afectada pola actividade doutra). As relacións entre compoñentes detallaranse a continuación.

- O usuario emite ordes en forma de eventos cara a Vista ao interactuar cos seus botóns e menús.
- A Vista identifica os eventos que recibe e redirixe cara o Controlador a orde asociada ao evento en cuestión.
- O Controlador envía un aviso directo á Vista en caso de que a orde que recibe dela levase ao sistema a un estado de erro.
- O Controlador manipula o Modelo segundo o contido da orde que recibe.
- O Modelo actúa como entidade observada de cara á Vista. A Vista ten acceso ao Modelo para actualizar a súa aparencia.
- Do mesmo xeito, que o Modelo sexa observado pola Vista implica que os cambios no modelo serán notificados á Vista, para que esta se actualice no momento do cambio.

Imos documentar máis en profundidade a relación entre a Vista e o Modelo. Comentábamos que a Vista accede ao Modelo para actualizarse cando o desexe, pero ademais o Modelo, cando cambia, notifica á Vista o suceso deste cambio. Temos entón que a Vista actúa como entidade "observadora" dun Modelo que é a entidade "observada". En esencia, estamos definindo o patrón de deseño Observer.

O patrón de deseño Observer é a mellor resposta que podemos atopar a nivel de deseño para solucionar o problema de manter actualizados ao mesmo nivel o Modelo e a Vista. Se non botásemos man desta solución, teríamos dúas alternativas:

- Refrescar a vista cada certo intervalo tempo, o cal é ineficiente e seguiría permitindo a posibilidade de que se dese unha incoherencia Vista-Modelo entre refrescos.
- Permitir ao Modelo que acceda directamente á Vista para invocar un método de actualización, é dicir, asociar directamente a Vista ao Modelo. Isto atenta contra a natureza do Modelo-Vista-Controlador, xa que o Modelo ten que almacenar a información do sistema e abstraerse por completo da Vista ou do módulo que se encargue de representar os seus contidos. Témonos que manter na premisa de que a Vista pode observar ao Modelo e acceder a el para ler datos, pero o Modelo non debe ser consciente da existencia de ningunha Vista.

O patrón Observer pódese aplicar de forma moi sinxela ao noso proxecto. Basta con facer que a Vista implemente a interface *Observer*, de xeito que a obrigue a implementar un método de actualización chamado *update()*, que se

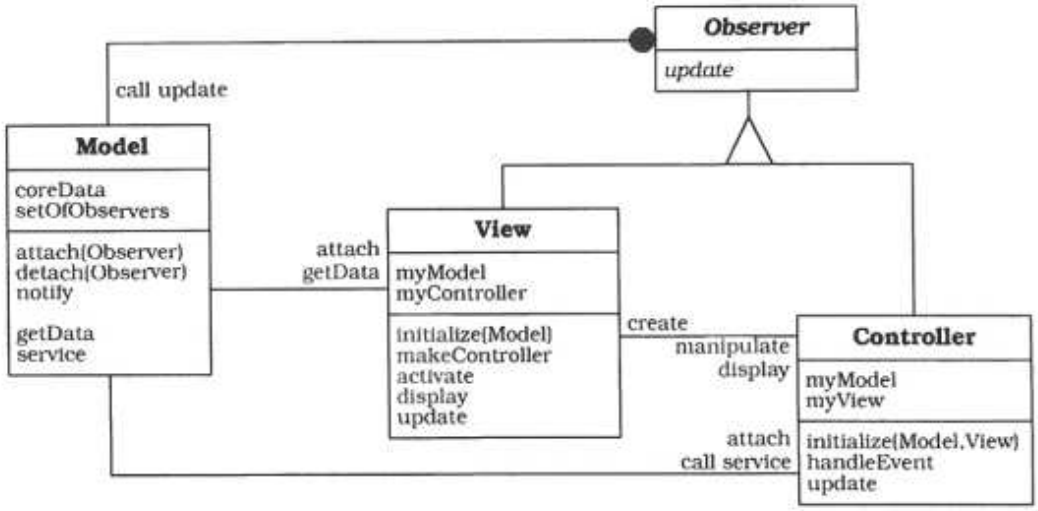


Figura 4.3: Modelo-Vista-Controlador con Observer

disparará cando un obxecto observado cambie o seu estado. Por outra parte, o Modelo só necesita estender a clase *Observable* para ser susceptible de ser observado por un *Observer*. Ao estender esta clase, a Vista xa pode chamar ao método *addObserver(Observer o)* do Modelo, pasándose a ela mesma como parámetro para así incluírse como observadora e ser notificada dos seus cambios.

A asociación dos patróns Modelo-Vista-Controlador e Observer baixo esta forma é bastante común. O libro Pattern-Oriented Software Architecture [13] fai unha boa exposición do que acabamos de mencionar, e aporta o diagrama da figura 4.3 para sintetizar ambos patróns.

Probablemente a diferencia máis destacable que podemos atopar entre este diagrama e o diagrama que correspondería coa nosa aplicación sería que no noso caso só a Vista implementará a interface *Observer*. O Controlador no noso proxecto non actuará de observador, pois non ten que recibir notificacións ante os cambios do Modelo, xa que de feito el é o responsable directo de todos eses cambios, e non necesita ser notificado de cambios do modelo que non provocase el.

Acabamos de expoñer as 3 clases principais do proxecto e as súas interrelacións, pero debemos ter en conta que cada unha desas clases necesita a axuda doutras que colaboren con ela no obxectivo de cumprir coas súas responsabilidades. O que faremos a continuación será ampliar as 3 entidades básicas do MVC aplicadas ao noso proxecto.

4.2.1. Modelo

O Modelo, como comentábamos, é o responsable de captar e almacenar a información coa que traballamos. A información almacenada clasifícase en 3 conxuntos:

Instancias:

Son as tuplas de información coas que se traballa. A estrutura para almacenalas e xestionalas foi reutilizada a partir da API de programación de Weka: a clase Instances. Esta estrutura almacena tanto as propias instancias ou tuplas de información, coma os atributos que levan asociadas. Ademais, contempla toda a información relacionada co atributo: nome, tipo, rango, etc. Esta clase tamén nos facilitará todos os métodos que necesitamos para acceder ou modificar instancias ou atributos. Ademais, a clase Instances de Weka tamén almacena un nome para a relación.

Como veremos, non empregaremos a clase Instances directamente, se non que a estenderemos a través dunha clase á que chamaremos ComparableInstances. A única diferencia entre ambas é que ComparableInstances sobreescribe o método equals(Object o), para determinar que dúas instancias sexan iguais en canto a nome da relación, atributos e tuplas. Isto resultará clave á hora de realizar as probas da aplicación, xa que permitirá verificar que un experimento acada un estado esperado ao aplicarlle unha acción, demostrando a efectividade da mesma.

Filtros:

O Modelo tamén albergará a secuencia de filtros que utilicemos no noso experimento. Utilizará para isto unha estrutura de tipo lista na que cada novo filtro se engada ao final (se ben é posible cambiar a posición dos filtros e polo tanto a orde da súa aplicación).

Para seren útiles de cara ao Modelo, os filtros deben de conter unha pequena porción de lóxica que a interface IFilter non pode incorporar. Por isto, o Modelo tratará de encapsular cada IFilter co que traballe dentro dun FilterHandler. O FilterHandler só é un manexador da interface que contén, pero con outros campos necesarios como o índice do atributo sobre o que opera o filtro, ou un parámetro de bandeira que indica si o filtro está seleccionado. En canto a métodos, o FilterHandler tamén ofrece ao Modelo un conxunto de procedementos prácticos para traballar co filtro en cuestión.

Outros elementos:

O Modelo tamén almacena outros datos máis discretos acerca do experimento, como son os índices de atributo temporal e nominal que se empregarán na visualización, a dirección ao ficheiro de orixe e o resumo SHA1 do mesmo (para posibilitar a creación e restauración de sesións).

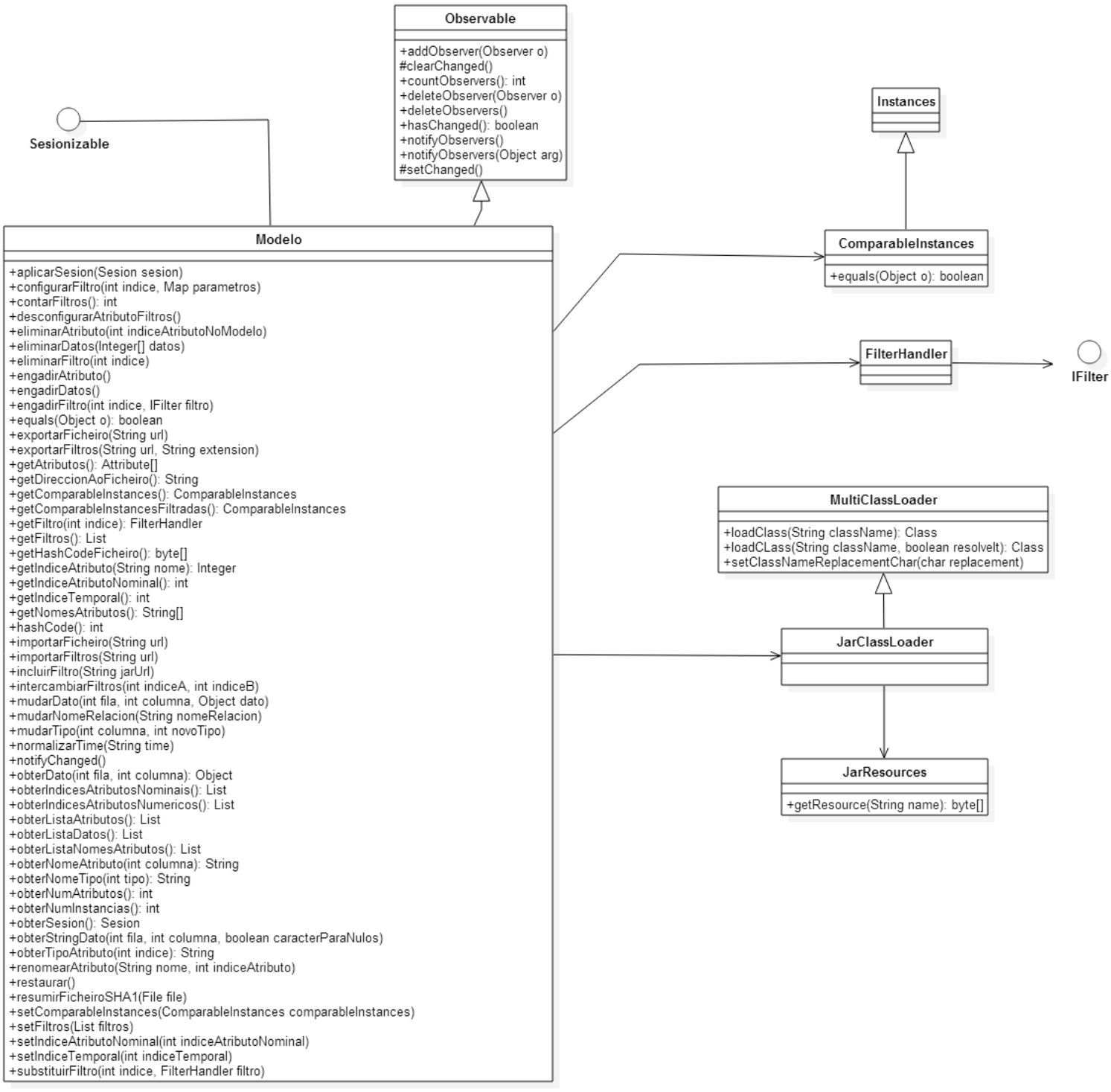


Figura 4.4: Diagrama de clases do modelo

A continuación expoñeremos e comentaremos o deseño da sección Modelo. O diagrama de clases preséntase na figura 4.4. O Modelo debe contar cunha gran batería de métodos útiles para o Controlador. Isto significa que se debe crear un número suficiente de métodos concretos para as necesidades de almacenamento de datos que vaia ter o proxecto, pois as responsabilidades do Controlador non contemplan o traballo directo con unidades de información pesadas como son as ComparableInstances. Isto é, polo xeral será preferible esixirlle ao Modelo unha tupla de información determinada, ca pedirlle todas as ComparableInstances e mesturar dentro do Controlador a xestión de eventos coa procura dunha tupla dentro das instancias totais. En definitiva, en certa medida o Modelo debe ofrecer ao Controlador unha API de programación que atenda ás súas necesidades individuais.

O Modelo, como xa comentáramos anteriormente, estendía a clase Observable (neste caso escolleremos a do paquete java.util). Ademais implementará a interface Sesionizable, que permitirá que os seus campos sexan almacenados nunha sesión para ser recuperados posteriormente. Os dous métodos que deberá implementar serán obterSesion(), que obligará ao Modelo a devolver unha SesionModelo cos datos que desexa salvar, e aplicarSesion(Sesion sesion), que tratará de recuperar a sesión a partir da SesionModelo previamente obtida.

Os datos do Modelo que desexaremos salvar son:

- A dirección ao ficheiro de orixe do experimento
- As cabeceiras do experimento (o conxunto dos atributos que manexa).
- O índice do atributo que representa a temporalidade.
- O resumo ou hash do ficheiro de orixe, para protexer a integridade.
- O nome da relación coa que se está a traballar.
- O índice do atributo nominal (atributo de clase) que se está a empregar.

Un dos métodos máis destacables do Modelo é notifyChanged(). Este método comproba que o obxecto en cuestión cambiou o seu estado, e en caso afirmativo invoca ao método update(Observable o, Object arg) de todos cantos obxectos estean observando ao Modelo. A comprobación de que o Modelo cambiou realízase cos métodos setChanged() e clearChanged() herdados da clase Observable, que activan ou desactivan unha variable de bandeira. O método setChanged() será chamado polo Modelo cada vez que este acade un novo estado que deba ser notificado (por exemplo, que se engadise unha nova instancia). A continuación, cando o Controlador invoque o notifyChanged() do Modelo comprobarase a bandeira, si esta está activada invocarase ao update(Observable o, Object arg) dos observadores, e a continuación con clearChanged() volverase a desactivar, indicando que non houbo cambios dende a última actualización. O Modelo tamén

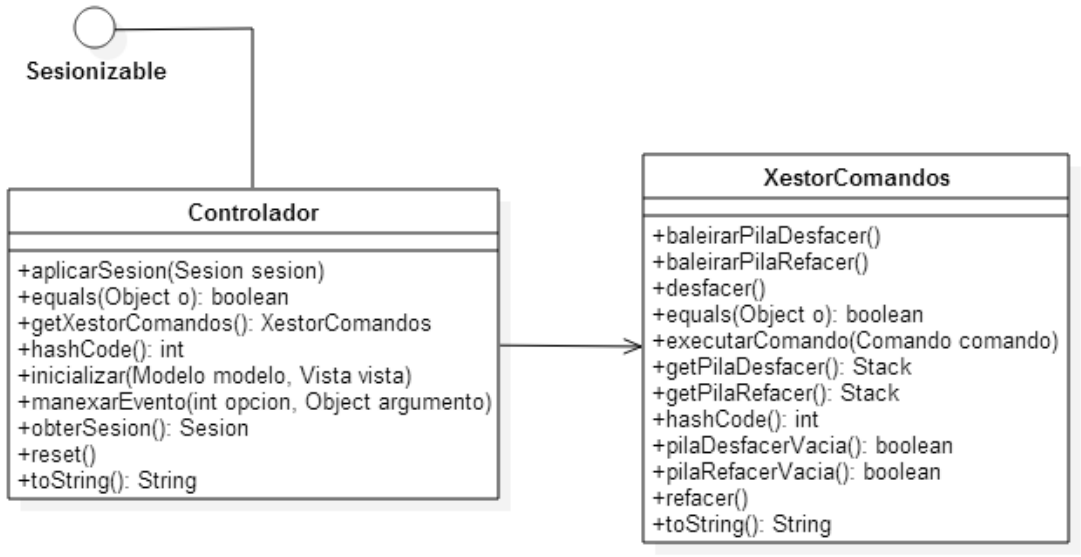


Figura 4.5: Diagrama de clases do controlador

contén un método reset() que reinicia a súa actividade previa cada vez que comeza un novo experimento.

Outras das clases das que se valerá o Modelo para realizar a súa labor é JarClassLoader. Esta clase estende a MultiClassLoader e utiliza a JarResources para permitir cargar en tempo de execución proxectos .jar xa compilados, posibilitando a importación dinámica de filtros. A clase JarClassLoader será estática e con visibilidade suficiente para permitir que a Vista a use, xa que a vai necesitar para ler directamente dende o directorio onde o Modelo almacena os filtros en .jar que importa.

4.2.2. Controlador

A continuación expoñeremos e comentaremos o deseño da sección Controlador. O diagrama de clases preséntase na figura 4.5. O Controlador é o motor da aplicación, a clase involucrada na maioría das transaccións que afectan ao experimento. A súa tarefa principal é recibir o fluxo de eventos que a Vista notifica, e realizar a operación axeitada en consecuencia. Si a operación require dun procesamento complexo dos datos, o Controlador delegará na lóxica do Modelo para obter o resultado que desexa.

O Controlador, como motor da aplicación, contén o método inicializar, que recibe unha Vista e un Modelo para comezar a desempeñar as súas funcións. Tamén contén un método reset() que reinicia a súa actividade previa cada vez que comeza un novo experimento.

Evento	opcion	argumento
Importar ficheiro	Controlador.IMPORTAR_FICHEIRO	String url
Abrir sesión	Controlador.ABRIR_SESION	String url
Gardar sesión	Controlador.GARDAR_SESION	String url
Exportar ficheiro	Controlador.EXPORTAR_FICHEIRO	{String extension, String path}
Mudar dato	Controlador.MUDAR_DATO	{int numFila, int numColumna, Object novoDato}
Engadir datos	Controlador.ENGADIR_DATOS	-
Eliminar datos	Controlador.ELIMINAR_DATOS	Integer[] numsFilas
Desfacer	Controlador.DESFACER	-
Refacer	Controlador.REFACER	-
Restaurar	Controlador.RESTAURAR	-
Mudar índice temporal	Controlador.MUDAR_INDICE_TEMPORAL	int novoIndiceTemporal
Mudar tipo	Controlador.MUDAR_TIPO	{int numColumna, int novoTipo}
Mudar nome relación	Controlador.MUDAR_NOME_RELACION	String novoNomeRelacion
Renomear atributo	Controlador.RENOMEAR_ATRIBUTO	{int numAtributo, String nome}
Engadir atributo	Controlador.ENGADIR_ATRIBUTO	-
Eliminar atributo	Controlador.ELIMINAR_ATRIBUTO	int numColumna
Engadir filtro	Controlador.ENGADIR_FILTRO	{int index, AbstractFilter filtro}
Eliminar filtro	Controlador.ELIMINAR_FILTRO	int index
Configurar filtro	Controlador.CONFIGURAR_FILTRO	{int index, Parameter[] configuracion}
Intercambiar filtros	Controlador.INTERCAMBIAR_FILTROS	{int indiceFiltroA, int indiceFiltroB}
Importar filtros	Controlador.IMPORTAR_FILTROS	String url
Exportar filtros	Controlador.EXPORTAR_FILTROS	{String url, Integer[] indicesFiltros}
Importar filtros dende JAR	Controlador.IMPORTAR_FILTRO_DENDE_JAR	String url

Figura 4.6: Inputs do método manexarEvento

A pesar de que o Controlador recibe os eventos a través do seu método máis importante: manexarEvento(int opcion, Object argumento). Para procesalos, botará man dunha clase auxiliar chamada XestorComandos, que expoñeremos a continuación. A especificación de que eventos e argumentos pode recibir este método amósase na figura 4.6. Esta especificación xa figura no código fonte como o JavaDoc do método, de forma que durante a implementación teñamos acceso áxil á especificación deste método, cando estemos configurando dende a Vista as chamadas a este método do Controlador.

O parámetro opción é un enteiro que representa ao evento, e todos os valores que pode tomar son constantes declaradas estaticamente na propia clase Controlador. O parámetro argumento debe ser de un tipo ou outro en función da opción escollida. Nos casos nos que na especificación figure unha lista de tipos entre corchetes (String a, Integer b), referirase a un array de obxectos ou Object[] (primeiro elemento de tipo String, segundo elemento de tipo Integer).

A clase Controlador recibe eventos a través de manexarEvento, e invoca a un método do XestorComandos para procesalo. Este método non será outro que executarComando(Comando comando). A este método o Controlador debe pasarlle unha instancia do comando que queira procesar, de xeito que a clase Controlador está traducindo eventos que recibe da Vista en comandos que envía ao XestorComandos. Entre as responsabilidades do Controlador tamén está a de recoller as excepcións que poda devolver o XestorComandos, e procesalas adecuadamente (por exemplo ordenándolle á Vista que informe do erro).

Ás veces os eventos non se traducen a comandos, se non que se procesan directamente polo controlador, por exemplo no caso de que o evento desencadee a necesidade de acceder ao Modelo e máis á Vista para aplicar sesións neles. O proceso de tradución de cada evento explícase a continuación:

IMPORTAR_FICHEIRO:
Envía ao XestorComandos unha instancia de ComandoImportarFicheiro

ABRIR_SESION:
Carga os filtros personalizados da carpeta filters no ClassLoader (por si a nova sesión botase man deles) e acto seguido aplica na Vista, no Modelo e no propio Controlador cadansúa sesión contida no ficheiro.

GARDAR_SESION:
Solicita as sesións (co método obterSesion(), da interface Serializable) de Modelo, Vista e a do propio Controlador para almacenalas nun ficheiro.

EXPORTAR_FICHEIRO:
Envía ao XestorComandos unha instancia de ComandoExportarFicheiro.

MUDAR_DATO:
Envía ao XestorComandos unha instancia de ComandoMudarDato.

ENGADIR_DATOS:
Envía ao XestorComandos unha instancia de ComandoEngadirDatos.

ELIMINAR_DATOS:
Envía ao XestorComandos unha instancia de ComandoEliminarDatos.

DESFACER:
Invoca ao método desfacer() de ManexadorEventos.

REFACER:
Invoca ao método refacer() de ManexadorEventos.

RESTAURAR:
Envía ao XestorComandos unha instancia de ComandoRestaurar.

MUDAR_INDICE_TEMPORAL:
Envía ao XestorComandos unha instancia de ComandoMudarIndiceTem-

poral.

MUDAR_TIPO:
Envía ao XestorComandos unha instancia de ComandoMudarTipo.

MUDAR_NOME_RELACION:
Envía ao XestorComandos unha instancia de ComandoMudarNomeRelacion.

RENOMEAR_ATRIBUTO:
Envía ao XestorComandos unha instancia de ComandoRenomearAtributo.

ENGADIR_ATRIBUTO:
Envía ao XestorComandos unha instancia de ComandoEngadirAtributo.

ELIMINAR_ATRIBUTO:
Envía ao XestorComandos unha instancia de ComandoEliminarAtributo.

ENGADIR_FILTRO:
Envía ao XestorComandos unha instancia de ComandoEngadirFiltro.

ELIMINAR_FILTRO:
Envía ao XestorComandos unha instancia de ComandoEliminarFiltro.

CONFIGURAR_FILTRO:
Envía ao XestorComandos unha instancia de ComandoConfigurarFiltro.

INTERCAMBIAR_FILTROS:
Envía ao XestorComandos unha instancia de ComandoIntercambiarFiltros.

IMPORTAR_FILTROS:
Envía ao XestorComandos unha instancia de ComandoImportarFiltros.

EXPORTAR_FILTROS:
Envía ao XestorComandos unha instancia de ComandoExportarFiltros.

IMPORTAR_FILTROS_DENDE_JAR:
Envía ao XestorComandos unha instancia de ComandoImportarFiltrosDendeJAR.

Cada vez que se termina de executar un comando, o Controlador solicita ao Modelo a execución de notifyChanged, para que en caso de que este último sufrise modificacións con motivo da execución do comando, a Vista sexa capaz de plasmar a nova información. Xuntando a relación evento-comando coa aplicación do patrón Observer conseguimos que o noso sistema se manteña sempre actualizado ante calquera cambio no Modelo, xa que a causa dos cambios debe pasar polo método manexarEvento do Controlador, e este método sempre remata chamando a notifyChanged. Ambas entidades manteranse sincronizadas sen recorrer a esperas activas ou refrescos innecesarios. Para apreciar mellor as interaccións

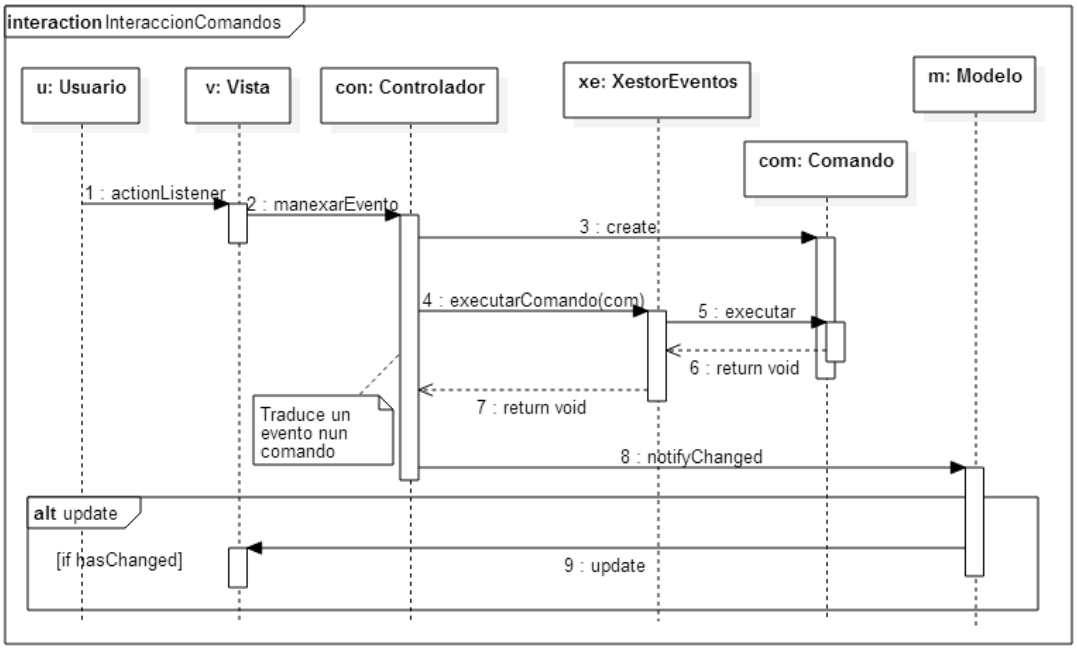


Figura 4.7: Diagrama de secuencia evento-notificación

durante todo este proceso, a figura 4.7 contén un diagrama de secuencia que o ilustra.

Os comandos que o Controlador emite ao XestorComandos son a forma de manter controladas as transaccións que alteran ao Modelo. Os comandos seguen unha xerarquía que se expón na figura 4.8.

Todos os comandos realizan a súa función no método executar(). O construtor da superclase Comando sempre debe recibir un Modelo (ao que tratará co nome de obxectivo). Este obxectivo, que poderemos recuperar por medio de getObxectivo(), é o que se terá que modificar no método executar(). O outro parámetro que debe recibir o construtor de Comando é o nome do comando en cuestión, que poderemos recuperar por medio de getNome().

A maioría de Comandos que o Controlador pode elixir para pasarlle ao XestorComandos son clases que estenderán a superclase ComandoDesfacible. Cando un XestorComandos recibe un Comando que deriva de ComandoDesfacible, almacenarao na pila de desfacer tras a súa correcta execución. Deste xeito, si o Controlador recibe un evento solicitando desfacer o último comando, executará o método desfacer() do XestorComandos directamente. Análogamente poderanse refacer comandos desfeitos. Cabe destacar que os eventos Desfacer e Refacer non xeran comando algún, se non que se tratan chamando ao método desfacer() ou refacer() directamente. O diagrama de secuencia destes dous eventos podémolo

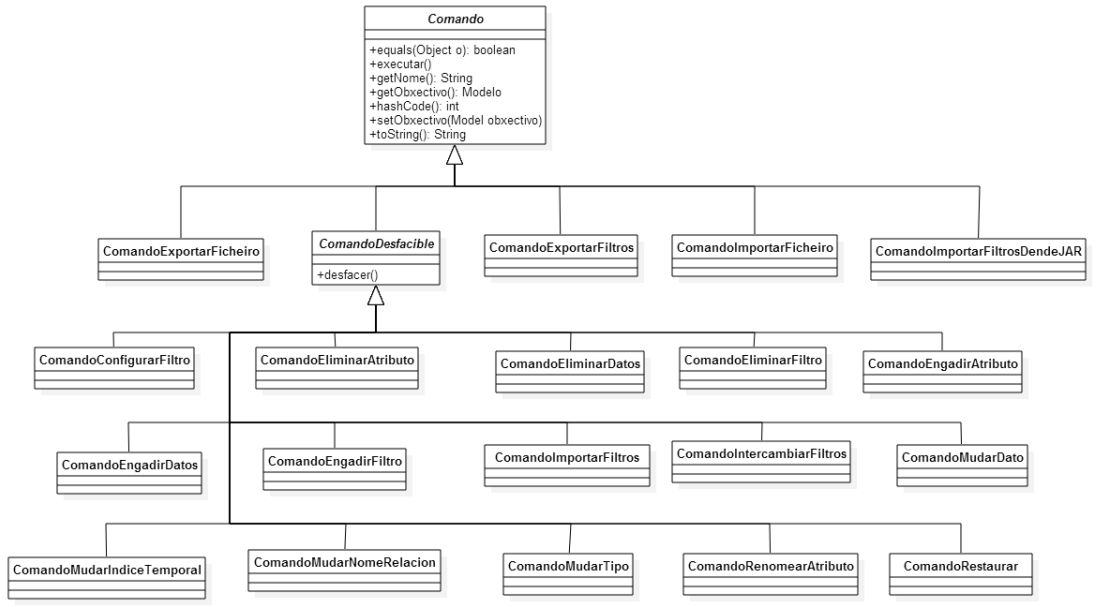


Figura 4.8: Xerarquía de comandos

observar na figura 4.8.

Do mesmo xeito que se debe implementar o método executar() para todos os comandos atendendo ao seu propósito, no caso dos ComandosDesfacibles tamén debemos implementar o método desfacer(), otorgándolle a este un comportamento que permita inverter os efectos do método executar(). A coherencia entre os procedementos executar() e desfacer() é unha responsabilidade á hora de programar comandos útiles para JDataMotion.

Existe un último grupo de comandos que comentar, e é aquel composto por comandos que aínda que estenden a clase Comando, non estenden a clase ComandoDesfacible. Estes comandos son executados sen posibilidade de entrar ou saír das pilas pilaDesfacer ou pilaRefacer que contén o XestorComandos. Trátase dos seguintes comandos:

ComandoExportarFicheiro:
A exportación dun experimento supón a creación dun novo ficheiro no sistema. Esta acción non debe poder desfacerse, xa que atentaría contra a propia finalidade do comando, que é a de salvar permanentemente os datos do experimento no disco duro.

ComandoExportarFiltros:
Baixo a mesma filosofía anterior, os accesos a disco carecen da necesidade de ser reversibles.

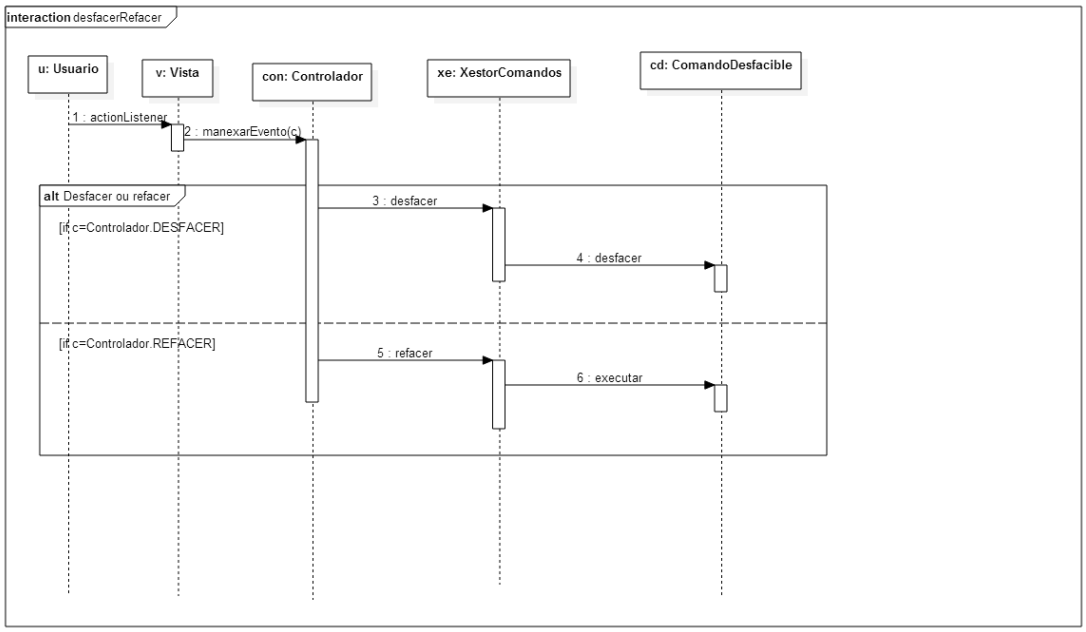


Figura 4.9: Diagrama de secuencia dos eventos desfacer e refacer

ComandoImportarFicheiro:
O inicio dun novo experimento reinicia por completo os datos do sistema. Non se deben manter as pilas do XestorComandos dun experimento a outro.

ComandoImportarFiltrosDendeJAR:
Este comando realiza unha copia do ficheiro JAR nos directorios da aplicación e carga as súas clases no ClassPath de Java. Mentres o JDataMotion se execute, este ficheiro estará en uso e non se poderá eliminar.

O Controlador, do mesmo xeito ca o Modelo, implementa a interface Sesionizable, que permitirá que os seus campos sexan almacenados nunha sesión para ser recuperados posteriormente. Os dous métodos que deberá implementar serán obterSesion(), que obligará ao Controlador a devolver unha SesionControlador cos datos que desexa salvar, e aplicarSesion(Sesion sesion), que tratará de recuperar a sesión a partir da SesionControlador previamente obtida.

Os datos do Controlador que desexaremos salvar son:

- O xestor de comandos, con todos os comandos das pilas desfacer e refacer. Cos da primeira poderemos restaurar o modelo de datos a partir das instancias iniciais do experimento.

4.2.3. Vista

A continuación expoñeremos e comentaremos o deseño da sección Vista. O diagrama de clases preséntase na figura 4.10, e contén un conxunto de clases (algunhas privadas e internas á clase Vista) que participan e comparten certas responsabilidades con ela. A sección Vista ten varios cometidos nesta aplicación:

- Implementar a interface gráfica da aplicación, as súas ventás e todos os widgets necesarios para permitir a interacción co usuario.
- Inicializar, configurar e xestionar o conxunto de diagramas de dispersión da aplicación.
- Implementar métodos de reprodución sobre o conxunto de diagramas de dispersión.

Imos comentar as clases do diagrama anteriormente mencionado, así como as súas relacións e métodos.

Vista:

É a clase de partida da sección, a primeira en ser instanciada. Estende á clase JFrame, que é o widget da librería Swing empregado para representar as ventás, pois a clase Vista é efectivamente unha ventá sobre a que se colocan outros widgets. Implementará 3 interfaces que a obrigarán a darlle corpo a unha serie de métodos:

Observer:

De acordo co patrón Observer, a clase Vista vai ser a clase observadora, neste caso da clase observable Modelo. Isto significa que cando o Modelo cambie algún aspecto do seu contido e chame ao método de actualización, a clase Vista executará o método update(Observable o, Object arg), sendo neste caso os argumentos da función de pouca utilidade, xa que o obxecto observado será sempre o Modelo, e non se necesitarán argumentos. O corpo deste método tratará de refrescar toda a interface gráfica, pintando outra vez os contedores dos 3 menús: Modelo, Filtros e Visualización. Refrescar o menú Modelo e Filtros non será normalmente un proceso custoso, pero co terceiro, no caso de que a matriz de diagramas de dispersión que se quere visualizar teña uns 1000 diagramas ou máis, a latencia comezará a facerse notable. Intentaremos combater esta problemática alixeirando o proceso na medida do posible, e a partir de aí comezaremos a traballar en termos de usabilidade para facer o proceso menos pesado de cara ao usuario. Algunhas medidas para isto comentaranse a continuación, pero a primeira delas que adoptamos consistirá en bloquear o refresco deste menú ata que se cambie á lapela de Visualización, permitindo

varias edicións nos menús Modelo e Filtros que eviten a latencia de refresco deste último menú.

Sesionizable:
Analogamente ás clases Modelo e Controlador, a clase Vista tamén pode gardar o seu contido (e o das clases que dependan dela) dentro dunha sesión. A interface Sesionizable permitirá que os seus campos sexan almacenados nunha sesión para ser recuperados posteriormente. Os dous métodos que deberá implementar serán obterSesion(), que obligará á Vista a devolver unha SesionVista cos datos que desexa salvar, e aplicarSesion(Sesion sesion), que tratará de recuperar a sesión a partir da SesionVista previamente obtida.

Os datos da Vista que desexaremos salvar, e que ampliaremos a continuación, son os seguintes:

- O conxunto de diagramas de dispersión que foron marcados para visualizar.
- A orde de visualización seleccionada para a reprodución dos datos.
- A lonxitude da estela.
- A cor da estela.
- O paso en milisegundos utilizado na reprodución.
- A fórmula da distancia configurada

PropertyChangeListener:
Esta interface permite que a Vista reaccione ante o cambio dalgunha propiedade noutra instancia, en concreto utilizaremos esta capacidade para "escoitar" os cambios no ManexadorScatterPlots da propiedade "estadoReprodutor". Deste xeito, a barra cos botóns de reprodución implementados na Vista poderá manterase actualizada ante calquera cambio no reprodutor. Esta sincronización producirase no corpo do método propertyChange(PropertyChangeEvent evt).

Outros métodos da Vista, aparte dos asociados ás interfaces, que merece a pena destacar son:

doEditProperties(Component c):
Visualiza dentro do compoñente c (por exemplo a propia Vista) un editor de parámetros para diagramas de dispersión. É dicir, abre un cadro de diálogo onde se poden seleccionar cores de fondo dos diagramas, forma e cores dos puntos, fonte das marcas dos eixos, etc. Toda a información recollida neste diálogo é primeiramente almacenada en ficheiro como configuración de usuario, e a continuación invócase un

refresco da configuración gráfica dos diagramas, que accede a ficheiro para utilizar sempre a última configuración gráfica a nivel global.

amosarDialogo(String mensaxe, int tipo):
Método invocado polo controlador para visualizar mensaxes de erro que poidan xurdir.

getControlador():
Este método é de acceso protected. Só desexamos que se acceda a el para os test de proba para acceder ao controlador (pois o Controlador instánciase na Vista).

getRecursosIdioma():
Devolve a instancia de ResourceBundle (unha clase parecida a un HashMap) que permite internacionalizar os textos da aplicación visibles ao usuario. Cada clave introducida nesta estrutura devolve un valor en función do idioma do sistema.

inicializar(Modelo modelo, boolean visualizar):
Método que usamos para iniciar a Vista integrándoa co modelo e cun Controlador.

reiniciarAplicacion():
Pecha e arranca de novo o JDataMotion. Isto empregarase ao cambiar o idioma ou outros parámetros que precisen dun reinicio.

reset():
Reinicia os campos da Vista (por exemplo, ao importar un novo ficheiro ou abrir unha nova sesión).

setScatterPlotsBackground():
Este método asigna a cor de fondo do menú de Visualización. Non recibe a cor como parámetro, xa que a lee da configuración de usuario, almacenada no ficheiro de configuración persoal.

GraphicConfigurationManager:
É unha clase composta por unha batería de métodos estáticos para ler e escribir parámetros de usuario. Os parámetros de usuario poden ser o idioma, as cores de fondo do menú Visualización, o formato dos puntos dos diagramas, etc. e almacénanse en ficheiro (configuracion.properties). Existe un ficheiro de configuración interno por defecto (default_config.properties) para todos os parámetros de usuario que aínda non fosen definidos. Os tipos de datos que estes métodos permiten tanto ler coma escribir son:

- Boolean, representando no ficheiro true coma 'y' e false coma 'n'
- Color, representando no ficheiro o seu formato RGB con comas entre cada compoñente.

- Double, representando no ficheiro o número coma un String.
- Font, representando no ficheiro o nome da fonte, seguido dun punto e os atributos (PLAIN, ITALIC, BOLD ou BOLD+ITALIC), e seguido dunha coma e o tamaño da fonte.
- ListColor, representando cada cor co formato dos parámetros Color, e separando cada un deles entre sí por medio de '—'.
- ListShape, representando cada forma con 'regular' no caso dun polígono regular e 'star' no caso dunha estrela, seguido dunha coma e o número de vértices da forma, e á súa vez separando cada unha das formas entre sí por medio de '—'.

Todas as clases da vista accederán a estes métodos estáticos para obter ou modificar a última configuración gráfica almacenada. Debido a isto, os métodos públicos e privados que almacenen ou lean cada propiedade deberán ter o modificador synchronized para evitar inconsistencias no sistema.

TarefaProgreso:

É unha das clases que mellora a experiencia de usuario durante a latencia que implica visualizar certo número de diagramas de dispersión. Esta clase iníciase xunto cun JProgressBar (o widget de Swing para as barras de progreso). Despois, establécese un número de puntos totais ou finais con setEnd(int end) e chámase a acumulate(int cantidade) cada vez que avancemos un número "cantidade" de puntos na nosa tarefa. A medida que os puntos aumenten, o método doInBackground() incrementará o valor da barra de progreso, ata chegar ao total estipulado. Para conseguir este comportamento é vital que TarefaProgreso estenda SwingWorker, xa que se non o Event Dispatch Thread (o fío que se encarga de visualizar e actualizar a interface gráfica) quedaría bloqueado ata que a tarefa rematase (e a barra de progreso pasaría do 0 % ao 100 % directamente). A clase SwingWorker executa o seu método doInBackground() mantendo o EDT activo, así que no seu corpo será onde programemos os incrementos da barra de progreso.

ManexadorScatterPlots:

Esta clase supón a mellor aliada que ten a Vista para xestionar os diagramas de dispersión. A Vista instancia un ManexadorScatterplots cada vez que volve a pintar o menú de Visualización, e facilítalle toda a configuración necesaria para que poda realizar o seu traballo. O seu cometido é iniciar os diagramas de dispersión (nun formato matricial), configuralos e almacenalos para que a vista poda solicitalos cando os necesite. Tamén ofrece á Vista unha serie de métodos de reprodución, coordinándose con outras clases para conseguir isto.

A reprodución será un proceso concorrente que non debe bloquear outras

funcionalidades. Mentres a visualización se desenvolve, débese poder avanzar cara adiante ou cara atrás cos botóns, cambiar a posición co desprazador ou por suposto pausar a reprodución. Todas estas interaccións van involucrar elementos compartidos coma atributos ou métodos de clases (o instante de tempo no que me atopo, o número de puntos que se visualizaron, etc.), así que para manter a consistencia do sistema será habitual declarar métodos ou bloques de execución coa partícula "synchronized".

A continuación describiremos os métodos fundamentais de ManexadorScatterplots:

addPropertyChangeListener(PropertyChangeListener propertyChangeListener):
Engade unha clase que implemente PropertyChangeListener á lista de "listeners", para notificarlle calquera cambio nunha propiedade. Neste caso o obxecto que vai estar escoitando vai ser a Vista, e quen a vai avisar de cambios no reprodutor (play, pause) vai ser unha instancia de TarefaPlay, unha clase colaboradora de ManexadorScatterPlots.

aplicarConfiguracionGraficaScatterPlots():
Este método aplica en todos os diagramas de dispersión a súa configuración gráfica (cores, fontes de marcas nos eixos, formato dos puntos debuxados, etc.). Non recibe ningún parámetro porque sempre colle a configuración gráfica de usuario almacenada no momento actual, léndoa dende o ficheiro de configuración.

columnaScatterPlotsVacia(int columna):
Indica si a columna de índice dado dentro da matriz de diagramas de dispersión está baleira. Isto significaría que non se solicitou a visualización de ningún diagrama que represente o atributo de índice dado no eixo de abscisas.

contarJFramesVisibles():
Devolve o número de diagramas de dispersión que foron ampliados nunha ventá (JFrame) aparte.

createPropertiesPopupMenu():
Devolve un menú contextual (JPopupMenu) cunha soa opción, "Propiedades". Esta opción permite editar a configuración gráfica de usuario. Esta opción está por defecto no menú contextual de todos os diagramas de dispersión, pero a Vista necesitará dispor del para engadirllo tamén ao propio contedor do menú Visualización, aínda que non haxa diagramas nel aínda.

cubrirConScatterPlot(int i, int j, List indicesNumericos, int indiceAtributoNominal):

Posiblemente un dos métodos máis importantes desta clase. Primeiro comprobará que na matriz de diagramas de dispersión o elemento (i, j) fose realmente marcado para visualizar. En caso afirmativo, instancia un ScatterPlot e colócao na matriz nesa posición. O parámetro "indicesNumericos" úsase para localizar facilmente que atributos son numéricos, e polo tanto susceptibles de ser representados, sen ter que iterar ao longo de todos os atributos. Este método será invocado na Vista, dentro do método interno que volve a pintar o menú de visualización, para encher os ocos da matriz de diagramas nos casos que sexa necesario. Para aliviar a espera en caso de matrices de diagramas de certo tamaño, este método será invocado dende un fío que non interrompa o resto da execución. Ademais, a medida que se creen, estes iranse visualizando, así que comezaremos a invocar este método pasándolle as posicións da esquina inferior esquerda da matriz, que é a zona que se visualiza primeiro dentro do scroll que ten este menú. Deste xeito, resultará menos tenso esperar a que se encha a barra de progreso si xa podemos ver algúns dos diagramas que escollemos.

filaScatterPlotsVacia(int fila):
Indica si a fila de índice dado dentro da matriz de diagramas de dispersión está baleira. Isto significaría que non se solicitou a visualización de ningún diagrama que represente o atributo de índice dado no eixo de ordenadas.

freeze():
Conxela a reprodución. O estado conxelado implica que a reprodución foi interrompida (non pausada) por un evento concreto e de curta duración, e se reanudará unha vez finalice este. No noso caso, o feito de arrastrar o deslizador no reprodutor conxela a reprodución, xa que ata que non soltemos o pivote, non se seguirá avanzando na secuencia.

getEstado():
Devolve un enteiro que representa o estado do reprodutor. Este valor é unha constante definida en ManexadorScatterPlots: PLAY = 0, PAUSE = 1 e FREEZE = 2.

getMsInstances():
Devolve un array de tantos enteiros como instancias teña o experimento. Cada enteiro representa os milisegundos asociados á instancia coa que comparte posición.

getTActual():
Devolve o momento de tempo actual da reprodución, en milisegundos.

getTFinal():
Devolve o número de milisegundos asociados á última instancia en ser

representada.

getTInicial():
Devolve o número de milisegundos asociados á primeira instancia en ser representada.

goTo(int toMs):
Despraza a reprodución ata o milisegundo "toMs".

goTo(double to):
Despraza a reprodución ata a posición relativa "to", sendo 0.0 o comezo e 1.0 o final.

goToNext():
Sitúa a reprodución no seguinte elemento.

goToPrevious():
Sitúa a reprodución no elemento anterior.

numColumnasNonBaleiras():
Devolve o número de columnas non baleiras da matriz de diagramas de dispersión, que equivale ao número de atributos que se solicitou que apareceran polo menos no eixo de abscisas dun diagrama.

numFilasNonBaleiras():
Devolve o número de filas non baleiras da matriz de diagramas de dispersión, que equivale ao número de atributos que se solicitou que apareceran polo menos no eixo de ordenadas dun diagrama.

obterCoresHSB(int cor, int totalCores):
Clase estática que devolve a cor que resulta de dividir o rango HSB en "totalCores" e coller a número "cor". Isto necesitarase, por exemplo, ao seleccionar varios puntos que están superpostos nun diagrama, para que noutros diagramas se resalten tamén coa mesma cor os que pertencen á mesma instancia.

pause():
Pausa a reprodución.

pecharJFramesChartPanel():
Pecha todas as ampliacións de diagramas de dispersión que se atopen abertas.

play():
Comeza ou reanuda a reprodución.

procesarSeleccion(ComparableInstances comparableInstances, List<Integer> indicesInstances)):

Resalta, para todos os diagramas de dispersión, todas as representacións das instancias cuxos índices se atopen dentro de "indicesInstances". Con isto acadamos o efecto de que ao seleccionar unha instancia ou punto nun diagrama, se resalte ese punto coa mesma cor en todos os demais diagramas. Cabe mencionar que nese punto que seleccionamos se atopen sobrepostas varias instancias (por iso "indicesInstances" é un array), co cal a cada unha asignaráselle unha cor diferente (método obterCoresHSB).

TarefaPlay:

É a clase que da soporte á función de reprodución (play() dentro de ManexadorScatterPlots). Debe estender tamén a clase SwingWorker, para poder realizar todo o seu traballo continuo dentro de doInBackground e así non bloquear ao Event Dispatch Thread (fío que se encarga de debuxar e refrescar a interface gráfica).

O funcionamento de doInBackground() para TarefaPlay parte dun bucle que en cada iteración comproba que o estado reprodutor sexa "PLAY" e que o instante de tempo (en milisegundos) actual da reprodución sexa inferior ao tempo de reprodución total. Entón pon ao fío de execución en estado "sleep" durante un tempo igual ao paso definido. Unha vez transcorrido este tempo, calcula cantos novos puntos se deben representar para o instante de tempo actual máis o paso durante o cal o fío foi durmido, e visualiza ese número de puntos en todos os diagramas activos. Finalmente actualiza o desprazador, o tempo actual e máis o indicador de puntos representados, antes de comezar unha nova iteración.

NodeList:

É unha estrutura de información moi útil para as necesidades de reprodución da aplicación. Trátase dun simple ArrayList de Nodos no que se sobrescribiu o método add(Nodo e) para que ao engadir un nodo, previamente se faga que o último nodo da lista enlace por medio do seu campo "Nodo next" co que se vai engadir, e o que se vai engadir enlace por medio do seu campo "Nodo previous" ao último da lista. É dicir, un NodeList non é outra cousa ca unha lista dobremente enlazada non circular.

O método addElement(E e) permite inserir na estrutura un elemento directamente, que será primeiramente encapsulado por un Nodo e a continuación inserido mediante o método que falamos ao principio. Os métodos getFirst() e getLast() manteranse actualizados coas sucesivas inserciónes para darnos acceso ao primeiro e o último nodo da estrutura.

Nodo:

É a unidade de información que manexa a NodeList. Podemos obter o seguinte nodo desta estrutura por medio de getNext() e o anterior por medio

de getPrevious(). Estes devolverán nulo se o nodo se atopa nun estremo da lista.

Os nodos conteñen un elemento de tipo E. Poderemos definir este tipo ao instancialo, e no noso caso escolleremos InstancesSimultaneas.

InstancesSimultaneas:
É unha clase que estende a un ArrayList de Instance, engadíndolle un campo "ms" que representa a marca de tempo en milisegundos asociada a todas as instancias.

A motivación desta clase é que non abonda con almacenar unha instancia en cada nodo, xa que varias instancias poden ter asignada unha marca de tempo igual, e polo tanto deben de ser representadas ao mesmo tempo.

Pódese acceder ás instancias que contén cos métodos propios do ArrayList (get()), e a maiores podemos obter os milisegundos asociados a todas elas por medio de getMs().

ScatterPlot:
A clase ScatterPlot instánciase para cubrir cada cela da matriz de diagramas de dispersión. Cada instancia de ScatterPlot contén dous ChartPanel, que son o contedor no que a librería JFreeChart aloxa cada diagrama de dispersión. Un deles (getChartPanelCela()) é ao que recorre a Vista para encher a matriz do menú de Visualización, e o outro está contido dentro dun JFrame (getJFrameAmpliado()). Este último visualizarase cando o usuario prema no botón de ampliar dentro do primeiro.

O método pintarEstela volve a debuxar a estela nos dous ChartPanel, usando o valor de lonxitudeEstela para determinar cantos puntos de recorrido terá. Tamén conta cos métodos getIndiceAtributoX() e getIndiceAtributoY() para obter os índices dos atributos que os dous diagramas do ScatterPlot representan no eixo de abscisas e ordenadas, respectivamente.

CircleDrawer:
É unha clase que implementa a interface Drawable para que podamos pintala nos diagramas de dispersión. Concretamente, no seu método draw pinta unha circunferencia partindo da cor de liña, cor de recheo e estilo de liña no construtor.

A cor de liña para cada CircleDrawer definiraa o método obterCoresHSB en función do conxunto de puntos que fosen seleccionados (por superposición), a cor de recheo é nula e o estilo de liña será sólido de grosor 1px.

XYDatasetModelo:
Esta clase estende a XYSeriesCollection, que representa a un conxunto de datos (Dataset) útiles para un diagrama de dispersión (puntos de cordena-

das (X, Y)), os cales poden ir agrupados en series (en cada serie os puntos represéntanse nun formato diferente). Este é o Dataset idóneo para esta aplicación de entre todos os que contén JFreeChart.

Os Dataset emprégaos a clase JFreeChart no seu construtor, e esa instancia de JFreeChart é a que se acompaña na instanciación dos ChartPanel, que son os contedores ou paneis que aloxan cada diagrama.

Esta clase esténdese en XYDatasetModelo para acadar o obxectivo da visualización dinámica por medio de dous métodos:

visualizarItems(ScatterPlot sp, int numeroItems):
permite visualizar "numeroItems" posicións máis dentro do NodeList de InstancesSimultaneas. Inclúe o refresco da estela. Debe ter o modificador synchronized para evitar inconsistencias no sistema.

agocharItems(ScatterPlot sp, int numeroItems):
permite agochar as últimas "numeroItems" posicións dentro do NodeList de InstancesSimultaneas. Inclúe o refresco da estela. Debe ter o modificador synchronized para evitar inconsistencias no sistema.

ChartPanelConfigurable:
Estende a ChartPanel só para sobrescribir un dos seus métodos: "doEditProperties()". Este método é chamado ao premer no item "Propiedades" dentro do menú contextual dun diagrama de dispersión. Deste xeito controlamos o seu comportamento para chamar ao método estático Vista.doEditProperties().

A motivación disto é a de bloquear a edición dos diagramas de dispersión a nivel individual que implementa a librería JFreeChart. Buscamos unha configuración global a todos os diagramas de dispersión.

JFrameChartPanel:

É un JFrame que contén un ChartPanelConfigurable.

Unha instancia de ChartPanelConfigurable (o diagrama que se emprega na matriz) e outra de JFrameChartPanel (a ventá no que se amplía) son as que compoñen cada unha das instancias da clase ScatterPlot.

4.2.3.1. Deseño da interface gráfica

Nesta sección configuraremos o deseño da interface gráfica. A Vista é a clase encargada de darlle soporte, delegando certas funcións ou responsabilidades en clases propias asociadas ou internas. Para o deseño da aplicación gráfica teremos

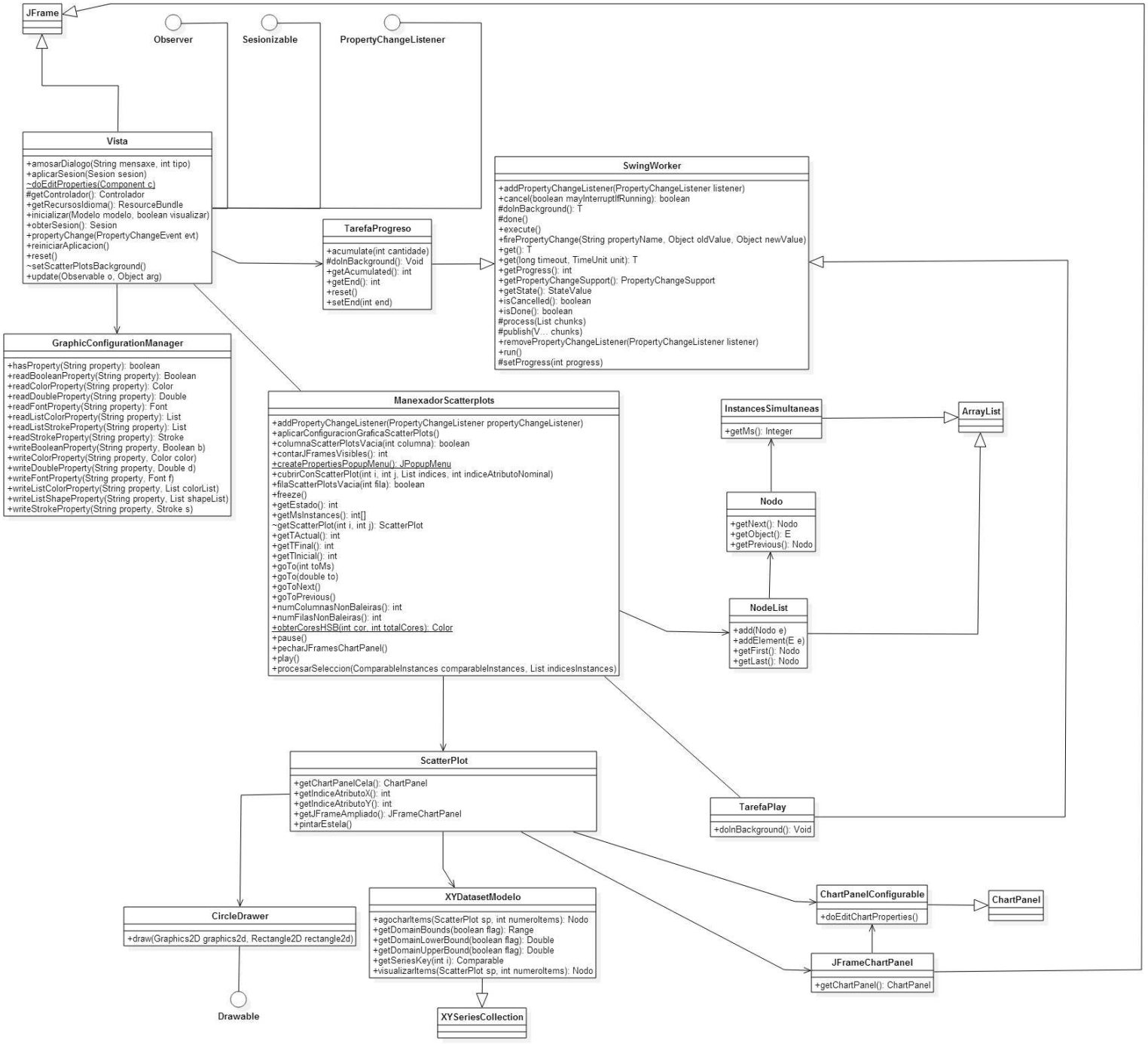
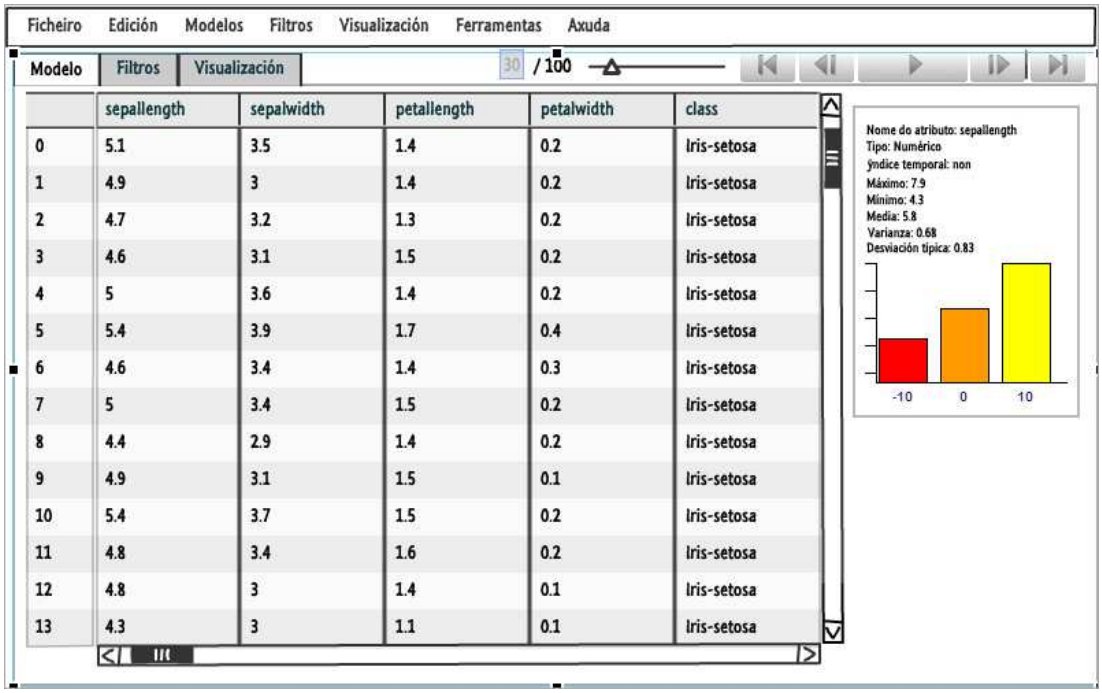


Figura 4.10: Diagrama de clases da vista



	sepallength	sepalwidth	petallength	petalwidth	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
11	4.8	3.4	1.6	0.2	Iris-setosa
12	4.8	3	1.4	0.1	Iris-setosa
13	4.3	3	1.1	0.1	Iris-setosa

Figura 4.11: Mockup da sección Modelo

en conta as posibilidades da librería gráfica Swing de Java, así como os elementos ou gadgets que a compoñen.

A utilidade da ferramenta radica no seu uso secuencial: comezamos o noso traballo a partir dun ficheiro (de datos ou de sesión) para revisar os datos e incluso formatealos ou editalos, opcionalmente aplicamos algún filtro global que afecte a todos os campos dunha determinada columna e finalmente visualizamos o resultado. A interface pode colaborar a que este proceso sexa intuitivo, de xeito que dividiremos a interface en tres seccións, ás que se accederá a través de lapelas (o gadget JTabbedPane permitiranos traballar cos 3 contedores de xeito separado). Estas seccións son:

Modelo:
conterá unha táboa na que cada columna representará un dos atributos, e cada fila unha instancia que pode conter valores para cada un desos atributos. As instancias á sua esquerda unha columna adicional e non editable que indique o índice da instancia. Á dereita da táboa reservaremos un espacio para arroxar información sobre un atributo ao seleccionalo. Por exemplo, ao seleccionar un atributo de tipo numérico (pulsando na cabeceira da columna que o representa) indicarase neste espacio á dereita a media de valores, o máximo, o mínimo, a desviación típica, etc. O mockup desta sección pódese observar na figura 4.11.

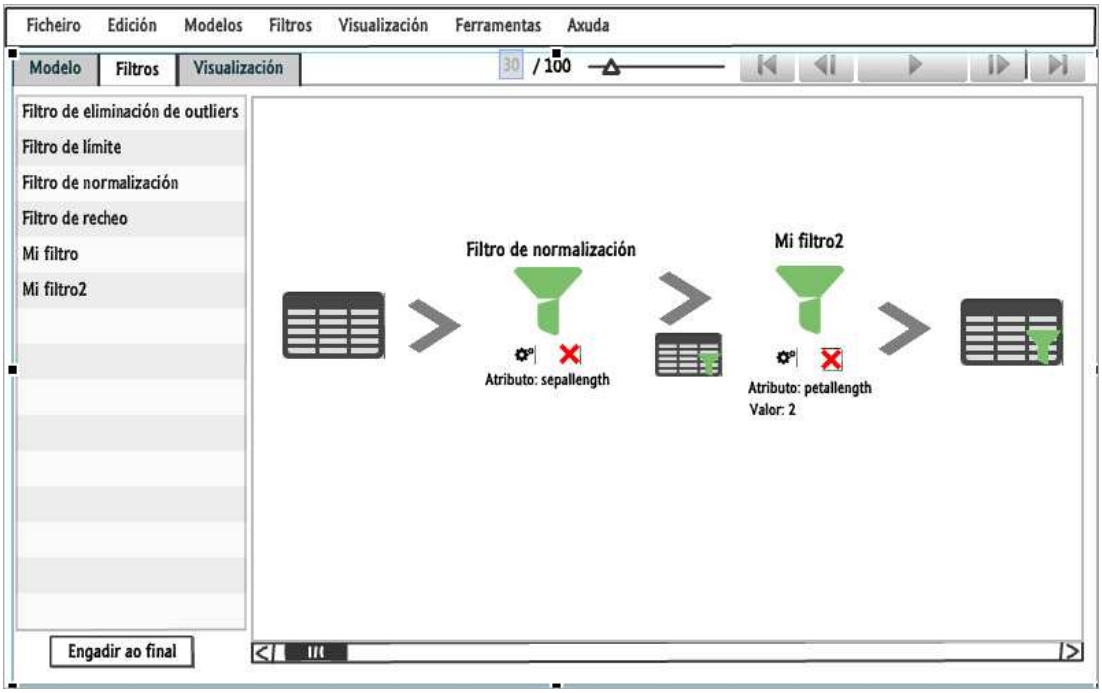


Figura 4.12: Mockup da sección Filtros

Filtros:
conterá unha lista de filtros dispoñibles pegada á borde esquerda, e un botón baixo ela que se active cando haxa un filtro desta lista seleccionado, para engadilo. O resto do contedor ocuparao unha secuencia de iconas en fila, facendo referencia aos distintos filtros que se engadiron e aos modelos parciais que temos antes e despois de cada filtro. O mockup desta sección pódese observar na figura 4.12.

Visualización:
a totalidade do espacio para esta sección ocuparao un conxunto de diagramas de dispersión organizados baixo unha matriz, de xeito que en cada fila da matriz os diagramas teñan o mesmo atributo para as ordenadas, e en cada columna da matriz os diagramas teñan o mesmo atributo para as abscisas. Cada diagrama disporá dun botón para ser ampliado nunha ventá aparte. Tamén figurarán dentro desta sección aínda que fora do propio contedor (en liña coas lapelas das seccións) 5 botóns asociados a funcións de reprodución (ir a principio, paso atrás, reproducir/pausar, paso adiante e ir ao final), así como un pivote desprazable ao longo dunha barra (moverase conforme se reproduza o experimento) e un indicador do número de elementos xa visualizados e totais. O mockup desta sección pódese observar na figura 4.13.

Cada sección foi previamente deseñada a nivel gráfico por medio da ferra-

Figura 4.13: Mockup da sección Visualizacion

menta Lumzy, da cal acabamos de presentar algunhas capturas de pantalla. Para acceder ao mockup e ter unha mínima interacción con el (navegando a través das seccións), podemos utilizar o seguinte enlace (probado a día 10/07/2015): http://lumzy.com/access/?id=83517845D38CCEEA54A9D6B484A32147

Dada a necesidade do espacio para táboas, listas e sobre todo para a matriz de diagramas de dispersión, adoptouse a medida de desplazar os activadores de case todas as funcionalidades á barra de menús (non operativa no mockup). Prácticamente os únicos botóns que por motivos de usabilidade non podían ser desprazados cara a barra de menús eran os de funcións de reprodución, e de feito tiveron que ir encaixados fóra da propia sección de Visualización sobre a que operan. Entre outros, na barra de menús haberá un título para cada unha das 3 seccións.

Os ítems da barra de menús son os seguintes:

Ficheiro:

Importar ficheiro:
Abre unha ventá cun explorador de ficheiros para seleccionar un arquivo en formato .arff ou .csv. A dirección a ese arquivo mándase ao Controlador como argumento, acompañando a un evento de tipo IMPORTAR_FICHEIRO.

Exportar ficheiro:
Abre unha ventá cun explorador de ficheiros para seleccionar unha ruta e un nome de arquivo en formato .arff ou .csv. A dirección a ese novo arquivo mándase ao Controlador como argumento, acompañando a un evento de tipo EXPORTAR_FICHEIRO.

Abrir sesión:
Abre unha ventá cun explorador de ficheiros para seleccionar un arquivo en formato .jdms. A dirección a ese arquivo mándase ao Controlador como argumento, acompañando a un evento de tipo ABRIR_SESION.

Gardar sesión:
Abre unha ventá cun explorador de ficheiros para seleccionar unha ruta e un nome de arquivo en formato .jdms. A dirección a ese novo arquivo mándase ao Controlador como argumento, acompañando a un evento de tipo GARDAR_SESION.

Restaurar:
Envía ao Controlador un evento de tipo RESTAURAR.

Pechar:
Pecha a aplicación JDataMotion.

Edición:

Desfacer:
Envía ao Controlador un evento de tipo DESFACER.

Rafacer:
Envía ao Controlador un evento de tipo REFACER.

Modelo:

Engadir instancia:
Envía ao Controlador un evento de tipo ENGADIR_DATOS.

Eliminar instancias seleccionadas:
Envía ao Controlador un evento de tipo ELIMINAR_DATOS, acompañado dun array de enteiros que contén os índices das filas do Modelo que están seleccionadas.

Engadir atributo:
Envía ao Controlador un evento de tipo ENGADIR_ATRIBUTO.

Eliminar atributo:
Envía ao Controlador un evento de tipo ELIMINAR_ATRIBUTO, acompañado do índice do atributo sobre o que se pulsou.

Renomear atributo:
Abre un diálogo para introducir o novo nome do atributo pulsado. Envía ao Controlador un evento de tipo RENOMEAR_ATRIBUTO, acompañado dun array de obxetos co índice do atributo sobre o que se pulsou e o novo nome do atributo.

Mudar nome da relación:
Abre un diálogo para introducir o novo nome para a relación. Envía ao Controlador un evento de tipo MUDAR_NOME_RELACION, acompañado do novo nome.

Amosar todas as columnas:
Volve a amosar todas as columnas que se ocultaron.

Filtros:

Importar filtros:
Abre unha ventá cun explorador de ficheiros para seleccionar un arquivo en formato .jdmf. A dirección a ese arquivo mándase ao Controlador como argumento, acompañando a un evento de tipo IMPORTAR_FILTROS.

Exportar filtros seleccionados:
Abre unha ventá cun explorador de ficheiros para seleccionar unha ruta e un nome de arquivo en formato .jdmf. Envía ao Controlador un evento de tipo EXPORTAR_FILTROS, acompañado dun array de obxetos coa dirección do arquivo e un array de enteiros cos índices dos filtros seleccionados.

Importar filtro dende JAR:
Abre unha ventá cun explorador de ficheiros para seleccionar un arquivo en formato .jar. A dirección a ese arquivo mándase ao Controlador como argumento, acompañando a un evento de tipo IMPORTAR_FILTRO_DENDE_JAR.

Visualización:

Engadir ou eliminar scatterplots:
Abre unha ventá cunha matriz que enfronta a todos os atributos entre si. Os diagramas de dispersión asociados que se seleccionen ou deseleccionen aparecerán ou desaparecerán da lapela de Visualización.

Establecer atributo nominal representado:
Abre un diálogo para seleccionar dunha lista despregable de atributos

nominais cal se desexa utilizar para diferenciar os puntos nos diagramas de dispersión.

Configurar reprodutor:
Abre un diálogo que permite modificar a orde da reprodución, o paso e a cor e a lonxitude da estela.

Calcular distancia:
Abre un diálogo que permite seleccionar dous puntos da lapela de Visualización para calcular a distancia entre eles, segundo unha fórmula editable incluída no diálogo.

Ferramentas:

Idioma:

Galego:
Cambia o idioma da aplicación a galego (require reiniciar).

Español:
Cambia o idioma da aplicación a español (require reiniciar).

Inglés:
Cambia o idioma da aplicación a inglés (require reiniciar).

Axuda:

Acerca de:
Abre un diálogo con información acerca da aplicación e o seu desenvolvemento.

E a continuación comentaremos as funcionalidades que sí se incorporaron na lapela correspondente ao seu ámbito de actuación:

Modelo:

Botón secundario nun atributo → Tipo:

Numérico:
Envía ao Controlador un evento de tipo MUDAR_TIPO, acompañado dun array de obxectos co índice do atributo sobre o que se pulsou e a constante Attribute.NUMERIC.

Nominal:
Envía ao Controlador un evento de tipo MUDAR_TIPO, acom-

pañado dun array de obxectos co índice do atributo sobre o que se pulsou e a constante Attribute.NOMINAL.

String:
Envía ao Controlador un evento de tipo MUDAR_TIPO, acompañado dun array de obxectos co índice do atributo sobre o que se pulsou e a constante Attribute.STRING.

Data:
Envía ao Controlador un evento de tipo MUDAR_TIPO, acompañado dun array de obxectos co índice do atributo sobre o que se pulsou e a constante Attribute.DATA.

Botón secundario nun atributo → Índice temporal:
Envía ao Controlador un evento de tipo MUDAR_INDICE_TEMPORAL, acompañado do índice do atributo sobre o que se pulsou.

Botón secundario nun atributo → Agochar columna:
Agocha a columna da táboa correspondente ao atributo no que se pulsou.

Botón secundario nun histograma:
Abre un menú contextual no que se pode almacenar o histograma como unha imaxe, copialo, imprimilo ou cambiarlle o zoom ou a escala.

Selección por arrastre dunha área dentro do diagrama de dispersión:
Reposiciona e escala o histograma para visualizar a área marcada.

Tecla control + arrastre do diagrama de dispersión:
Reposiciona o histograma movendo o lenzo na dirección do cursor.

Roda do rato:
Acerca ou afasta o zoom do histograma.

Filtros:

Botón "Engadir ao final":
Envía ao Controlador un evento de tipo ENGADIR_FILTRO, acompañado dun array de obxectos co índice que representa a última posición e o filtro que se seleccionou da lista.

Botóns de instancias parciais/finais:
Abre unha nova ventá co modelo do experimento ao aplicar os filtros ata ese punto da secuencia.

Botón mover filtro á dereita:
Envía ao Controlador un evento de tipo INTERCAMBIAR_FILTROS,

acompañado dun array de obxectos co índice do filtro sobre o que se pulsou este botón e o índice seguinte.

Botón mover filtro á esquerda:
Envía ao Controlador un evento de tipo INTERCAMBIAR_FILTROS, acompañado dun array de obxectos co índice do filtro sobre o que se pulsou este botón e o índice anterior.

Botón configurar filtro:
Abre un diálogo con campos para cubrir os valores dos parámetros do filtro (o atributo sobre o que se aplica vai incluído). Envía ao Controlador un evento de tipo CONFIGURAR_FILTRO, acompañado dun array de obxectos co índice do filtro e o array con estes novos valores.

Botón eliminar filtro:
Envía ao Controlador un evento de tipo ELIMINAR_FILTRO, acompañado do índice do filtro sobre o que se pulsou este botón.

Visualización:

Botón "Comezar visualización":
Mesmo efecto ca Visualización → Engadir ou eliminar scatterplots

Desprazador:
Permite moverse a distintos puntos da reprodución, chamando ao método goTo() do ManexadorScatterplots e pasándolle a fracción de tempo na que se situou o pivote do desprazador.

Botón "Ir ao comezo":
Leva á reprodución ao instante inicial, chamando ao método goTo() do ManexadorScatterplots e pasándolle o valor 0.0.

Botón "Paso atrás":
Retrocede un paso no reprodución, chamando ao método goToPrevious() do ManexadorScatterplots.

Botón "Reproducir/Pausar":
Continúa a reprodución no último punto si esta se atopa parada ou pausada, chamando ao método play() do ManexadorScatterplots. Pausa a reprodución si esta está sucedendo, chamando ao método pause() do ManexadorScatterplots.

Botón "Paso adiante":
Retrocede un paso no reprodución, chamando ao método gotoNext() do ManexadorScatterplots.

Botón "Ir ao final":
Leva á reprodución ao instante inicial, chamando ao método goTo() do ManexadorScatterplots e pasándolle o valor 1.0.

Botón secundario nun diagrama de dispersión:
Abre un menú contextual no que se pode almacenar a imaxe, copiala, imprimila, cambiarlle o zoom ou automatizar a escala durante a reprodución.

Botón secundario nun diagrama de dispersión → Propiedades:
Este ítem do menú contextual abre unha ventá con opcións de configuración gráfica (colores, fontes, formas, etc.) para aplicar aos diagramas de dispersión desta sección de xeito global. As preferencias establecidas neste menú gardaranse como configuración de usuario.

Selección por arrastre dunha área dentro do diagrama de dispersión:
Reposiciona e escala o diagrama para visualizar a área marcada.

Tecla control + arrastre do diagrama de dispersión:
Reposiciona o diagrama movendo o lenzo na dirección do cursor.

Roda do rato:
Acerca ou afasta o zoom o diagrama de dispersión sobre o que repousa o cursor.

4.2.4. Clase JDataMotion

É a clase que contén o método estático main() da aplicación JDataMotion. Este método é o que se procesa en canto executamos o programa. O seu contido basease unicamente na instanciación dun Modelo, seguido da instanciación dunha Vista. Por último, invoca ao método inicializar(Modelo modelo, boolean visualizar) da instancia de Vista que se creou, pasándolle o modelo como primeiro parámetro e un valor true como segundo.

A responsabilidade do main() remata cando xunta unha vista cun modelo por medio do procedemento inicializar(Modelo modelo, boolean visualizar). Este proceso encárgase de que a vista rexistre a instancia de Modelo que acaba de recibir, e invoca no modelo ao método público addObserver(Observer o), pasándose a vista a sí mesma como parámetro. Como vimos anteriormente, o Modelo estende a clase Observable e por tanto herda os seus métodos. Un destes métodos, o addObserver, permite que as instancias de calquera clase que implemente a interface Observer poidan engadirse como observadoras.

O método inicializar da vista tamén ten a responsabilidade de instanciar un

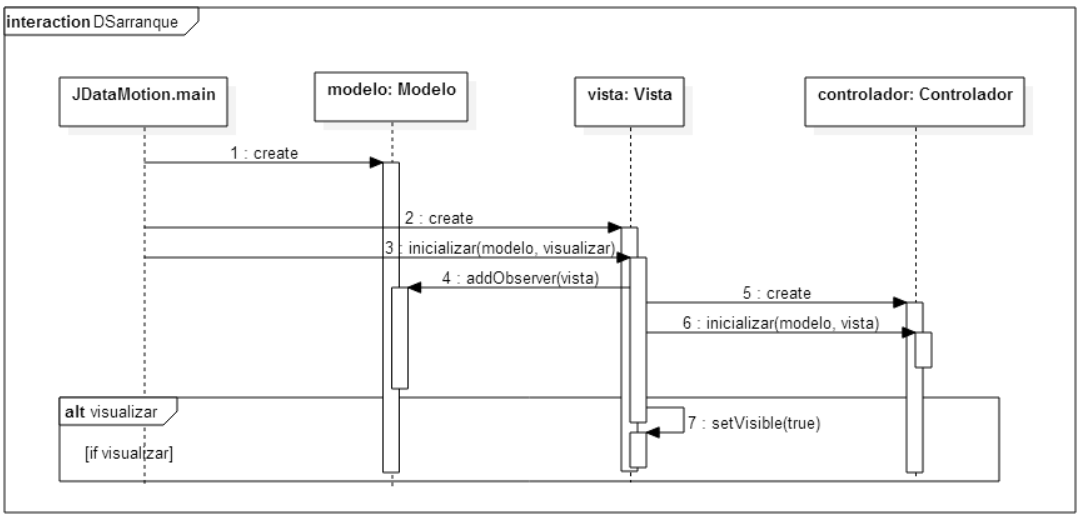


Figura 4.14: Diagrama de secuencia do arranque

controlador para interactuar co Modelo e que interceda polos dous en transaccións como a xestión e notificación de comandos. Do mesmo xeito, a vista invocará ao método inicializar(Modelo modelo, Vista vista) do controlador, pasando o modelo que recibiu e rexistrou de primeiro argumento, e a si mesma como segundo argumento. O método inicializar que ten o controlador unicamente rexistra as dúas instancias que recibiu, e polas que terá que interceder en varios procesos. Ao rematar, segue a execución do método inicializar da vista, ao cal só lle queda arrancar a interface gráfica se o parámetro visualizar é igual a true. Isto pode non ser de interese no caso da execución de tests de proba, nos que non se desexa interactuar coa nova ventá, se non executar unicamente métodos e comprobar o seu resultado.

Podemos observar mellor o proceso de arranque da aplicación a través da figura 4.14, que contén un diagrama de secuencia con todas as instanciacións e invocacións.

4.3. Deseño de JDataMotion.common

Este subproxecto contén todo o material que resulta útil tanto para a aplicación JDataMotion coma para outros subproxectos que conteñan filtros válidos para seren importados. O diagrama de clases co seu contido amósase na figura 4.15.

Ademais das clases da figura, o proxecto tamén inclúe as librerías "weka-dev-3.7.10.jar", "weka-dev-3.7.10-javadoc.jar" e "weka-dev-3.7.10-sources.jar", que

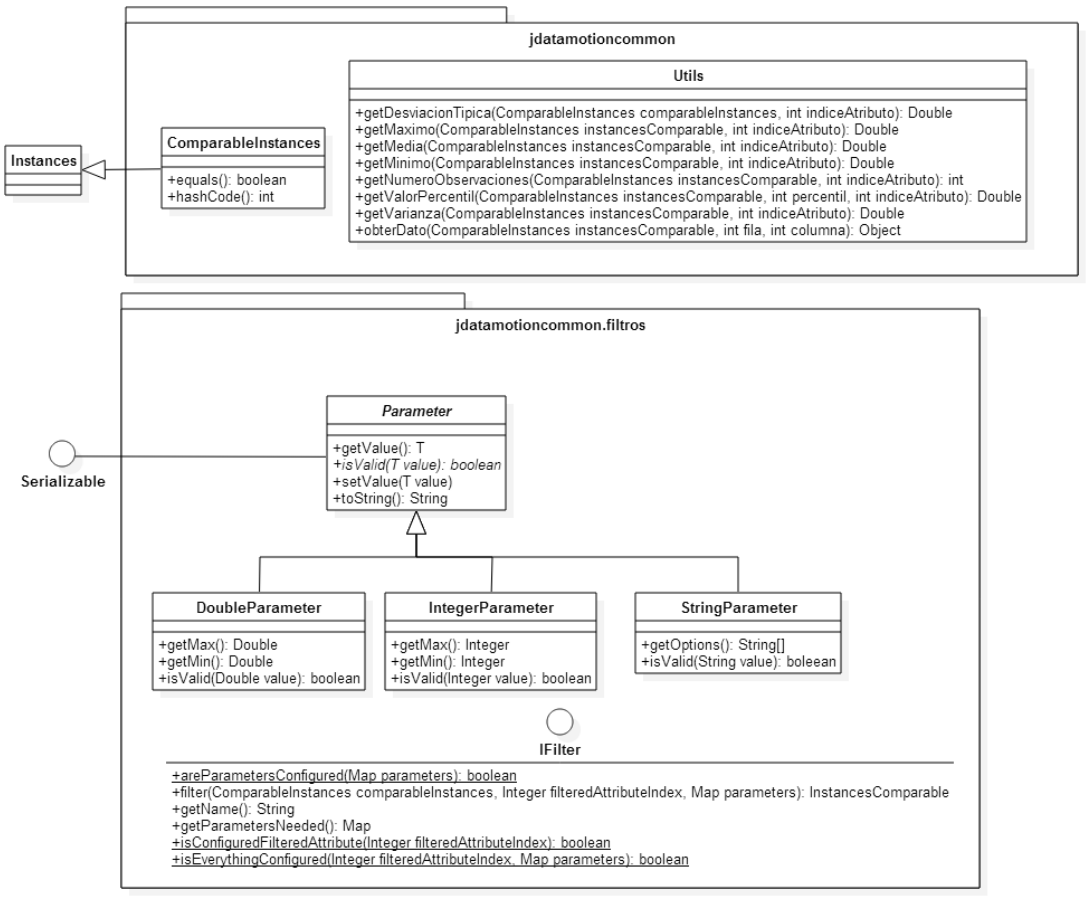


Figura 4.15: Diagrama de clases de JDataMotion.common

conteñen respectivamente a interface de programación, a documentación e os códigos fonte do proyecto Weka na súa versión 3.7.10.

O paquete jdatamotioncommon contén a extensión das Instances de Weka que vimos mencionando todo este tempo. Só engaden unha implementación da clase equals(Object o) que nos permite determinar, a partir dos atributos e instancias da clase, que dúas Instances son iguais. Isto axudaranos á hora de realizar os test de proba, comprobando que un obxecto de tipo Instances tras executar un método que o modifica equivale a outro obxecto Instances que nós esperamos.

A clase Utils contén unha batería de métodos estáticos de interese estatístico que resultan moi útiles á hora de programar novos filtros. Case todos reciben como mínimo unha instancia de InstancesComparable e un atributo do que deben extraer un dato (máximo, mínimo, media, varianza, desviación típica, percentil ou o número de observacións).

O paquete jdatamotioncommon.filters contén en primeiro lugar a interface IFilter, que toda clase susceptible de ser importada debe implementar. Os seus métodos descríbense a continuación:

areParametersConfigured(Map<String, Parameter> parameters):
Método estático que comproba que os parámetros que necesita o filtro xa teñan un valor asignado.

filter(ComparableInstances comparableInstances, Integer filteredAttributeIndex, Map<String, Parameter> parameters):
Contén toda a lóxica do filtro. Conta coas comparableInstances que debe filtrar e co índice do atributo sobre o que debe actuar, ademais dos valores dos parámetros que o filtro demandou (o parámetro Map asocia o nome de cada parámetro co parámetro en si). A partir destes inputs, a súa execución devolver unha instancia de InstancesComparable, que consideraremos como filtrada.

getName():
Devolve o nome do filtro

getParametersNeeded():
Devolve un Map<String, Parameter> que conteña os parámetros (baleiros) que o filtro necesita ter cubertos para funcionar, utilizando coma clave para acceder a cada filtro o nome que se lle queira dar. Este nome debe almacenarse no filtro para logo recuperar o parámetro cuberto da estrutura Map.

isConfiguredFilteredAttribute(Integer filteredAttributeIndex):
Método estático que comproba que o atributo sobre o que ten que actuar o filtro está definido.

isEverythingConfigured(Integer filteredAttributeIndex, Map<String, Parameter> parameters)
Comproba que tanto os parámetros teñan un valor asignado como que o índice no que debe actuar o filtro esté definido. Normalmente cando todo isto se cumpra, o filtro poderá ser aplicado.

De todos estes métodos, os únicos que deben ser implementados son filter, getName e getParametersNeeded. O resto son métodos estáticos de utilidade para implementadores e para a propia aplicación.

O paquete jdatamotioncommon.filters tamén contén unha xerarquía de parámetros, que serán os que participen das implementacións de IFilter. A clase Parameter serve para almacenar un valor dun certo tipo T, que é o valor do parámetro. Este valor pode ter unha restrición para ser válido, por iso o método isValid(T value) é abstracto.

As clases que implementarán a Parameter van ser:

DoubleParameter:
É un parámetro que almacena un valor de tipo Double. O seu construtor admite un valor double máximo e outro mínimo para utilizalos na implementación do método isValid(Double value).

IntegerParameter:
É un parámetro que almacena un valor de tipo Integer. O seu construtor admite un valor double máximo e outro mínimo para utilizalos na implementación do método isValid(Integer value).

StringParameter:
É un parámetro que almacena un valor de tipo String. O seu construtor admite un valor de tipo array de String con opcións válidas para utilizalas na implementación do método isValid(String value).

Estes parámetros son axeitadamente interpretados polo JDataMotion, de forma que á hora de configurar un filtro, para un DoubleParameter ou IntegerParameter apareza unha entrada de formulario que deba ser Double ou Integer e comprendida entre os límites (se os ten). Para os StringParameter, si se especifica un array de String, aparecerá un JComboBox (lista despregable de opcións de Swing).

4.4. Deseño de JDataMotion.filters.sample

Este subproxecto constitúe un exemplo e unha proba da funcionalidade da librería común. O seu deseño, bastante simple, aparece na figura 4.16.

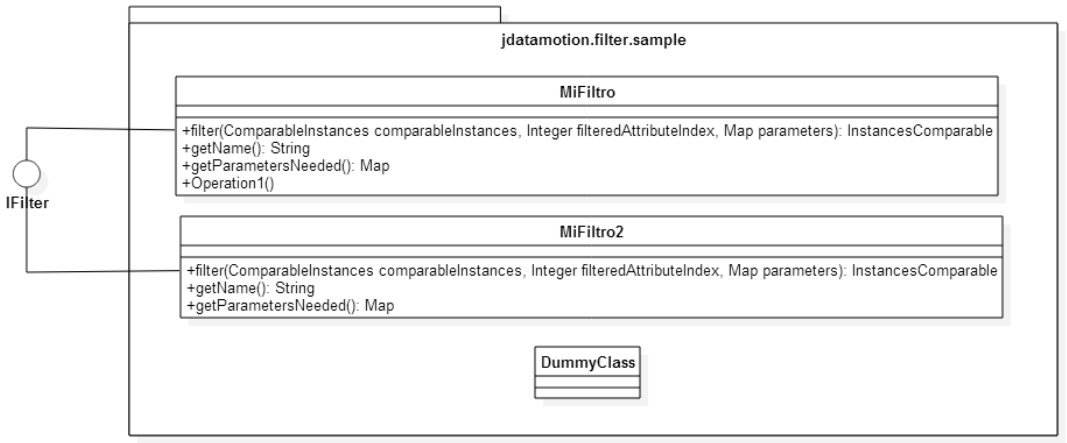


Figura 4.16: Diagrama de clases de JDataMotion.filters.sample

É un proxecto constituido por 3 clases:

MiFiltro:
É unha clase que implementa IFilter. Ten un parámetro DoubleParameter de nome "VALOR", chámase "Mi filtro" e ao filtrar, se todo está correctamente configurado, devolve as instancias do atributo que ten que filtrar multiplicadas por -1 e polo valor do DoubleParameter. Espérase que a importación do .jar admita esta clase.

MiFiltro2:
É unha clase que implementa IFilter. Ten un parámetro DoubleParameter de nome "VALOR", chámase "Mi filtro" e ao filtrar, se todo está correctamente configurado, devolve as instancias do atributo que ten que filtrar multiplicadas por un número aleatorio maior ca 0.0 e menor ca 1.0. Espérase que a importación do .jar admita esta clase.

DummyClass:
É unha clase que non implementa IFilter. Espérase que ao importar o arquivo .jar, esta clase sexa descartada.

Imos a ver e comentar a implementación da clase MiFiltro:

```
import java.util.HashMap; //importamos algunhas das clases que facilita
 Java
import java.util.Iterator;
import java.util.Map;
import jdatamotioncommon.ComparableInstances; // importamos a clase
ComparableInstances do paquete jdatamotioncommon
import jdatamotioncommon.filtros.DoubleParameter; // importamos a clase
 DoubleParameter do paquete jdatamotioncommon.filtros
```

```
import jdatamotioncommon.filtros.IFilter; // importamos a interface
IFilter do paquete jdatamotioncommon.filtros
import jdatamotioncommon.filtros.Parameter; // importamos a clase
ComparableInstances do paquete jdatamotioncommon.filtros
import weka.core.Instance; // importamos a clase Instance de weka.core

/**
 *
 * @author usuario
 */
public class MiFiltro implements IFilter { // MiFiltro implementa a
interface IFilter

    private static final String VALOR = "valor"; // nome do parametro
    que imos utilizar, debemos gardalo e mantelo (modificador final)

      // sobrescribimos o metodo filter
    @Override
    public ComparableInstances filter(ComparableInstances
    comparableInstances, Integer filteredAttributeIndex, Map<String,
    Parameter> parameters) {
            // se falta algun parametro por cubrir ou non se defineu o
            atributo sobre o que se vai actuar, devolvemos as instancias
             recibidas e o filtro non ten aplicacion (recomendado)
        if (!IFilter.isEverythingConfigured(filteredAttributeIndex,
        parameters) || !comparableInstances.attribute(
        filteredAttributeIndex).isNumeric()) {
            return comparableInstances;
        }
            // iteramos ao longo das instancias que recibimos
        Iterator<Instance> it = comparableInstances.iterator();
        while (it.hasNext()) {
            Instance instance = it.next();
                 // se a instancia non ten un valor definido no atributo
                  que temos que filtrar, v = null, en caso contrario v
                 toma ese valor
            Double v = instance.isMissing(filteredAttributeIndex) ? null
             : instance.value(filteredAttributeIndex);
                 // establecemos para ese atributo da instancia un valor
                  igual ao que obtivemos (v) multiplicado por -1 e polo
                 valor que se aplicou no parametro
            instance.setValue(filteredAttributeIndex, -(double)
            parameters.get(VALOR).getValue() * v);
        }
            // devolvemos as instancias filtradas
```

```
        return comparableInstances;
    }

      // sobrescribimos o metodo getParametersNeeded
    @Override
    public Map<String, Parameter> getParametersNeeded() {
            // creamos un mapa que relacione String con Parameter
        Map<String, Parameter> p = new HashMap<>();
            // colocamos para o nome do parametro que necesitamos unha
            instancia sen parametros de DoubleParameter (queremos un
            parametro de tipo double, sen maximo nin minimo)
        p.put(VALOR, new DoubleParameter());
            // devolvemos o mapa
        return p;
    }

      // sobrescribimos o metodo getName para darlle nome ao filtro
    @Override
    public String getName() {
        return "Mi filtro";
    }

}
```

Capítulo 5

Validación e probas

Neste capítulo realizaremos o deseño das probas da nosa aplicación. As probas son experimentos cunha especificación determinada. Céntranse en probar un aspecto puntual do sistema, parten dunhas premisas (parámetros, situación, contexto, etc.) definidas a priori, e espérase delas unha resposta consistente. A correspondencia entre o estado ou resposta que se espera da proba e o resultado da súa realización determinan a validez do aspecto que están a probar.

Para probar a nosa aplicación recorreremos a dous métodos, cada un centrarase nun escenario concreto. Por unha parte, a libraría JUnit pon á nosa disposición unha serie de clases, deseñadas para lanzar probas contra pequenos métodos ou bloques de execución. Esta solución será de gran utilidade para validar as funcionalidades que versan sobre o Modelo, xa que o Modelo manipula grandes cantidades de datos que de xeito programático se poden comprobar facilmente.

Os tests en JUnit adoitan finalizar cunha sentencia assetEquals(), que recibe dous obxectos e comproba que sexa iguais para aceptar a proba. En caso contrario devolven un mensaxe de erro e a traza para axudar a solventalo. Para poder asumir que as probas funcionan, necesitamos algo co que comparar os nosos métodos. Nalgúns casos, imos botar man das prestacións da libraría Weka para verificar que o estado final dunha estrutura de datos é o esperado. Deste xeito, poderemos defender que o método funciona apoiándose en que os métodos de Weka co que o validamos xa foron probados previamente.

En base a isto, comezaremos os nosos test creando dous métodos estáticos nunha clase chamada "ValidFileLoading.java". Estos métodos son loadARFF(String resourcePath) e loadCSV(String resourcePath). Ambos serán usados de acordo á documentación sobre o seu uso para ler un ficheiro en formato ARFF ou CSV (respectivamente) e extraer del unha instancia de ComparableInstances. Deste xeito xa poderemos asumir como válido que as instancesComparable que devolven son as correctas, as que contén o ficheiro do cal lle pasamos a dirección.

```
public class ValidFileLoading {

    public static ComparableInstances loadARFF(String resourcePath) {
        ComparableInstances comparableInstances = null;
        try {
            ArffLoader loaderARFF = new ArffLoader();
            loaderARFF.setFile(new File(resourcePath));
            comparableInstances = new ComparableInstances(loaderARFF.
            getDataSet());
        } catch (IOException ex) {
            Logger.getLogger(ValidFileLoading.class.getName()).log(Level
            .SEVERE, null, ex);
        }
        return comparableInstances;
    }

    public static ComparableInstances loadCSV(String resourcePath) {
        ComparableInstances comparableInstances = null;
        try {
            CSVLoader loaderCSV = new CSVLoader();
            loaderCSV.setSource(new File(resourcePath));
            comparableInstances = new ComparableInstances(loaderCSV.
            getDataSet());

        } catch (IOException ex) {
            Logger.getLogger(ValidFileLoading.class.getName()).log(Level
            .SEVERE, null, ex);
        }
        return comparableInstances;
    }

}
```

Tamén incluiremos no directorio dos filtros dous ficheiros compatibles con JDataMotion, que empregaremos para realizar as probas: "213_N3.csv" (40 atributos e 3251 instancias) e "example01.arff" (1 atributo nominal e 30 numéricos, e 569 instancias)

Sen embargo, JUnit non nos facilita do mesmo xeito a validación de métodos que implican cambios directos na interface de usuario. Probas a nivel gráfico como comprobar que se visualizan correctamente os diagramas de dispersión son menos asequibles realizándoos mediante código ca por simple observación. É por isto que adoptaremos a avaliación heurística para aqueles casos nos que a usabilidade (apariencia intuitiva, sencillez, facilidade de uso, etc.) sexa un importante factor

a validar.

A continuación percorreremos de novo todos os requisitos que describimos na análise, para asignarlle un test de proba e executalo mostrando os resultados.

5.0.1. Requisitos funcionais

RF01

Título
Importar arquivos con datos para o experimento

Descrición
A aplicación debe permitir cargar do sistema de arquivos un ficheiro que conteña unha secuencia de datos (nun formato axeitado segundo o RNF01) para ser utilizados no experimento.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF01_1 e TestRF01_2

Código fonte
TestRF01_1:

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            vista.getControlador().manexarEvento(Controlador.
            IMPORTAR_FICHEIRO, pathEntrada);
            ComparableInstances is2 = modelo.getComparableInstances
            ();
            assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
```

```
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

TestRF01_2:

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "213_N3.csv";
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadCSV(
            pathEntrada);
            vista.getControlador().manexarEvento(Controlador.
            IMPORTAR_FICHEIRO, pathEntrada);
            ComparableInstances is2 = modelo.getComparableInstances
            ();
            assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

Descrición

Instáncianse un Modelo e unha vista, inicialízanse utilizando o parámetro visualizacion=false para bloquear a aparición da interface gráfica e actívase a depuración. Importamos o recurso "example01.arff" para TestRF01_1 ou "213_N3.csv" para TestRF01_2, con axuda dos métodos de Weka nos que imos confiar, e almacenamos as instancias obtidas en "is1". A continuación lanzamos ao Controlador un evento de tipo IMPORTAR_FICHEIRO e almacenamos as instancias do modelo en "is2". Por último, metemos "is1" e "is2" como parámetros do método assertEquals(Object expected, Object actual), e si ambos son iguais o RF01 quedará validado para formatos CSV e ARFF. A sobrescritura do método equals(Object o) dentro de ComparableInstances permitiralle ás instancias desta clase seren pasadas ao assertEquals.

Resultado
Correcto.

RF02

Título
Exportar datos

Descrición
A aplicación debe permitir almacenar nun arquivo o conxunto de datos do experimento actual (tendo en conta filtrados, modificacións, datos engadidos ou eliminados...). Os arquivos de saída deberán respectar o RNF01 en canto a formato de almacenamento.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF02_1 e TestRF02_2

Código fonte
TestRF02_1:

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        String tempSaida = "temp01.csv";
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            String pathSaida = new URI(getClass().getResource(".").
            toString()).getPath() + tempSaida;
            vista.getControlador().manexarEvento(Controlador.
            EXPORTAR_FICHEIRO, new Object[]{"csv", pathSaida});
            ComparableInstances is2 = ValidFileLoading.loadCSV(
            pathSaida);
```

```
            is1.setRelationName("");
            is2.setRelationName("");
            assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

TestRF02_2:

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "213_N3.csv";
        String tempSaida = "temp01.arff";
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadCSV(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            String pathSaida = new URI(getClass().getResource(".").
            toString()).getPath() + tempSaida;
            vista.getControlador().manexarEvento(Controlador.
            EXPORTAR_FICHEIRO, new Object[]{"arff", pathSaida});
            ComparableInstances is2 = ValidFileLoading.loadARFF(
            pathSaida);
                  assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

Descrición

Instáncianse un Modelo e unha vista, inicialízanse utilizando o parámetro visualizacion=false para bloquear a aparición da interface gráfica e actívase a depuración. Importamos o recurso "example01.arff" para TestRF02_1 ou "213_N3.csv" para TestRF02_2, con axuda dos métodos de Weka nos que imos confiar, e almacenamos as instancias obtidas en "is1". A continua-

ción lanzamos ao Controlador un evento de tipo EXPORTAR_FICHEIRO, pasándolle a dirección dun arquivo temporal en formato CSV para TestRF02_1 ou en formato ARFF para TestRF02_2. Despois disto, utilizamos as clases de importación nas que estamos confiando para obter as instancias dende o arquivo ao que acabamos de exportar e almacenamos o seu contido en "is2". Por último, metemos "is1" e "is2" como parámetros do método assertEquals(Object expected, Object actual), e si ambos son iguais o RF02 quedará validado para formatos CSV e ARFF. A sobrescritura do método equals(Object o) dentro de ComparableInstances permitiralle ás instancias desta clase seren pasadas ao assertEquals. En TestRF02_1 debemos eliminar os nomes de relación das dúas ComparableInstances para ser xustos, xa que o arquivo ARFF perdeu o seu no proceso de exportación a CSV.

Resultado
Correcto.

RF03

Título
Gardar sesión

Descrición
A aplicación debe permitir gardar en disco a sesión (ou experimento) actual tal e como está no momento de executar esta acción.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF0304

Código fonte

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        String archivoSesion = "temp01.jdms";
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
```

```
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            modelo.setDireccionAoFicheiro(pathEntrada);
            modelo.setHashCodeFicheiro(Modelo.resumirFicheiroSHA1(
            new File(pathEntrada)));
            String pathSaida = new URI(getClass().getResource(".").
            toString()).getPath() + archivoSesion;
            /* se modificamos un dato manualmente, a sesion non o
            rexistra
             e ao restaurar a sesion o arquivo segue sendo igual ao
              inicial */
            modelo.getComparableInstances().instance(5).setValue(5,
             666);
            /* se lanzamos un comando, a sesin rexistrao e entonces
             a proba fallaria
             porque se volveria a executar o comando ao restaurar a
              sesion, e xa non seria igual ao inicial */
            //vista.getControlador().manexarEvento(Controlador.
            MUDAR_DATO, new Object[]{5, 5, 666});
            vista.getControlador().manexarEvento(Controlador.
            GARDAR_SESION, pathSaida);
            vista.getControlador().manexarEvento(Controlador.
            ABRIR_SESION, pathSaida);
            ComparableInstances is2 = modelo.getComparableInstances
            ();
            assertEquals(is1, is2);
        } catch (Exception ex) {
            Logger.getLogger(TestRF0304.class.getName()).log(Level.
            SEVERE, null, ex);
        }
    }
```

Descrición

Tras inicializar o sistema do mesmo xeito ca en test anteriores, cárgase o ficheiro "example01.arff" en "is1" por medio de ValidFileLoading (importación confiable). Asígnase ao modelo unha copia das ComparableInstances obtidas, de xeito que "is1" siga referenciando as instancias orixinais. Tamén se asigna a dirección ao ficheiro e o seu hash, pois o Modelo compróbaos ao abrir unha sesión por motivos de consistencia. A continuación modifícanse directamente as instancias, sen axuda de eventos. Despois lanzamos un evento GARDAR_SESION e outro ABRIR_SESION, ambos coa mesma dirección. Almacenamos as instancias que temos agora en "is2", colocándoas

ás dúas no assertEquals. A modificación manual non debería provocar o error da proba, pois foi creada manualmente e non por un comando, así que ao restaurar a sesión esta modificación non se repetirá.

Resultado
Correcto.

RF04

Título
Abrir sesión

Descrición
A aplicación debe permitir restaurar unha sesión (ou experimento) gardada anteriormente, de xeito que se atope exactamente igual ca no momento en que se gardou.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF0304 (xa mencionado).

RF05

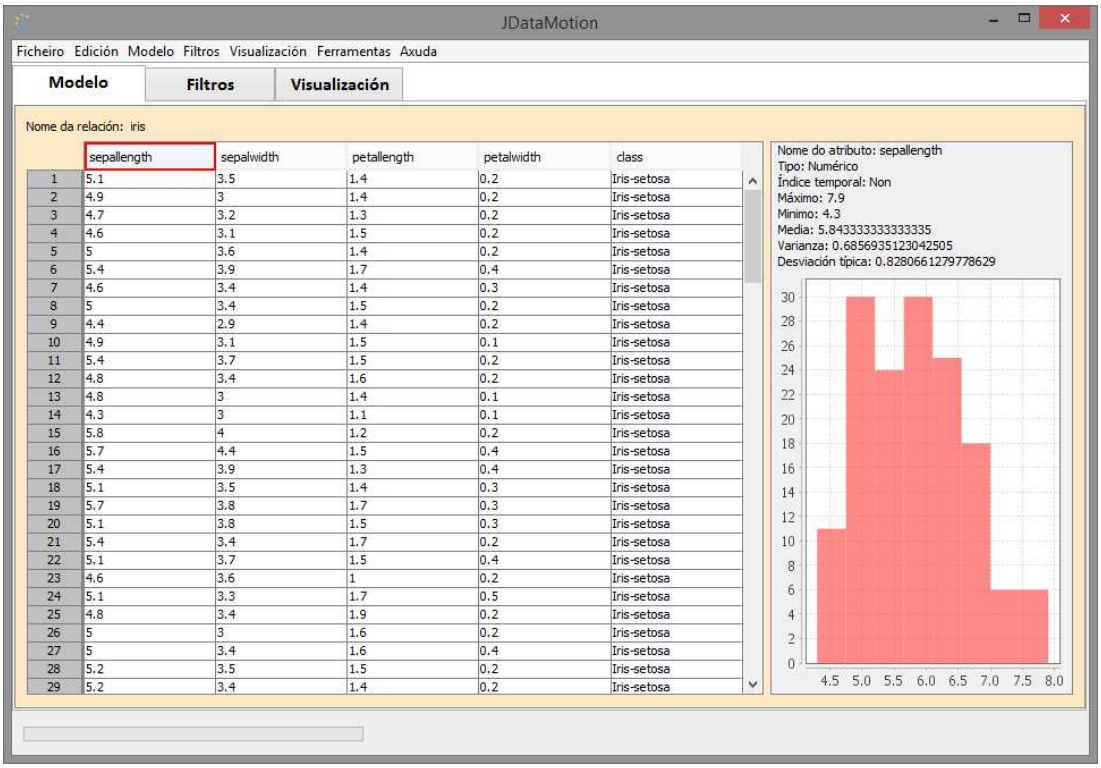
Título
Representar os datos en forma de táboa

Descrición
A aplicación debe ser capaz de amosar os datos segundo unha táboa na que figuren cabeceiras, tipos, valores, etc.

Importancia
Esencial

Tipo de proba
Avaliación heurística

Descrición
Ao abrir un ficheiro de tipo ARFF ou CSV coa aplicación, aparece no menú Modelo toda a información que necesitamos.



	sepallength	sepalwidth	petallength	petalwidth	class
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5	3.6	1.4	0.2	Iris-setosa
6	5.4	3.9	1.7	0.4	Iris-setosa
7	4.6	3.4	1.4	0.3	Iris-setosa
8	5	3.4	1.5	0.2	Iris-setosa
9	4.4	2.9	1.4	0.2	Iris-setosa
10	4.9	3.1	1.5	0.1	Iris-setosa
11	5.4	3.7	1.5	0.2	Iris-setosa
12	4.8	3.4	1.6	0.2	Iris-setosa
13	4.8	3	1.4	0.1	Iris-setosa
14	4.3	3	1.1	0.1	Iris-setosa
15	5.8	4	1.2	0.2	Iris-setosa
16	5.7	4.4	1.5	0.4	Iris-setosa
17	5.4	3.9	1.3	0.4	Iris-setosa
18	5.1	3.5	1.4	0.3	Iris-setosa
19	5.7	3.8	1.7	0.3	Iris-setosa
20	5.1	3.8	1.5	0.3	Iris-setosa
21	5.4	3.4	1.7	0.2	Iris-setosa
22	5.1	3.7	1.5	0.4	Iris-setosa
23	4.6	3.6	1	0.2	Iris-setosa
24	5.1	3.3	1.7	0.5	Iris-setosa
25	4.8	3.4	1.9	0.2	Iris-setosa
26	5	3	1.6	0.2	Iris-setosa
27	5	3.4	1.6	0.4	Iris-setosa
28	5.2	3.5	1.5	0.2	Iris-setosa
29	5.2	3.4	1.4	0.2	Iris-setosa

Figura 5.1: Comprobación heurística do RF05

Resultado
Correcto.

RF06

Título
Insertar datos no experimento actual

Descrición
A aplicación debe permitir a inserción dinámica de datos no experimento actual.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF06_1 e TestRF06_2

Código fonte
TestRF06_1:

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            vista.getControlador().manexarEvento(Controlador.
            ENGADIR_DATOS, null);
            is1.add(new DenseInstance(is1.numAttributes()));
            ComparableInstances is2 = modelo.getComparableInstances
            ();
            assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
```

```
        }
    }
```

TestRF06_2:

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            vista.getControlador().manexarEvento(Controlador.
            ENGADIR_ATRIBUTO, null);
            is1.insertAttributeAt(new Attribute("novoAtributo1", (
            List<String>) null), is1.numAttributes());
            ComparableInstances is2 = modelo.getComparableInstances
            ();
            assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

Descrición

TestRF06_1 debe probar a funcionalidade de engadir datos (filas), mentres que TestRF06_2 debe probar a inserción de atributos. En calquera caso, empézase asignándolle a "is1" as ComparableInstances de ValidFileLoading, e pasándolle unha copia ao Modelo. En TestRF06_1, "is1" engadirá unha instancia baleira de DenseInstance ao final das InstanceComparables, mentres que "is2" asignarase tras lanzar un evento de tipo ENGADIR_DATOS ao Controlador. En TestRF06_2, "is1" engadirá un atributo de tipo String ás InstanceComparables, mentres que "is2" asignarase tras lanzar un evento de tipo ENGADIR_ATRIBUTO. Unha vez máis, cada par de instancias é comparado por assertEquals.

Resultado
Correcto.

RF07

Título
Modificar datos no experimento actual

Descrición
A aplicación debe permitir a modificación dinámica de datos no experimento actual.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF07

Código fonte

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        int fila = 5;
        int columna = 6;
        int novoDato = 666;
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            vista.getControlador().manexarEvento(Controlador.
            MUDAR_DATO, new Object[]{fila, columna, novoDato});
            is1.instance(fila).setValue(columna, novoDato);
            ComparableInstances is2 = modelo.getComparableInstances
            ();
            assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
```

```
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

Descrición

De forma análoga aos tests anteriores, esta proba compara o funcionamento do evento MUDAR_DATO coa modificación programática dos datos, é dicir, utilizando o método setValue propio da clase Instance, a cal desenvolveu Weka.

Resultado

Correcto.

RF08

Título

Eliminar datos no experimento actual

Descrición

A aplicación debe permitir a eliminación dinámica de datos no experimento actual.

Importancia

Esencial

Tipo de proba

Test implementado en JUnit.

Nome do test

TestRF08_1 e TestRF08_2

Código fonte

TestRF08_1:

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        Integer indices[] = new Integer[]{2, 4};
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
```

```
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            vista.getControlador().manexarEvento(Controlador.
            ELIMINAR_DATOS, indices);
                 Arrays.sort(indices);
            for (int i = 0; i < indices.length; i++) {
                is1.remove((int) indices[i] - i);
            }
            ComparableInstances is2 = modelo.getComparableInstances
            ();
            assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

TestRF08_2:

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        int columna = 6;
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            vista.getControlador().manexarEvento(Controlador.
            ELIMINAR_ATRIBUTO, columna);
            is1.deleteAttributeAt(columna);
            ComparableInstances is2 = modelo.getComparableInstances
            ();
            assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
```

```
}
```

Descrición

TestRF08_1 debe probar a funcionalidade de eliminar instancias (filas), mentres que TestRF08_2 debe probar a eliminación de atributos (columnas). O primeiro compara a eliminación dun conxunto de índices por medio de ELIMINAR_DATOS, coa eliminación en bucle a través do método remove, de Instances. O segundo compara a eliminación dun atributo por medio do evento ELIMINAR$_A TRIBUTO$, $ea eliminación de seatributo por medio do método de leteAttributeAt, de Ins$

RF09

Resultado

Título

Asignar tipos aos atributos dun arquivo importado

Descrición

A aplicación debe permitir especificar os tipos de atributos presentes no arquivo importado. Por exemplo, os datos cuantitativos poderían ser enteiros ou reais, mentres que os cualitativos serían algo distinto (mesmamente strings).

Importancia

Esencial

Tipo de proba

Test implementado en JUnit.

Nome do test

TestRF09

Código fonte

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "213_N3.csv";
        int columna = 39;
        int novoDato = Attribute.NOMINAL;
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadCSV(
            pathEntrada);
```

```
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            vista.getControlador().manexarEvento(Controlador.
            MUDAR_TIPO, new Object[]{columna, novoDato});
            NumericToNominal filtro = new NumericToNominal();
            filtro.setOptions(new String[]{"-R", String.valueOf(
            columna + 1)});
            filtro.setInputFormat(is1);
            is1 = new ComparableInstances(Filter.useFilter(is1,
            filtro));
            ComparableInstances is2 = modelo.getComparableInstances
            ();
            is1.setRelationName("");
            is2.setRelationName("");
            assertEquals(is1, is2);
        } catch (Exception ex) {
            Logger.getLogger(TestRF09.class.getName()).log(Level.
            SEVERE, null, ex);
        }
    }
```

Descrición

Para comparar con algo fiable a conversión de tipos que implementa JDataMotion tivemos que recorrer á documentación de Weka, na que mencionan como empregar a clase NumericToNominal para converter un tipo numérico en nominal. O resultado asignámolo como novo valor de "is1", e comparámolo co que resulta de enviar ao Controlador o evento MUDAR_TIPO acompañado do índice dun atributo que é numérico e da constante Attribute.NOMINAL, que é o tipo ao que o queremos converter.

Resultado

Correcto.

RF10

Título

Sinalar identificación temporal

Descrición

A aplicación debe permitir sinalar unha columna que exprese o orde ou a temporalidade dunha tupla, ou ben definir esta columna manualmente.

Importancia

Esencial

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF10

Código fonte

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        int columna = 6;
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            vista.getControlador().manexarEvento(Controlador.
            MUDAR_INDICE_TEMPORAL, columna);
            assertEquals(modelo.getIndiceTemporal(), columna);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

Descrición
Este test non comproba InstancesComparable, se non enteiros. Concretamente, tras disparar un evento MUDAR_INDICE_TEMPORAL co índice de columna, chequéase que logo no Modelo o índice temporal equivala ao índice da columna dada.

Resultado
Correcto.

RF11

Título
Representar os datos graficamente mediante diagrama de dispersión

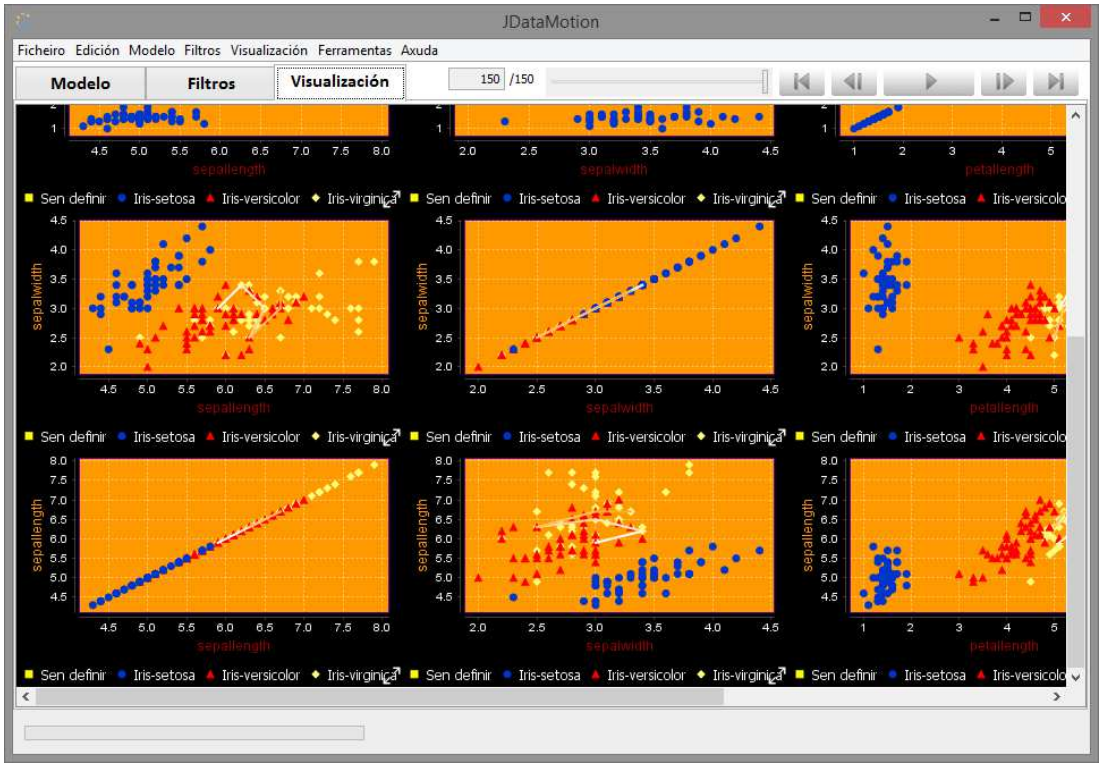


Figura 5.2: Comprobación heurística do RF11

Descrición

A aplicación debe ser capaz de representar graficamente (mediante diagrama de dispersión) o conxunto de parámetros de entrada. Concretamente, débense poder representar ata 3 parámetros por cada diagrama de dispersión (ordeadas, abscisas e cor e forma dos puntos). Todos os diagramas de dispersión estarán englobados dentro do "menú de visualización", que cumprirá co RNF04.

Importancia

Esencial

Tipo de proba

Avaliación heurística

Descrición

Abrimos un ficheiro ARFF ou CSV e imos á lapela de Visualización. Na barra de menús, iremos a Visualización > Establecer atributo nominal representado e seleccionamos un atributo. Aceptamos e observamos como en cada diagrama hai 2 atributos representados nos eixos, máis un atributo de clase representado polo estilo dos puntos que contén.

Resultado
Correcto.

RF12

Título
Engadir diagramas de dispersión ao menú de visualización

Descrición
A aplicación debe permitir engadir dinamicamente novos diagramas de dispersión dentro do menú de visualización.

Importancia
Esencial

Tipo de proba
Avaliación heurística

Descrición
Abrimos un ficheiro ARFF ou CSV e imos á lapela de Visualización. Na barra de menús, iremos a Visualización > Engadir ou eliminar scatterplots e seleccionamos da matriz os elementos que queremos engadir á visualización.

Resultado
Correcto.

RF13

Título
Eliminar un diagrama de dispersión do menú de visualización

Descrición
A aplicación debe permitir eliminar un diagrama de dispersión do menú de visualización.

Importancia
Esencial

Tipo de proba
Avaliación heurística

Descrición
Consideramos que xa se esta visualizando algún scatterplot. Na barra de

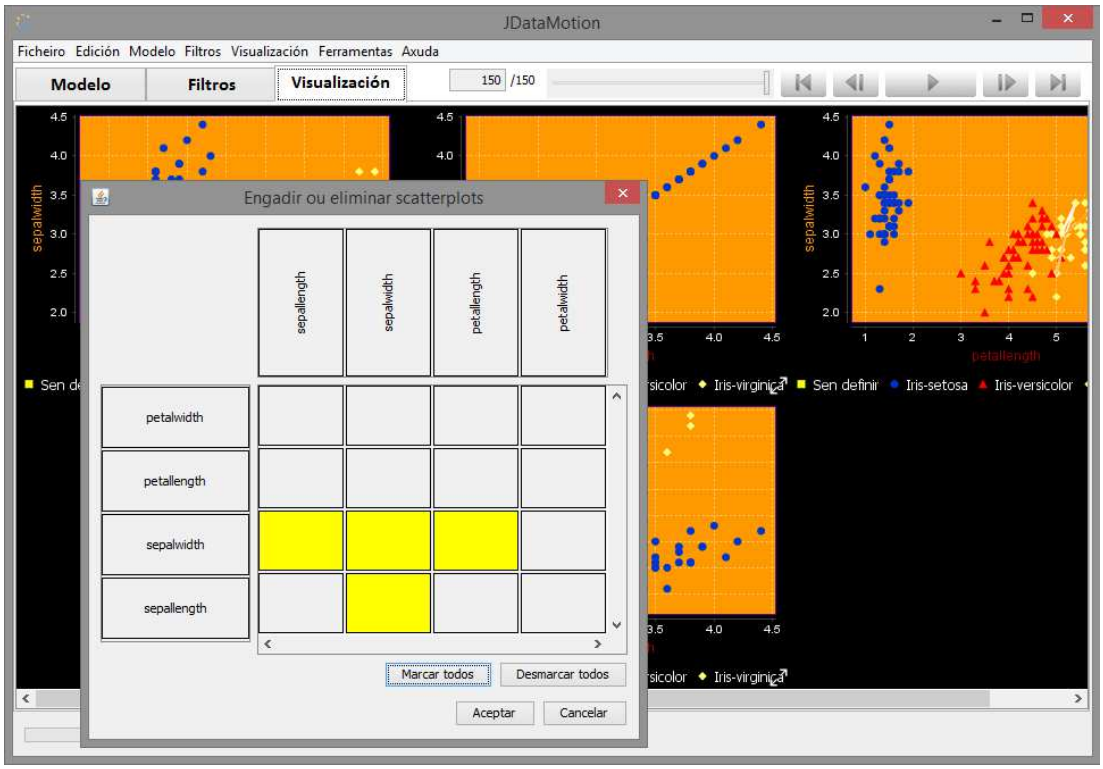


Figura 5.3: Comprobación heurística do RF12

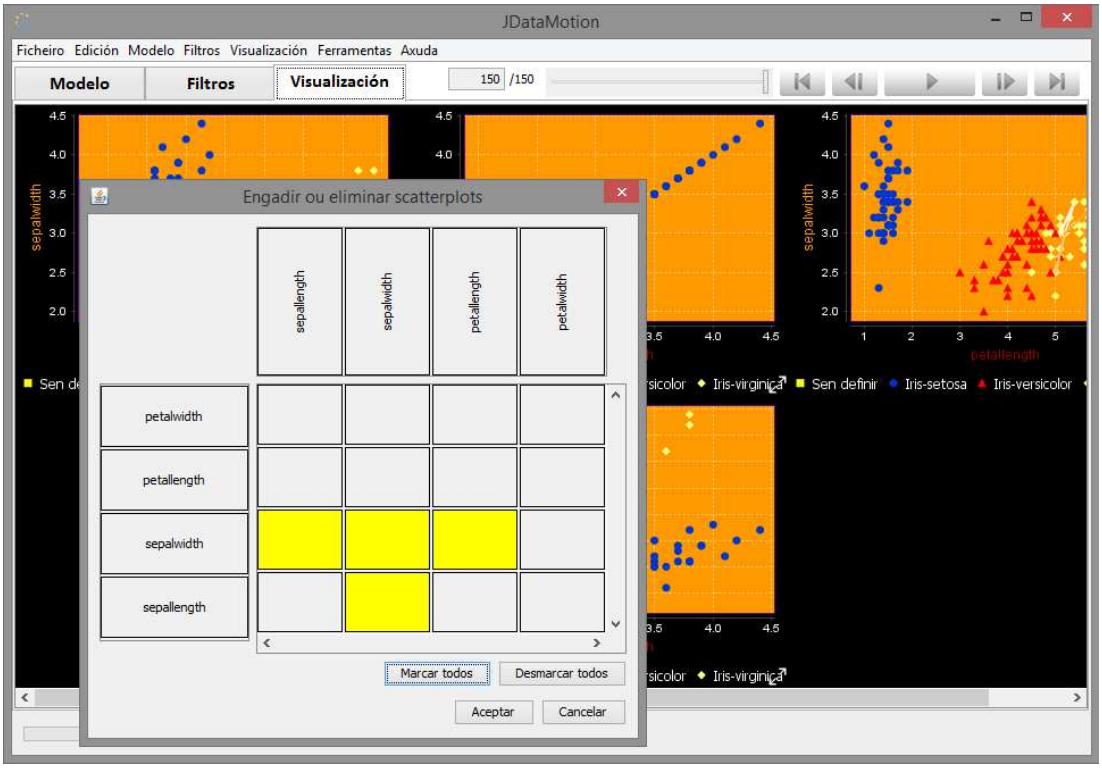


Figura 5.4: Comprobación heurística do RF13

menús, iremos a Visualización > Engadir ou eliminar scatterplots e desmarcamos da matriz os elementos que queremos quitar da visualización.

Resultado

Correcto.

RF14

Título

Configurar diagramas de dispersión do menú de visualización

Descrición

A aplicación debe permitir especificar para os diagramas de dispersión do menú de visualización a súa configuración, respecto a que parámetros se representarán en cada un dos eixos ou si as cores e formas dos puntos se desexan usar para representar algún atributo nominal.

Importancia

Esencial

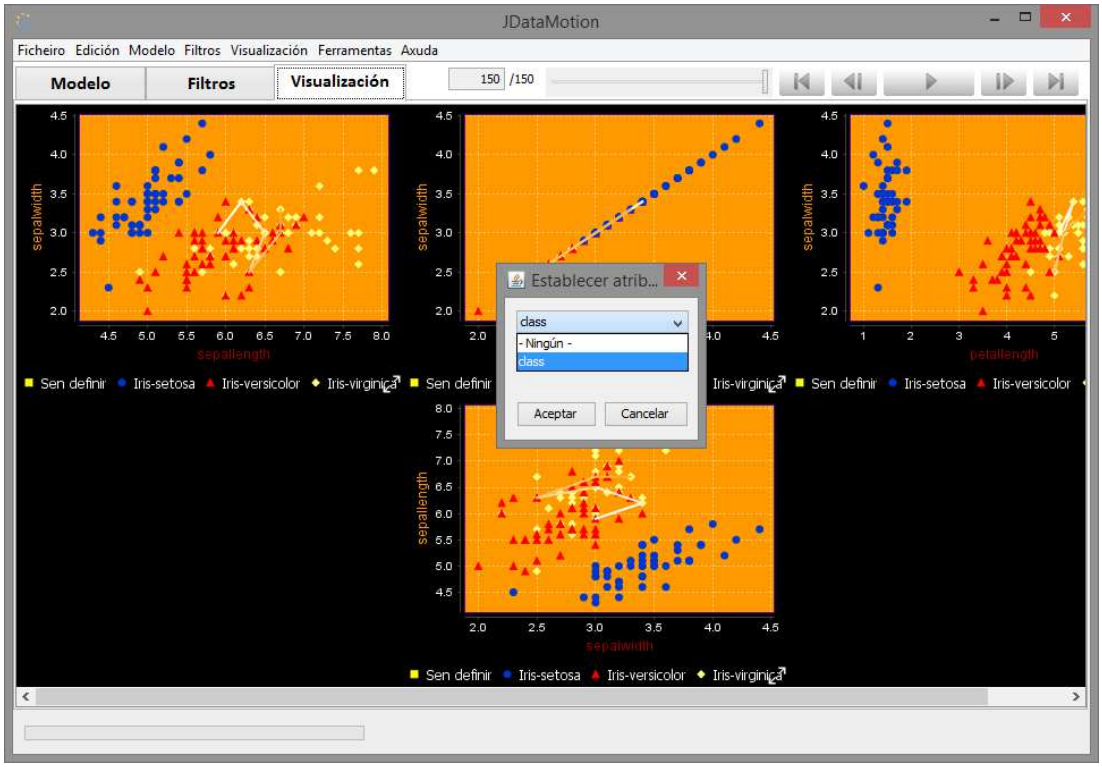


Figura 5.5: Comprobación heurística do RF14

Tipo de proba

Avaliación heurística

Descrición

Para configurar os atributos representados en cada eixo está a matriz e3 selección de Visualización > Engadir ou eliminar scatterplots. (ver RF12 e RF13). Para representar un atributo de clase por medio de cores e formas diferentes, só temos que seleccionalo en Visualización > Establecer atributo nominal representado. Ao aceptar podemos apreciar o resultado.

Resultado

Correcto.

RF15

Título

Detallar punto seleccionado dentro do diagrama de dispersión

Descrición

Cada punto dos diagramas de dispersión pode ser seleccionado para ver nun

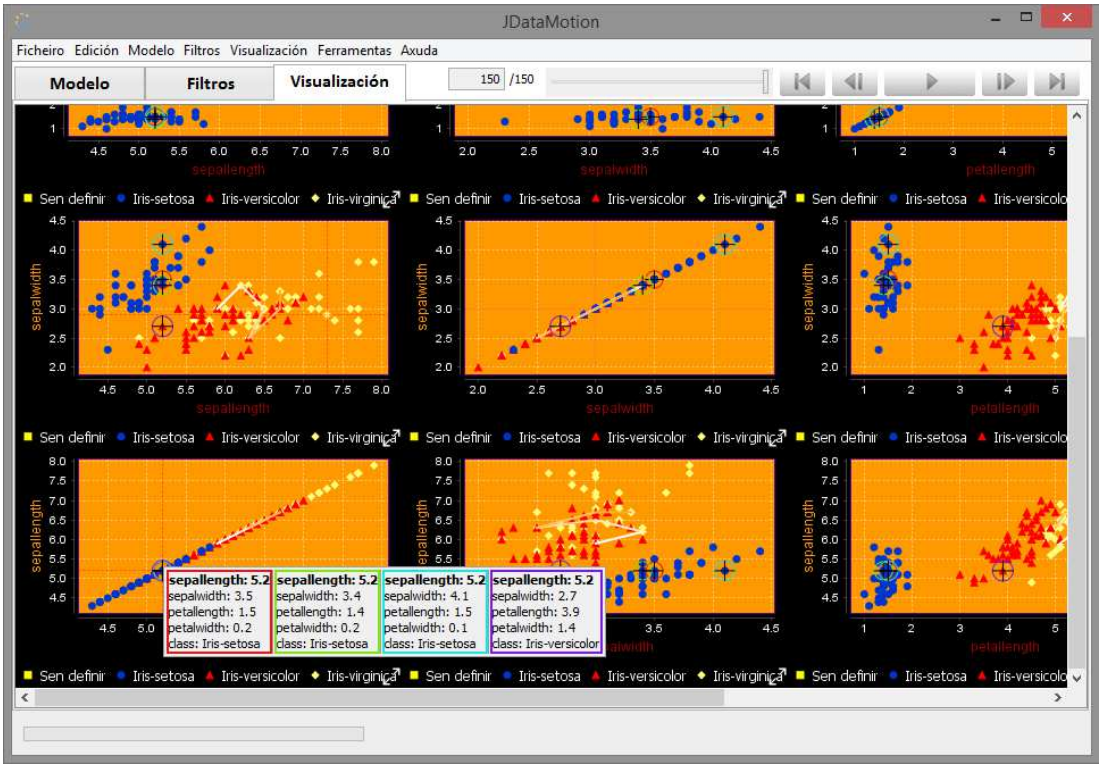


Figura 5.6: Comprobación heurística do RF15

apartado os seus detalles (todos os seus atributos).

Importancia
Esencial

Tipo de proba
Avaliación heurística

Descrición
Tendo algún diagrama representado podemos apreciar este efecto pulsando preto de calquera punto. Abrirase un menú que detalle o punto seleccionado, ou unha lista deles (puntos solapados).

Resultado
Correcto.

RF16

Título
Resaltar punto en diagramas de dispersión

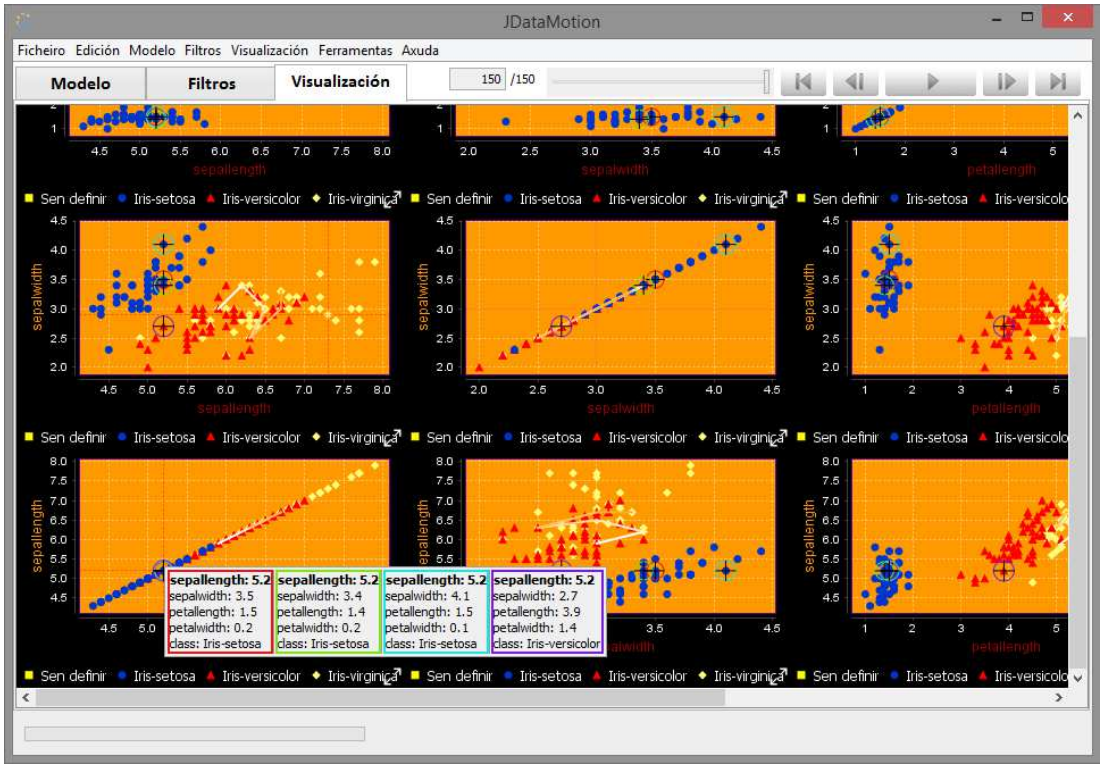


Figura 5.7: Comprobación heurística do RF16

Descrición

Cada punto seleccionalo dentro dun diagrama de dispersión resaltarase tanto nel coma en todos os demais diagramas de dispersión (que plasmarán outras proxeccións do mesmo punto).

Importancia

Esencial

Tipo de proba

Avaliación heurística

Descrición

Tendo algúns diagramas representados podemos apreciar este efecto pulsando nun punto no que haxa algún outro punto coincidindo nesa mesma posición (puntos solapados). As proyeccións desos puntos noutros diagramas tamén se resaltan, mantendo o mesmo color as que referencian a mesma instancia.

Resultado

Correcto.

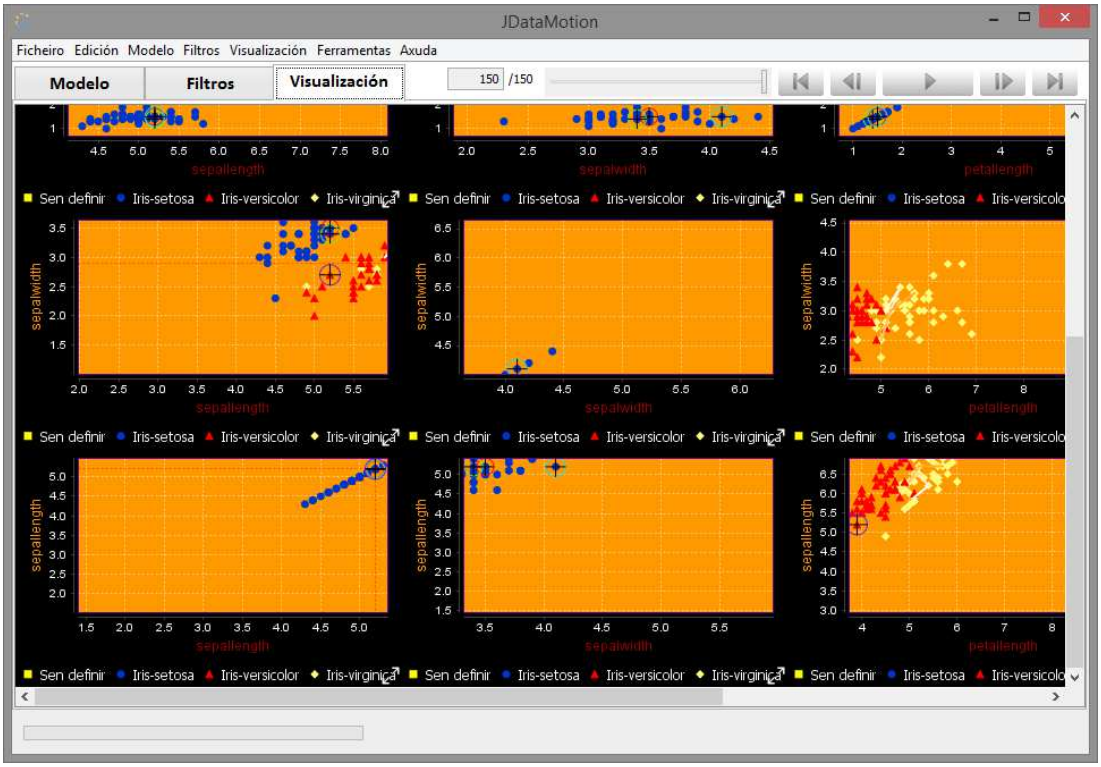


Figura 5.8: Comprobación heurística do RF17

RF17

Título

Desprazar a ventá de visualización por arrastre de cada diagrama de dispersión

Descrición

Para cada diagrama de dispersión poderemos usar unha ferramenta "man" para desprazar a ventá polo diagrama de dispersión.

Importancia

Esencial

Tipo de proba

Avaliación heurística

Descrición

Tendo algún diagrama representado podemos apreciar este efecto pulsando a tecla Control e arrastrando co rato un diagrama de dispersión. Veremos como a ventá que enfoca o gráfico se despraza na dirección que marquemos.

Resultado
Correcto.

RF18

Título
Escalar a ventá de visualización de cada diagrama de dispersión

Descrición
Para cada diagrama de dispersión poderemos usar unha ferramenta de escalado da ventá para facer zoom no diagrama de dispersión.

Importancia
Esencial

Tipo de proba
Avaliación heurística

Descrición
Tendo algún diagrama representado podemos apreciar este efecto colocando o cursor enriba dun diagrama e xirando a roda do rato. Veremos como a ventá de visualización do diagrama escala. Tamén se pode probar pulsando co botón secundario nun diagrama e seleccionando unha opción dentro de Acercar ou Alonxar.

Resultado
Correcto.

RF19

Título
Escalar e reposicionar dinamicamente

Descrición
Para cada diagrama de dispersión permitirase que a ventá de visualización que o enfoca se adapte dinamicamente ao conxunto de datos representados (movéndose, afastándose e aproximándose para englobar todos os datos).

Importancia
Esencial

Tipo de proba
Avaliación heurística

Descrición
Tendo algún diagrama representado podemos apreciar este efecto colocando

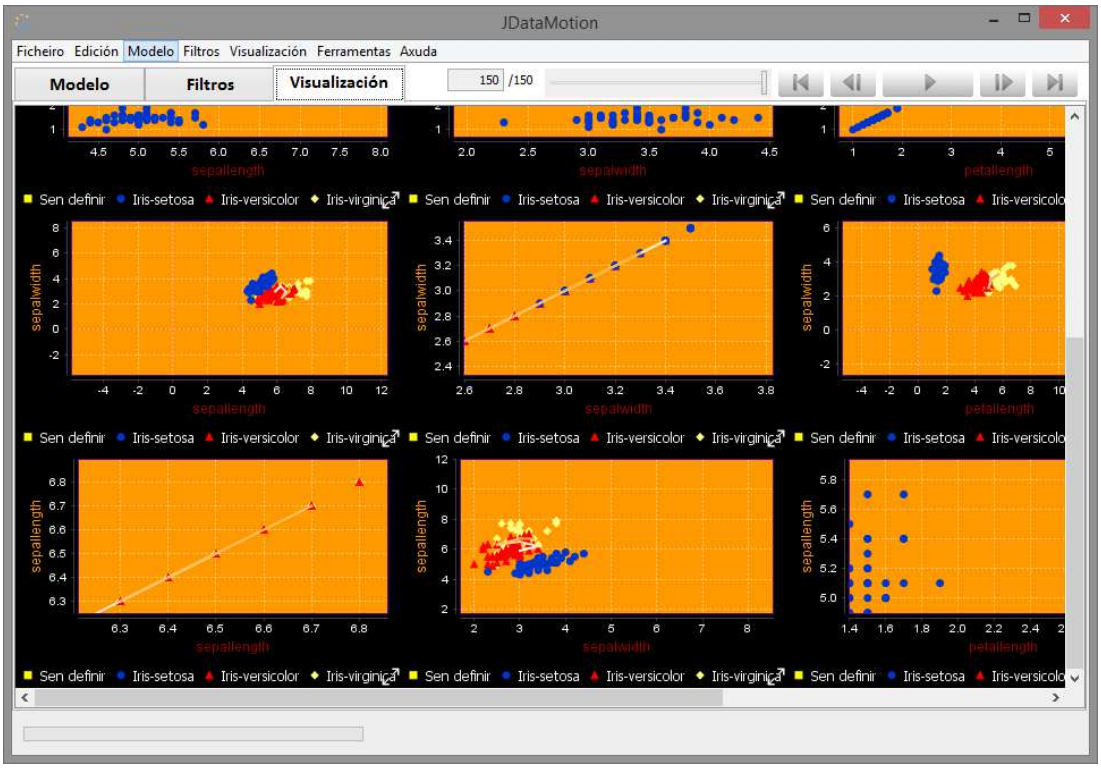


Figura 5.9: Comprobación heurística do RF18

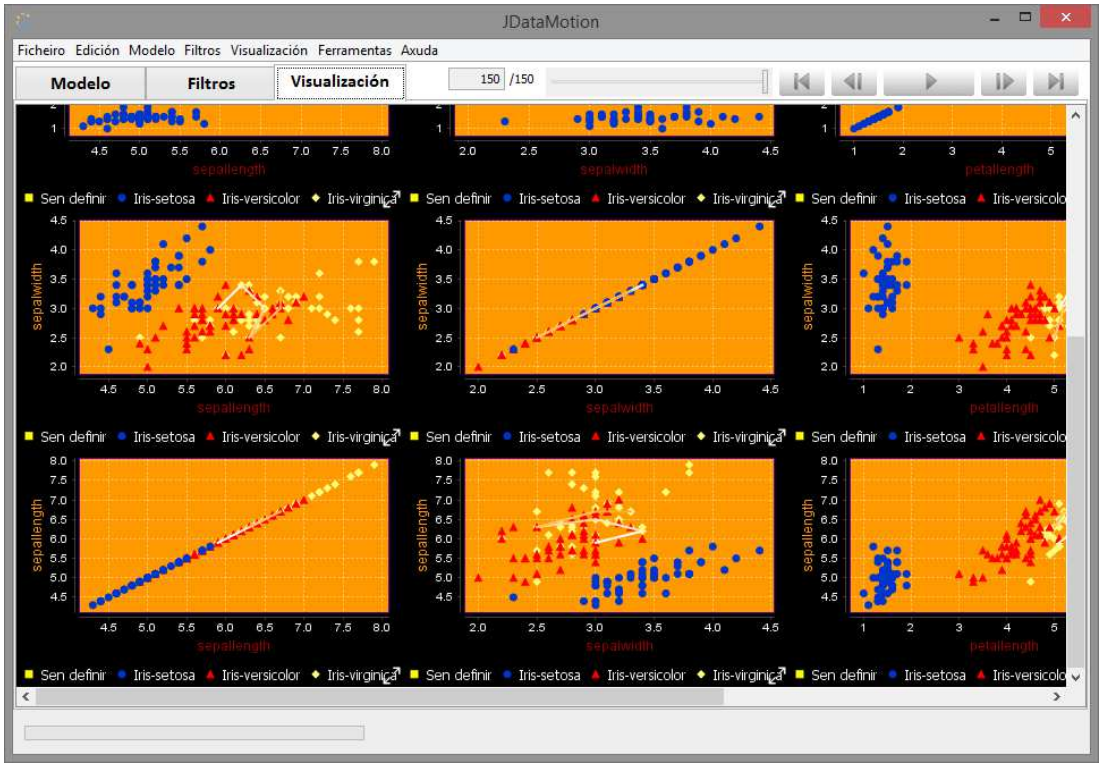


Figura 5.10: Comprobación heurística do RF19

o cursor enriba dun diagrama e xirando a roda do rato. Veremos como a ventá de visualización do diagrama escala.

Resultado

Correcto.

RF20

Título

Reproducir a secuencia de datos

Descrición

A aplicación debe de permitir que a visualización dos diagramas de dispersión poida basearse na variable temporal (ou de orde) para reproducir a secuencia de datos, amosando os datos de cada diagrama de dispersión baixo unha secuencia de vídeo. Nesta secuencia engadiríase á visualización en cada instante a tupla de atributos asociada a esa marca temporal.

Importancia

Esencial

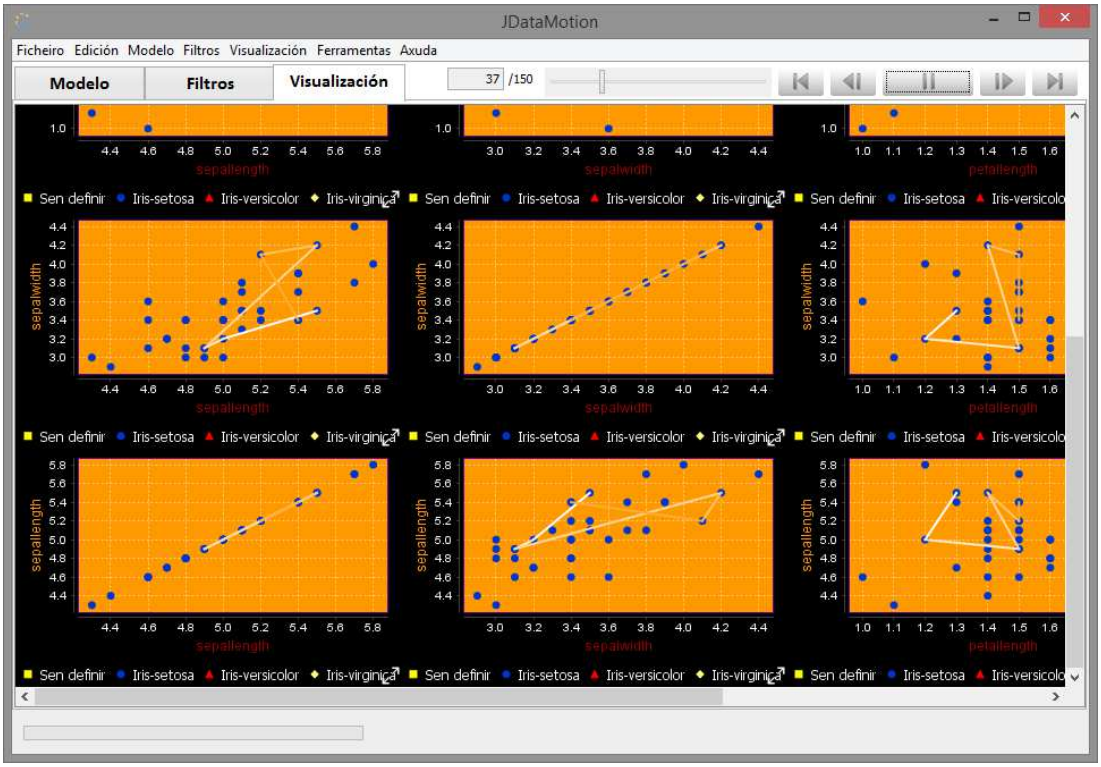


Figura 5.11: Comprobación heurística do RF20

Tipo de proba

Avaliación heurística

Descrición

Tendo algún diagrama representado, volvemos ao Modelo e seleccionamos un atributo numérico ou String (pero cun formato de tempo) pinchando nel co botón secundario. Marcámolo como índice temporal. Agora imos a Visualización > Configurar reprodución e personalizamos os valores que se amosan. Para que se teña en conta o índice temporal, seleccionaremos en Orde do modelo a opción "Orde dos índices temporais numéricos" ou ben "Orde dos índices temporais numéricos ponderados", en función de si o índice temporal expresa só orde ou unha duración relativa, respectivamente. Con todo isto, ao premer no botón de reprodución do menú Visualización comezaremos a ver a secuencia de puntos avanzar da forma que acabamos de configurar.

Resultado

Correcto.

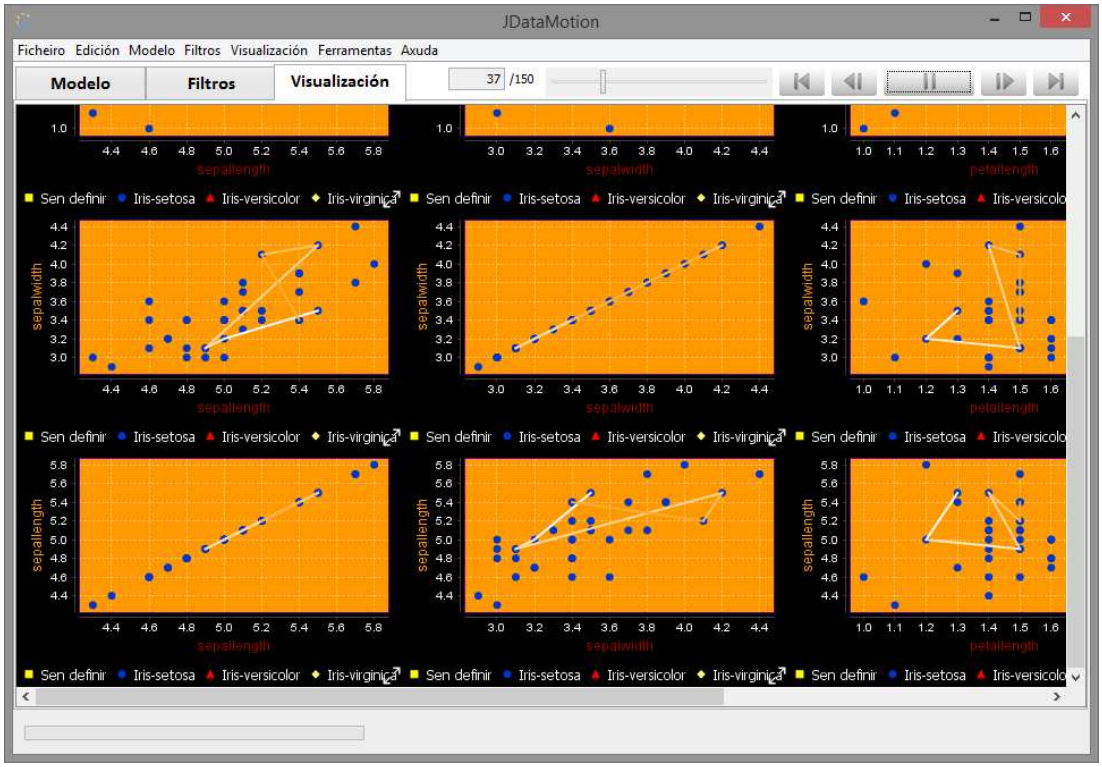


Figura 5.12: Comprobación heurística do RF21

RF21

Título

Representar estela

Descrición

A aplicación debe de permitir que cada novo punto pintado se ligue ao último representado no diagrama de dispersión por medio dunha liña recta.

Importancia

Esencial

Tipo de proba

Avaliación heurística

Descrición

Tendo algún diagrama representado, imos a Visualización > Configurar reprodución e asegurámonos de que o valor da lonxitude de estela non sexa 0. Entón premendo no botón de reproducir veremos que a estela abarcará o número de puntos especificado, tendo a súa cabeza no último punto que se visualiza en cada momento.

Resultado
Correcto.

RF22

Título
Difuminar estela ao longo da reprodución

Descrición
A aplicación debe permitir difuminar as estelas xa representadas a través do avance temporal.

Importancia
Esencial

Tipo de proba
Avaliación heurística

Descrición
Tendo algún diagrama representado, imos a Visualización > Configurar reprodución e asegurámonos de que o valor da lonxitude de estela sexa un número significativo. Entón premendo no botón de reproducir veremos que a estela abarcará o número de puntos especificado, tendo unha cor máis intensa nos tramos nos que une puntos máis recentes, e máis tenue naqueles con maior antigüidade.

Resultado
Correcto.

RF23

Título
Configurar a reprodución da secuencia de datos

Descrición
A aplicación debe de permitir que a visualización dos diagramas de dispersión sexa configurable en canto a tempo transcorrido entre marcas temporais. Para a reprodución usando marcas temporais ponderadas, este tempo representará a separación entre as dúas marcas temporais mais próximas (tempo mínimo). Además débese poder especificar o número de marcas temporais que durará o difuminado dos puntos que se ploteen, de xeito que durante ese intervalo cada punto se vaia difuminando ata desaparecer. Pode ser igual a 0 para que os puntos non se difuminen.

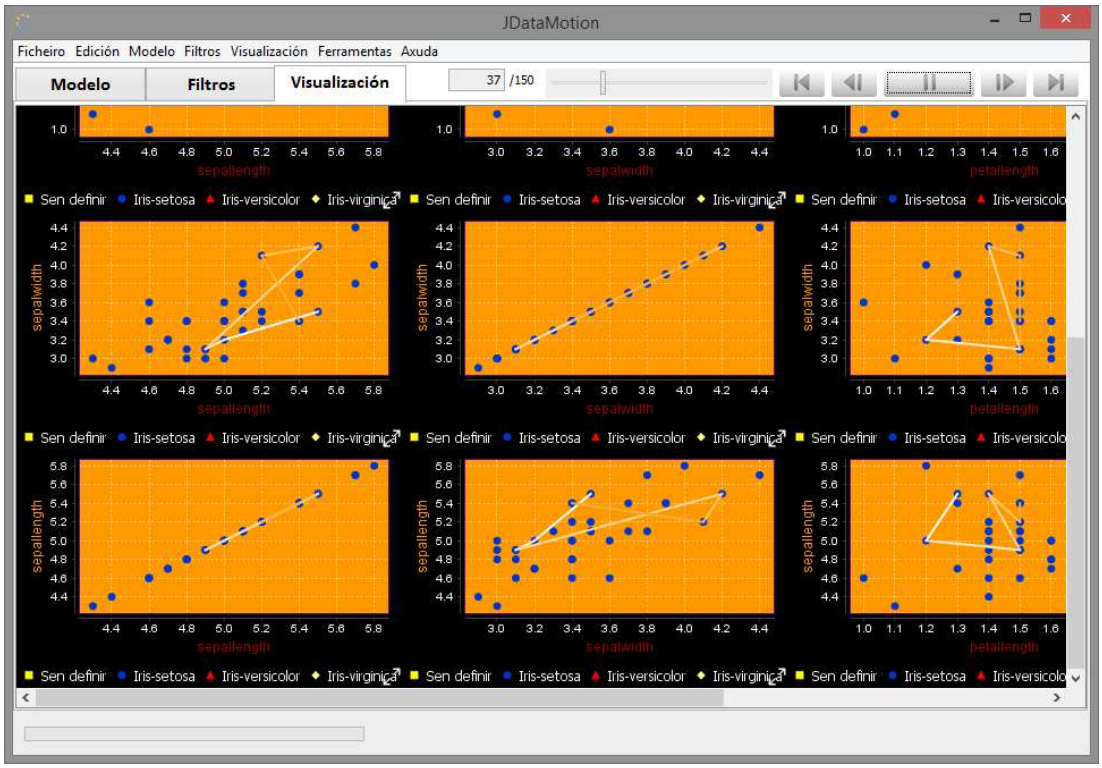


Figura 5.13: Comprobación heurística do RF22

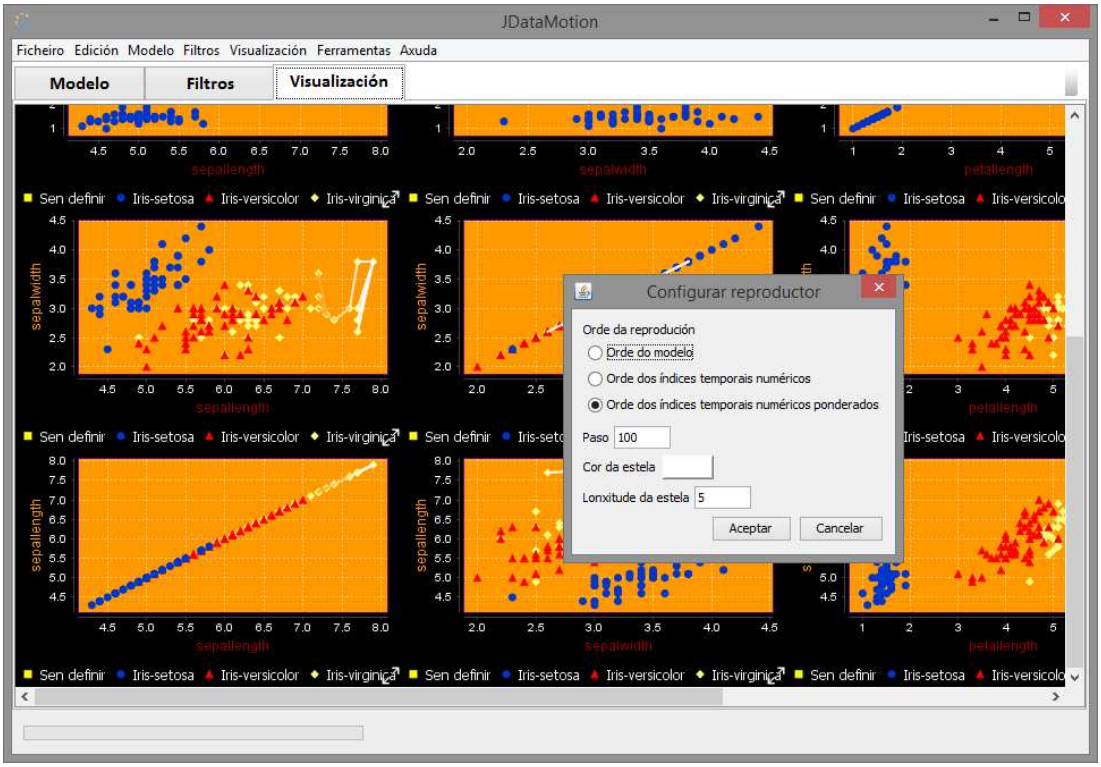


Figura 5.14: Comprobación heurística do RF23

Importancia

Esencial

Tipo de proba

Avaliación heurística

Descrición

Tendo algún diagrama representado, volvemos ao Modelo e seleccionamos un atributo numérico ou String (pero cun formato de tempo) pinchando nel co botón secundario. Marcámolo como índice temporal. Agora imos a Visualización >. Para que se teña en conta o índice temporal, seleccionaremos en Orde do modelo a opción "Orde dos índices temporais numéricos ponderados". Con todo isto, ao premer no botón de reprodución do menú Visualización comezaremos a ver a secuencia de puntos avanzar da forma que acabamos de configurar. O difuminado dos puntos non se pode probar porque non foi implementado todavía (debido a unha restricción das estruturas do Dataset que non permitían a reprodución cara atrás).

Resultado

Correcto.

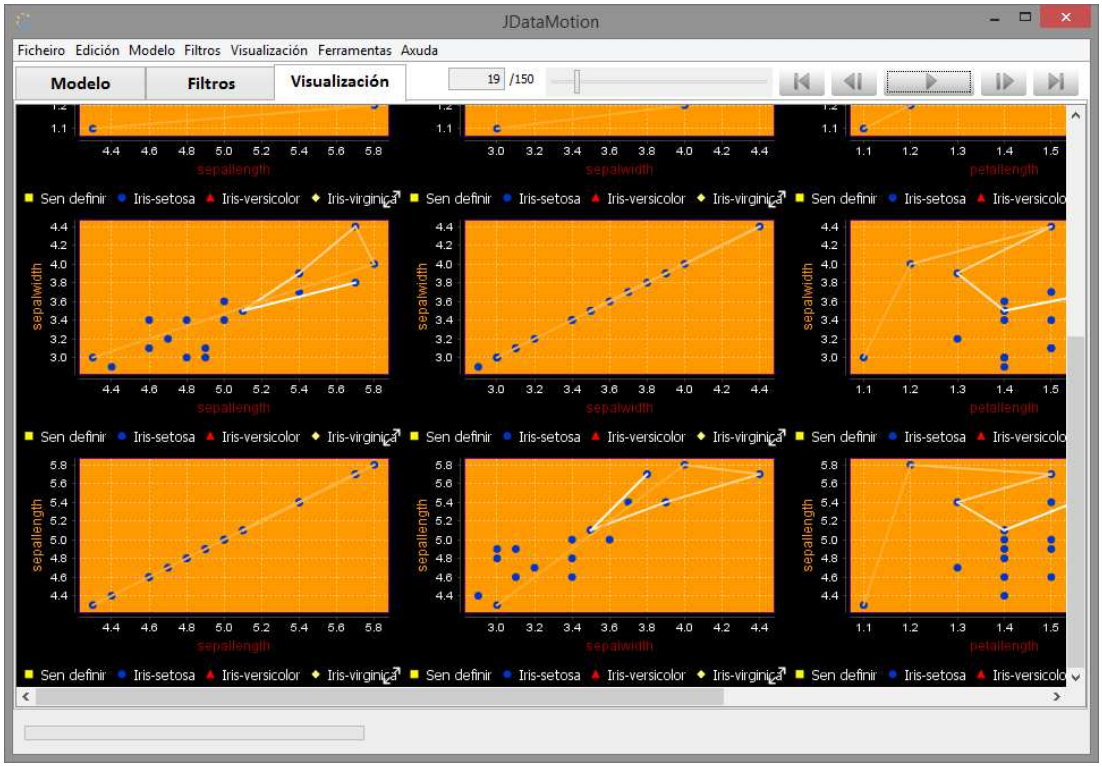


Figura 5.15: Comprobación heurística do RF24

RF24

Título

Pausar a reprodución

Descrición

A aplicación debe permitir parar a reprodución na marca de tempo na que se atope ao executar esta acción, mantendo as visualizacións para ese momento.

Importancia

Esencial

Tipo de proba

Avaliación heurística

Descrición

Tendo algún diagrama representado e coa reprodución en curso, prememos outra vez no botón de reprodción (que agora terá un símbolo de dúas barras verticais paralelas). Comprobamos que todos os diagramas deteron a súa reprodución.

Resultado
Correcto.

RF25

Título
Ir a un determinado instante dentro do intervalo temporal da reprodución

Descrición
A aplicación debe permitir situarse directamente sobre un instante de tempo, mantendo a reprodución pausada sobre esa marca temporal, e visualizando os diagramas de dispersión tal e como deben estar nese momento.

Importancia
Esencial

Tipo de proba
Avaliación heurística

Descrición
Tendo algún diagrama representado, observamos o desprazador que está no menú de Visualización, ao lado dos demais botóns de reprodución. Arrastramos o seu pivote por todo o ancho e observamos como aparecen (cando nos movemos acara a dereita) ou se agochan (cando nos movemos cara a esquerda) puntos en todos os diagramas. Soltando o pivote nunha posición a reprodución establécese nela, seguindo a partir de aí no caso de reanudarse.

Resultado
Correcto.

RF26

Título
Insertar filtros para os datos do experimento

Descrición
A aplicación debe permitir engadir unha serie de filtros que se aplicarán de xeito secuencial sobre a secuencia de datos coa que se esté a traballar. Chamarémoslle "secuencia de filtros" a esta secuencia.

Importancia
Esencial

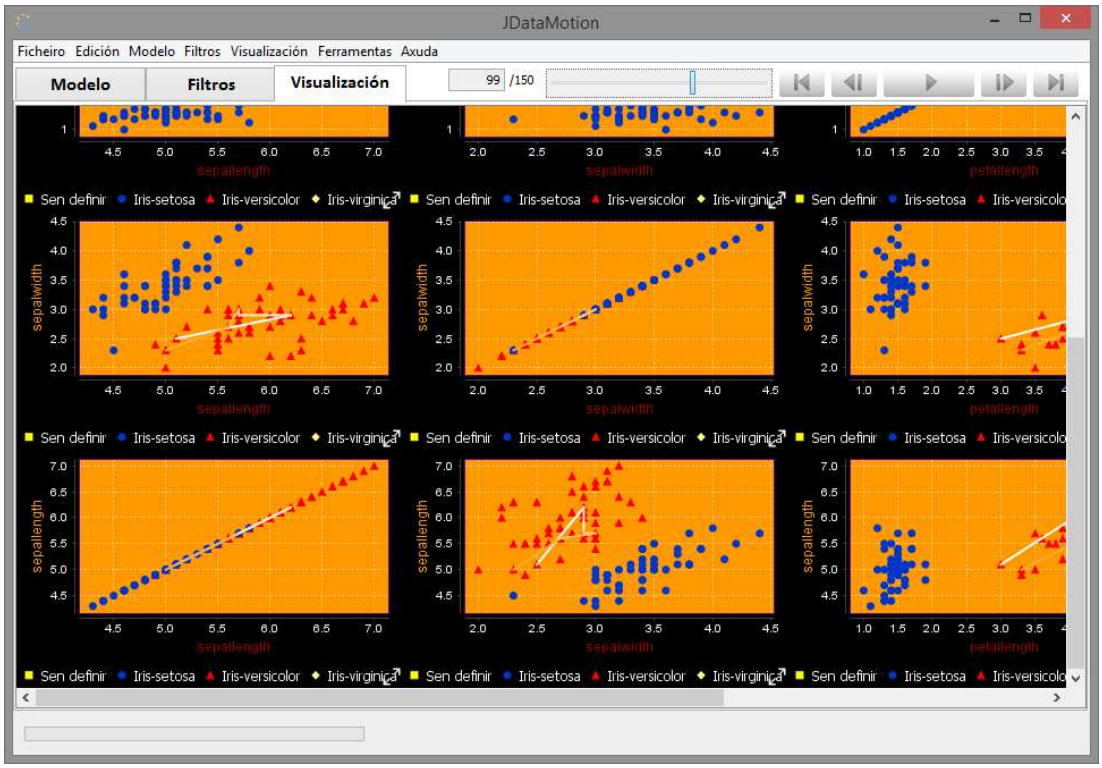


Figura 5.16: Comprobación heurística do RF25

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF2628

Código fonte

```
public void test() {
    Modelo modelo = new Modelo();
    Vista vista = new Vista();
    vista.inicializar(modelo, false);
    Controlador.setDebug(true);
    String resource = "example01.arff";
    IFilter filtro = new FiltroLimite();
    Double maximo = 15.0;
    int columna = 1;
    StringParameter sp1 = new StringParameter(), sp2 = new
    StringParameter();
    DoubleParameter sp3 = new DoubleParameter();
    sp1.setValue(recursosIdioma.getString("limiteValor"));
    sp2.setValue(recursosIdioma.getString("cotaSuperior"));
    sp3.setValue(maximo);
    try {
        String pathEntrada = new URI(getClass().getResource(
        resource).toString()).getPath();
        ComparableInstances is1 = ValidFileLoading.loadARFF(
        pathEntrada);
        modelo.setComparableInstances(new ComparableInstances(
        is1));
        vista.getControlador().manexarEvento(Controlador.
        ENGADIR_FILTRO, new Object[]{0, filtro});
        modelo.getFiltro(0).getParameters().put(recursosIdioma.
        getString("tipoLimite"), sp1);
        modelo.getFiltro(0).getParameters().put(recursosIdioma.
        getString("tipoCota"), sp2);
        modelo.getFiltro(0).getParameters().put(recursosIdioma.
        getString("valor"), sp3);
        modelo.getFiltro(0).setIndiceAtributoFiltrado(columna);
        boolean todosMenores = true;
        Enumeration<Instance> e = modelo.
        getComparableInstancesFiltradas().enumerateInstances();
        while (e.hasMoreElements()) {
            Instance i = e.nextElement();
            if (i.value(columna) > maximo) {
                todosMenores = false;
            }
```

```
            }
            assertEquals(todosMenores, true);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

Descrición

Este test comproba si ao enviar un evento de tipo ENGADIR_FILTRO e configurar o filtro en cuestión, se aprecian os seus efectos no método getComparableInstancesFiltradas. En concreto pasaráselle un FiltroLimite que poña a 15 todos os valores da 2º columna que o superen. Estes datos constitúen a configuración do filtro, a cal tamén se proba neste método.

Resultado

Correcto.

RF27

Título

Eliminar un filtro para os datos do experimento

Descrición

A aplicación debe permitir eliminar un determinado filtro dentro da secuencia de filtros.

Importancia

Esencial

Tipo de proba

Test implementado en JUnit.

Nome do test

TestRF2627

Código fonte

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        IFilter filtro = new FiltroLimite();
        Double maximo = 15.0;
```

```
        int columna = 1;
        StringParameter sp1 = new StringParameter(), sp2 = new
        StringParameter();
        DoubleParameter sp3 = new DoubleParameter();
        sp1.setValue(recursosIdioma.getString("limiteValor"));
        sp2.setValue(recursosIdioma.getString("cotaSuperior"));
        sp3.setValue(maximo);
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            vista.getControlador().manexarEvento(Controlador.
            ENGADIR_FILTRO, new Object[]{0, filtro});
            modelo.getFiltro(0).getParameters().put(recursosIdioma.
            getString("tipoLimite"), sp1);
            modelo.getFiltro(0).getParameters().put(recursosIdioma.
            getString("tipoCota"), sp2);
            modelo.getFiltro(0).getParameters().put(recursosIdioma.
            getString("valor"), sp3);
            modelo.getFiltro(0).setIndiceAtributoFiltrado(columna);
            vista.getControlador().manexarEvento(Controlador.
            ELIMINAR_FILTRO, 0);
            ComparableInstances is2 = modelo.
            getComparableInstancesFiltradas();
            assertEquals(is1, is2);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

Descrición

Este test comproba si despois de eliminar un filtro que engadimos e configuramos, obtemos unhas ComparableInstances equivalentes ao estado inicial do experimento, o que significa que o evento ELIMINAR_FILTRO e ENGADIR_FILTRO funcionan correctamente.

Resultado

Correcto.

RF28

Título
Configurar filtros para os datos do experimento

Descrición
A aplicación debe permitir seleccionar un determinado filtro dentro da secuencia de filtros para modificar a regla de filtrado implícita.

Importancia
Esencial

Nome do test
TestRF2628

Descrición
Ver TestRF2628.

Resultado
Correcto.

RF29

Título
Gardar unha secuencia de filtros do experimento

Descrición
A aplicación debe permitir gardar unha secuencia de filtros, non necesariamente correlativos, dentro dos que se estean aplicando sobre o experimento. Esta secuencia pode comprender tanto un só filtro como a secuencia de filtros enteira.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF262930

Código fonte

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
```

```
String resource = "example01.arff";
String archivoSesion = "tempFiltros.jdmf";
IFilter filtro = new FiltroLimite();
Double maximo = 15.0;
int columna = 1;
StringParameter sp1 = new StringParameter(), sp2 = new
StringParameter();
DoubleParameter sp3 = new DoubleParameter();
sp1.setValue(recursosIdioma.getString("limiteValor"));
sp2.setValue(recursosIdioma.getString("cotaSuperior"));
sp3.setValue(maximo);
try {
    String pathSaida = new URI(getClass().getResource(".").
    toString()).getPath() + archivoSesion;
    String pathEntrada = new URI(getClass().getResource(
    resource).toString()).getPath();
    ComparableInstances is1 = ValidFileLoading.loadARFF(
    pathEntrada);
    modelo.setComparableInstances(new ComparableInstances(
    is1));
    vista.getControlador().manexarEvento(Controlador.
    ENGADIR_FILTRO, new Object[]{0, filtro});
    modelo.getFiltro(0).getParameters().put(recursosIdioma.
    getString("tipoLimite"), sp1);
    modelo.getFiltro(0).getParameters().put(recursosIdioma.
    getString("tipoCota"), sp2);
    modelo.getFiltro(0).getParameters().put(recursosIdioma.
    getString("valor"), sp3);
    modelo.getFiltro(0).setIndiceAtributoFiltrado(columna);
    vista.getControlador().manexarEvento(Controlador.
    EXPORTAR_FILTROS, new Object[]{pathSaida, new Integer
    []{0}});
    vista.getControlador().manexarEvento(Controlador.
    IMPORTAR_FILTROS, pathSaida);
    boolean valido = true;
    for (Entry<String, Parameter> entry : modelo.getFiltro
    (0).getParameters().entrySet()) {
        if (!modelo.getFiltro(0).getParameters().get(entry.
        getKey()).getValue().equals(modelo.getFiltro(1).
        getParameters().get(entry.getKey()).getValue())) {
            valido = false;
            break;
        }
    }
    if (valido == true
```

```
                && (!modelo.getFiltro(0).getFiltro().getClass().
                equals(modelo.getFiltro(1).getFiltro().getClass
                ())
                || modelo.getFiltro(0).getIndiceAtributoFiltrado
                () != columna
                || modelo.getFiltro(1).getIndiceAtributoFiltrado
                () != null)) {
            valido = false;
        }
        assertEquals(valido, true);
    } catch (URISyntaxException ex) {
        Logger.getLogger(getClass().getName()).log(Level.SEVERE
        , null, ex);
    }
}
```

Descrición

Este test comproba que a exportación e importación de filtros acada os seus obxectivos. En concreto, os filtros deben ser da mesma clase, o filtro importado debeu perder o índice do atributo durante a exportación, e o resto de parámetros mantivéronse intactos durante o proceso.

Resultado

Correcto.

RF30

Título

Cargar unha secuencia de filtros para o experimento

Descrición

A aplicación debe permitir cargar do sistema de arquivos unha secuencia de filtros que se engadirá á cabeza da secuencia de filtros (a cal pode estar baleira). Esta secuencia tamén pode estar composta por un só filtro.

Importancia

Esencial

Nome do test

TestRF262930

Descrición

Ver TestRF262930.

Resultado
Correcto.

RF31

Título
Mover os filtros dentro da secuencia de filtros

Descrición
A aplicación debe permitir desprazar un filtro dentro da secuencia de filtros do experimento, de xeito que o orde de aplicación dos filtros varíe. O desprazamento realizarase inserindo o filtro en cuestión nunha nova posición.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Nome do test
TestRF2631

Código fonte

```
public void test() {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        vista.inicializar(modelo, false);
        Controlador.setDebug(true);
        String resource = "example01.arff";
        String archivoSesion = "tempFiltros.jdmf";
        IFilter filtro1 = new FiltroLimite();
        IFilter filtro2 = new FiltroNormalizacion();
        Double maximo = 15.0;
        int columna = 1;
        StringParameter sp1 = new StringParameter(), sp2 = new
        StringParameter();
        DoubleParameter sp3 = new DoubleParameter();
        sp1.setValue(recursosIdioma.getString("limiteValor"));
        sp2.setValue(recursosIdioma.getString("cotaSuperior"));
        sp3.setValue(maximo);
        try {
            String pathEntrada = new URI(getClass().getResource(
            resource).toString()).getPath();
            ComparableInstances is1 = ValidFileLoading.loadARFF(
            pathEntrada);
```

```
            modelo.setComparableInstances(new ComparableInstances(
            is1));
            vista.getControlador().manexarEvento(Controlador.
            ENGADIR_FILTRO, new Object[]{0, filtro1});
            modelo.getFiltro(0).getParameters().put(recursosIdioma.
            getString("tipoLimite"), sp1);
            modelo.getFiltro(0).getParameters().put(recursosIdioma.
            getString("tipoCota"), sp2);
            modelo.getFiltro(0).getParameters().put(recursosIdioma.
            getString("valor"), sp3);
            modelo.getFiltro(0).setIndiceAtributoFiltrado(columna);
            vista.getControlador().manexarEvento(Controlador.
            ENGADIR_FILTRO, new Object[]{1, filtro2});
            modelo.getFiltro(1).setIndiceAtributoFiltrado(columna);
            vista.getControlador().manexarEvento(Controlador.
            INTERCAMBIAR_FILTROS, new Object[]{0, 1});
            assertEquals(modelo.getFiltro(1).getFiltro().equals(
            filtro1) && modelo.getFiltro(0).getFiltro().equals(
            filtro2), true);
        } catch (URISyntaxException ex) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE
            , null, ex);
        }
    }
```

Descrición

Este test comproba que ao engadir dous filtros por medio do evento ENGADIR_FILTRO e intercambialos por medio de INTERCAMBIAR_FILTROS, cada un dos dous filtros está na posición que antes tiña o outro.

Resultado

Correcto.

RF32

Título

Configurar o menú de visualización

Descrición

A aplicación debe permitir cambiar os parámetros de visualización dos diagramas de dispersión que compoñen o menú de visualización, por exemplo, a cor das etiquetas e lendas, do fondo, dos eixos... ou a fonte, tamaño de letra...

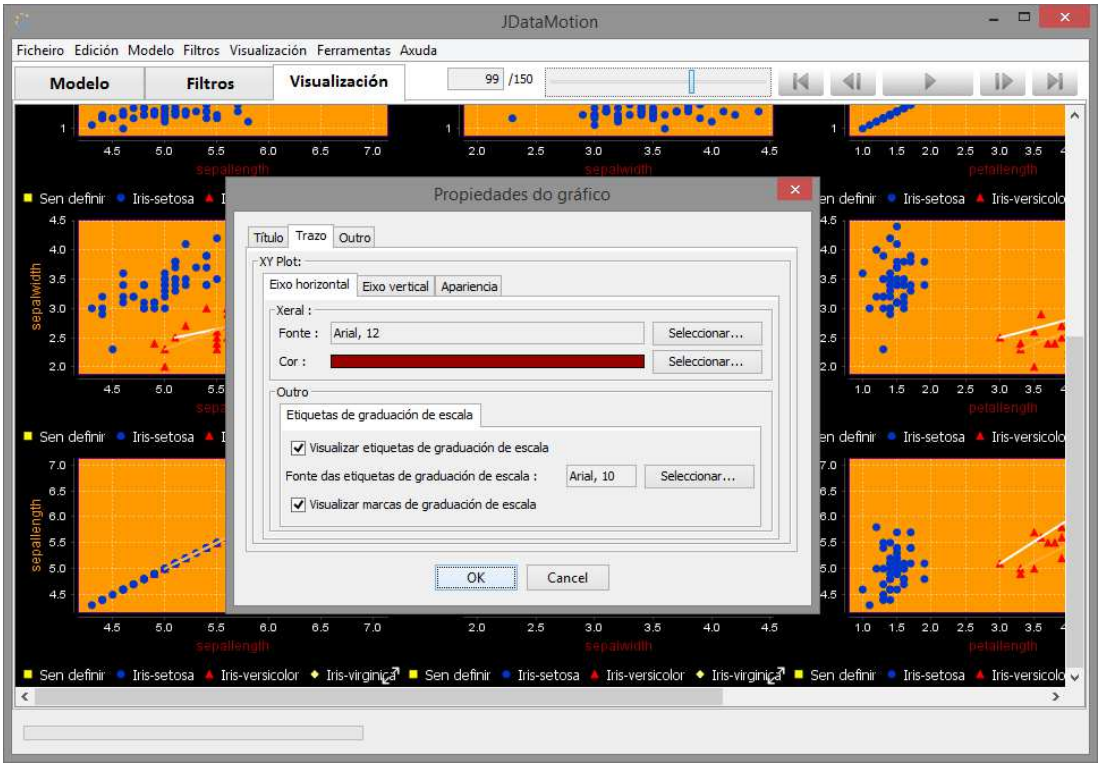


Figura 5.17: Comprobación heurística do RF32

Importancia

Optativa

Tipo de proba

Avaliación heurística

Descrición

Abrimos un ficheiro ARFF ou CSV e imos á lapela de Visualización. Podemos acceder ao menú de configuración gráfica pinchando co botón secundario nun diagrama ou mesmo no lenzo do menú. Sairá un menú contextual cunha opción chamada Propiedades, que abrirá o menú en cuestión. Podemos comprobar como toda a configuración insertada se aplica ao experimento en canto prememos "Aceptar".

Resultado

Correcto.

5.0.2. Requisitos de calidade

RC01

Título
Latencia mínima para o procesamento

Descrición
A aplicación debe responder nun tempo razoable ás operacións executadas polo usuario, e intentar que esa latencia escale de xeito controlado ao aumentar a talla dos parámetros.

Importancia
Esencial

Tipo de proba
Avaliación de prestacións

Descrición
Preparamos 16 ficheiros ARFF de datos, con 11, 22, 33 e 44 atributos, e 2500, 5000, 7500 e 10000 instancias, cubrindo todas as combinacións. Executamos estos ficheiros activando a bandeira Controlador.DEBUG para obter os tempos de execución das tres rexións que compoñen o refresco (a operación máis custosa do programa). En concreto, tomaremos individualmente o tempo que tarda en refrescar cada menú (Modelo, Filtros e Visualización coa matriz de diagramas de dispersión completa). O de filtros descartarémolo xa que a súa latencia é insignificante e non aumenta coa talla do arquivo. Os resultados obtidos foron os seguintes: A latencia vólvese crítica no refresco do menú de visualización, e a escalabilidade empeora co aumento do número de atributos. O Modelo sen embargo escala relativamente mellor con respecto a instancias e atributos. Cabe destacar que é ata certo punto razonable que o menú de Visualización escale tan mal, pois sufre os aumentos no número de atributos ao cadrado (a matriz de diagramas ten n*n elementos).

Resultado
Aceptable.

5.0.3. Requisitos de deseño

RD01

Título
Modularidade no deseño dos filtros

Instancias	Atributos	Modelo	Visualización
2500	11	0,110	1,312
	22	0,156	3,405
	33	0,578	9,910
	44	0,359	27,349
5000	11	0,172	0,984
	22	0,469	3,219
	33	0,515	13,535
	44	0,593	20,332
7500	11	0,234	1,032
	22	0,484	4,200
	33	0,797	12,679
	44	0,985	35,012
10000	11	0,157	6,360
	22	0,706	7,829
	33	0,975	14,438
	44	2,200	64,323

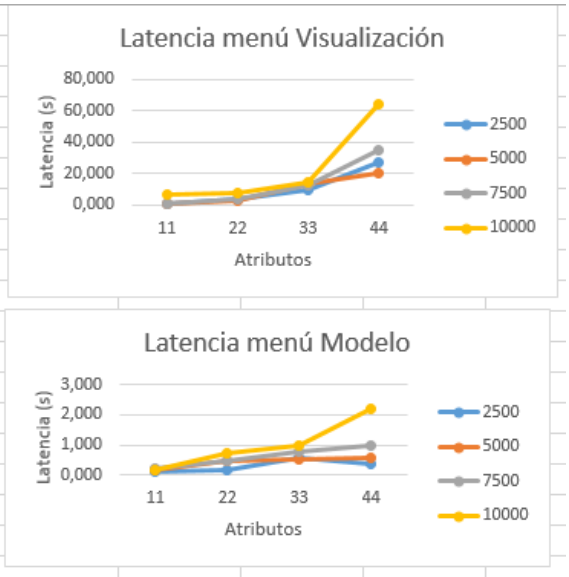


Figura 5.18: Comprobación prestacións do RC01

Descrición

A aplicación debe facilitar unha interface para a inclusión e uso de filtros personalizados por parte de calquera desenvolvedor de software que a implemente dentro do proxecto.

Importancia

Esencial

Tipo de proba

Avaliación heurística

Descrición

O diagrama de deseño exposto neste documento da fe de que se cumpleu este requisito. De todos xeitos pódese probar a nivel de implementación importando un filtro dende JAR.

Resultado

Correcto.

5.0.4. Requisitos non funcionais

RNF01

Título

Formatos de arquivo admitidos ao importar e exportar arquivos

Descrición
A aplicación debe estar preparada para importar e exportar arquivos en distintos formatos, como son o CSV e ARFF.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Descrición
Xa foi comprobado no TestRF01_1 e TestRF01_2

Resultado
Correcto.

RNF02

Título
Relación programa-sesión

Descrición
Cada instancia do programa debe traballar cunha única sesión (experimento).

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Descrición
Xa foi comprobado no TestRF0304, xa que ao abrir unha nova sesión perdíase a anterior

Resultado
Correcto.

RNF03

Título
Implementación en Java

Descrición
O software tense que desenvolver na linguaxe de programación Java.

Importancia
Esencial

Tipo de proba
Avaliación heurística

Descrición
Podemos comprobalo simplemente accedendo ao código fonte do proxecto.

Resultado
Correcto.

RNF04

Título
Representación matricial dos diagramas de dispersión

Descrición
Os diagramas de dispersión represéntanse de xeito matricial, facendo que cada parámetro dentro dun eixo sexa enfrontado a cada un dos demais do outro eixo, e en cada punto desa dupla se sitúe o diagrama de dispersión que compara ambos parámetros. Deste xeito, os diagramas de dispersión non son acumulables: se temos un que representa X (abscisas) fronte a Y (ordenadas), non podemos engadir outro que represente X (abscisas) fronte a Y (ordenadas), pois ocuparían ambos a mesma cela dentro da matriz de diagramas de dispersión.

Importancia
Esencial

Tipo de proba
Test implementado en JUnit.

Descrición
Xa foi comprobado na avaliación heurística de RF11.

Resultado
Correcto.

RNF05

Título
Entrega dentro de prazo

Descrición
Débese entregar unha versión funcional e documentada antes do día 10 de

Xullo de 2015, ás 14:00 horas, pois é o momento no que remata o prazo de entrega.

Importancia
Esencial

A matriz de trazabilidade que relacionaría cada requisito cun tipo de proba (heurística ou test en JUnit) é a que se presenta a continuación:

	Heurist	Avaliar	TestRF01_1	TestRF01_2	TestRF02_1	TestRF02_2	TestRF0304	TestRF06_1	TestRF06_2	TestRF07	TestRF08_1	TestRF08_2	TestRF09	TestRF10	TestRF2627	TestRF2628	TestRF262930	TestRF2631
RF01			X	X														
RF02					X	X												
RF03							X											
RF04							X											
RF05	X																	
RF06								X	X									
RF07										X								
RF08											X	X						
RF09													X					
RF10														X				
RF11	X																	
RF12	X																	
RF13	X																	
RF14	X																	
RF15	X																	
RF16	X																	
RF17	X																	
RF18	X																	
RF19	X																	
RF20	X																	
RF21	X																	
RF22	X																	
RF23	X																	
RF24	X																	
RF25	X																	
RF26															X	X	X	X
RF27															X			
RF28																X		
RF29																	X	
RF30																	X	
RF31																		X
RF32	X																	
RC01		X																
RD01	X																	
RNF01			X	X														
RNF02							X											
RNF03	X																	
RNF04	X																	
RNF05																		

Figura 5.19: Matriz de trazabilidade

Capítulo 6

Conclusións e posibles ampliacións

Conclusións e posibles ampliacións

Apéndice A

Manuais técnicos

Manuais técnicos: en función do tipo de Traballo e metodoloxía empregada, o contido poderase dividir en varios documentos. En todo caso, neles incluirase toda a información precisa para aquelas persoas que se vaian a encargar do desenvolvemento e/ou modificación do Sistema (por exemplo código fonte, recursos necesarios, operacións necesarias para modificacións e probas, posibles problemas, etc.). O código fonte poderase entregar en soporte informático en formatos PDF ou postscript.

Apéndice B

Manuais de usuario

Manuais de usuario: incluirán toda a información precisa para aquelas persoas que utilicen o Sistema: instalación, utilización, configuración, mensaxes de erro, etc. A documentación do usuario debe ser autocontida, é dicir, para o seu entendemento o usuario final non debe precisar da lectura de outro manual técnico.

Apéndice C

Licenza

Bibliografía

[1] Weka 3 - Data Mining with Open Source Machine Learning Software in Java. Sitio web *http://www.cs.waikato.ac.nz/ml/weka/*.

[2] Proxecto JFreeChart. Sitio web *http://www.jfree.org/jfreechart*.

[3] Formato de Archivo Atributo-Relación (ARFF). Información dispoñible en *http://www.cs.waikato.ac.nz/ml/weka/arff.html* [Última consulta: 10/07/2015].

[4] Java. Sitio web *https://www.java.com/en/*.

[5] Tarifasgasluz. Precio del kWh en España. Sitio web *http://www.endesaonline.com/es/empresas/dual/empresas/4/precios/index.asp* [Última consulta: 10/07/2015].

[6] Electricity usage of a Laptop or Notebook. Sitio web *http://energyusecalculator.com/electricity_laptop.htm* [Última consulta: 10/07/2015].

[7] Scrum. Definicións extraídas de *http://es.wikipedia.org/wiki/Scrum* [Última consulta: 10/07/2015].

[8] Acunote. Sitio web *http://www.acunote.com/*.

[9] GitHub. Sitio web *https://github.com/*.

[10] JUnit. Sitio web *http://junit.org/*.

[11] Lumzy. Sitio web *http://www.lumzy.com/*.

[12] IEEE-STD-830-1998: Especificaciones de los requisitos del software. Ler máis *http://www.ctr.unican.es/asignaturas/is1/IEEE830_esp.pdf* [Última consulta: 10/07/2015].

[13] Buschmann, Frank. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley.

Glosario

Sesión
: Interacción co sistema e as súas funcionalidades. Inclúe a importación duns datos, o seu procesado, filtrado e visualización.

Experimento
: Ver sesión.

Diagrama de dispersión
: Diagrama matemático que fai uso das coordenadas cartesianas para amosar os valores de dúas variables para un conxunto de datos. Cada punto no diagrama referencia un valor ao longo do eixo de ordenadas e outro ao longo do eixo de abscisas.

Modelo
: Estrutura e contido dos datos cos que se traballa no experimento. Abrangue tanto os tipos dos atributos coma os seus valores dentro de cada entrada, así coma o nome do conxunto de datos, ou a marca do atributo que actúa de índice temporal.

Filtro
: Ferramenta configurable que en función dos seus parámetros pode converter, a partir dun modelo A de entrada, un modelo B de saída.

Visualización
: Presentación gráfica da relación, neste caso baixo a forma de diagramas de dispersión.

Reprodución
: Activación dunha visualización para que cambie o seu estado ao longo do tempo, neste caso en función dun índice ou variable temporal.

Índice temporal
: Atributo do modelo que representa a orde na que os datos foron tomados, foron detectados, queren ser priorizados ou simplemente se desexan amosar. Se non se define, asúmese como índice temporal a orde das entradas do modelo.

Entrada
: Ver instancia.

Instancia
: Vector de datos que contén un valor para cada atributo do modelo. Pode conter valores nulos.

Atributo
: Cada unha das variables coas que traballa o modelo.

Atributo numérico
: Atributo que só pode tomar valores numéricos.

Atributo de tipo data
: Atributo que só pode tomar valores de tipo data.

Atributo de tipo string
: Atributo que pode tomar calquera valor en forma de cadea de caracteres.

Atributo nominal
: Atributo que só pode tomar unha serie de valores concretos.

Relación
: Ver modelo.

Reprodución
: Visualización dinámica da relación, é dicir, presentación dunha relación na que cada entrada se visualiza nun momento do tempo determinado.

Scatterplot
: Diagrama de dispersión.

Comando
: Entidade que representa unha orde para o sistema. Contén información sobre o artefacto ao que afecta, o tipo de orde que se expresa e os parámetros necesarios para a súa execución.