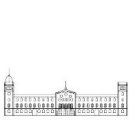

Treball fi de carrera

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Facultat de Matemàtiques
Universitat de Barcelona

Migració d'una plataforma de simulació de xarxes de sensors a *Repast*

Javier Morales Matamoros

Directora: *Meritxell Vinyals Salgado*
Tutor: *Jesús Cerquides Bueno*
Realitzat a: *Departament de Matemàtica Aplicada i Anàlisi. UB*
amb: *l'Institut d'Investigació en Intel·ligència Artificial*

Barcelona, 30 de juny de 2008

A la meva mare, pilar essencial de la meva vida. Sense tú no seria qui soc, t'ho dec tot.

Agraïments

En primer lloc he d'agrair la constant ajuda, constància i comprensió de la Meritxell Vinyals, una persona amb solucions per a tot. Sens dubte una de les millors professionals que he conegut, aquest projecte no hauria sigut possible sense ella.

Agraeixo també l'oportunitat que m'ha donat en Jesús Cerquides donant-me accés a aquest projecte. Sens dubte és un dels professors que més m'ha marcat a la meva vida acadèmica, és d'aquelles persones que fan que t'agradi allò que explica.

Tota la meva gratitud a l'equip de l'Institut d'Investigació en Intel·ligència Artifical, en especial al Jar i al Marc, per donar-me el seu recolzament i proporcionar-me tota l'ajuda que he necessitat.

Gràcies també amb tot el meu cor a la meva xicota Silvia, que ha estat sempre al meu costat durant tota la carrera. Sense els teus ànims, comprensió i consells mai no hauria arribat fins aqui.

Finalment gràcies a tots els meus companys de carrera i aquells amics que han estat al meu costat durant la mateixa.

"La vida de cada home és un camí a si mateix, l'assaig d'un camí, l'esboç d'una sendera."

Hermann Hesse

Índex

Índex de figures

Índex de taules

1 Introducció i motivació

Les xarxes de sensors són xarxes formades per dispositius amb diferents capacitats de còmput i percepció que estan distribuïts en un àrea i col·laboren per produir una informació amb significat global a partir de les informacions sense processar obtingudes a nivell individual [1] [2].

L'interès de la comunitat científica per les xarxes de sensors com a domini d'aplicació i com a font de reptes ha crescut enormement en els últims anys. L'avanç tecnològic en aquest camp ha permès el desenvolupament de sensors cada cop de menor tamany i amb un preu més reduït i han fet factible el desplegament de xarxes de sensors reals. Això ha permès detectar un gran nombre d'aplicacions per aquestes xarxes. Exemples d'aquesta aplicació són xarxes per a la seguretat i vigilància, monitorització de l' hàbitat i l'entorn, aplicacions biomèdiques, creació d'espais intel·ligents (*Smart spaces*) o aplicacions en la robòtica distribuïda (*Robocup Rescue*).

Tot i la facilitat del seu desenvolupament i a la disponibilitat de plataformes comercials disponibles a un preu accessible (`http://www.sunspotworld.com`) els investigadors segueixen utilitzant plataformes de simulació per estudiar aquestes xarxes. Això és degut a que el desplegament d'una xarxa amb un nombre elevat de sensors suposa una elevada despesa econòmica i els investigadors queden limitats a petites xarxes per a les proves. A més a més, la inevitable variabilitat dels resultats obtinguts, que depenen de factors incontrolables, i la impossibilitat d'abstraure's de detalls de baix nivell fan que els investigadors els sigui molt més fàcil provar algoritmes, com a mínim en una fase prèvia, en una simulació on poden controlar els paràmetres i el nivell d'abstracció desitjat.

1.1 El simulador *SNS*

El grup de recerca del projecte IEA (Institucions Electròniques Autònomes, TIN2006-15662-C02-0) de l'Institut d'Investigació en Intel·ligència Artificial de Bellaterra (IIIA-CSIC) ha optat per aquest domini i la simulació alhora de presentar un demostrador que permeti portar a terme experiments i demostracions als membres del grup. Al projecte IEA s'estudien les xarxes de sensors des de la perspectiva dels sistemes multi-agents. En un projecte previ [3] es va realitzar el disseny i la implementació del nucli d'una plataforma que permetia l'instanciació d'aquestes xarxes i complia els requisits funcionals per a estudiar-les utilitzant sistemes multi-agents.

El resultat va ser el simulador *SNS*, una plataforma basada en events per a estudiar les xarxes de sensors des de la perspectiva dels sistemes multi-agents. Aquest simulador ofereix una sèrie de components (fenòmens, agents, sensors) amb els quals podem modelar xarxes de sensors fàcilment.

La plataforma, implementada en *Java*, de codi lliure i orientada a objectes, està composada, entre altres coses, de:

- **Un model** que encapsula tots els elements de la simulació, a més de realitzar la seva construcció i inicialització.
- **Un planificador d'events** que s'encarrega de marcar el temps de la simulació i d'executar les accions de la mateixa al seu instant indicat. Al *SNS* els canvis en la

simulació es provoquen mitjançant el processament d'un event, i el planificador és el que s'encarrega de que aquest event arribi a l'objecte indicat que pot processar-lo.

- **Una interfície gràfica** on es mostren els diferents elements que formen part de la simulació. Gracies a aquesta podem veure visualment alguns dels resultats que donen les simulacions implementades.
- **Una taula** que mostra a la interfície gràfica del simulador cadascun dels events que s'executen a la simulació. Mitjançant la mateixa podem accedir a les propietats de cadascun dels events que provoquen els canvis a la simulació.
- **Un model de dades** que guarda en temps d'execució informació actualitzada sobre diverses propietats d'alguns elements especificats per l'usuari.

1.2 El simulador *Repast*

Repast [4] és un projecte de la *Universitat de Chicago*, una plataforma general de simulació basada en agents, implementada en *Java*, de codi lliure i orientada a objectes. Aquest integra tota una sèrie de funcionalitats per a la simulació: eines d'interfície gràfica per a visualitzar la simulació, eines per a crear estadístiques de la simulació, multi-plataforma, etc. A més a més també té disponibles tota una sèrie de paquets que proporcionen funcionalitats addicionals i més específiques a la plataforma:

- Eines de modelatge de xarxes socials.
- Suport per a sistemes d'informació geogràfica (GIS).
- Llibreries per algorismes genètics.
- Eines per a la construcció de gràfiques.
- Simulacions amb i sense interfície gràfica (*gui* i *batch* respectivament).
- Xarxes neurals...

A part de proporcionar totes aquestes eines i funcionalitats una altra característica de *Repast* és que és un plataforma comú per a la simulació de sistemes multi-agents que posseeix una important comunitat d'usuaris.

Al igual que el simulador *SNS*, aquest està composat, entre altres coses, de:

- **Un model** que conté tots els elements que hi formen part de la simulació.
- **Un planificador d'events** que controla el temps de la simulació i l'execució de les accions que canvien l'estat de la mateixa.
- **Un sistema de displays** que permet dibuixar els elements de la simulació especificats per l'usuari.
- **Un sistema de gràfiques** que ens permet crear gràfiques que representin els valors de diferents propietats d'alguns objectes de la simulació.

1.3 Objectius i motivació del projecte

L' objectiu d'aquest projecte és migrar la plataforma de simulació per a xarxes de sensors anomenada *SNS* [3], com a un paquet per a un entorn de simulació basat en agents com és *Repast*. Així la finalitat d'aquest projecte és proporcionar un nou paquet, *Repast-SNS*, amb la mateixa funcionalitat del *SNS* completament integrat en *Repast-SNS* i que serà accessible per a tota la seva comunitat.

També es requereix que aquest paquet sigui capaç d'executar qualsevol simulació prèviament implementada per a ésser executada al *SNS*, fent les dues plataformes compatibles. En conseqüència, el disseny de la migració de la plataforma també ha de considerar la migració d'exemples de *SNS* a *Repast-SNS* i viceversa amb els mínims canvis possibles.

La motivació del projecte radica en la necessitat d'oferir a la comunitat científica un simulador de xarxes de sensors que utilitzi com a base la plataforma *Repast*, ja que un cop feta la migració haurem obtingut un nou simulador que oferirà eines de modelatge per a xarxes de sensors, a més de les eines que *Repast* ens ofereix per defecte (GIS, algorismes genètics, gràfiques, xarxes neurals, etc. . .)

A més a més el producte resultant serà accessible a tota la comunitat que utilitza *Repast*, ja que al estar basat en aquest la manera d'utilitzar la nova eina serà molt similar a la manera en que els usuaris utilitzaven el simulador original (el *Repast*).

1.4 Organització de la memòria

La memòria està organitzada en les següents seccions:

- **Capítol 1**: Es fa una introducció de les xarxes de sensors i les dues plataformes que formaran part del projecte, el *Repast* i el *SNS*
- **Capítol 2**: Es detallen els requeriments llistats pel projecte.
- **Capítol 3**: Es presenta l'estudi de viabilitat del projecte així com una planificació temporal de les tasques a realitzar.
- **Capítol 4**: Es realitza un estudi dels components necessaris de la plataforma *Repast* per a poder realitzar la migració.
- **Capítol 5**: S'exposa l'arquitectura, disseny i implementació de *Repast-SNS*.
- **Capítol 6**: Es detallen exemples de migració entre la plataforma *SNS* i *Repast-SNS*.
- **Capítol 7**: S'exposen les conclusions del projecte, que repassen els objectius assolits.

2 Requeriments

Un cop contextualitzat el projecte i mostrada la motivació del mateix, és el moment d'establir quins són els requeriments que s'han d'acomplir per tal d'assolir els objectius esmentats en l'apartat anterior. Aquests requeriments han estat formulats a partir de les necessitats dels membres del projecte IEA (Institucions Electròniques Autònomes) de l'Institut d'Investigació en Intel·ligència Artificial (IIIA-CSIC). Els hem distribuït en dues categories:

- **Funcionals:** Els requeriments funcionals són aquells que fan referència a les funcionalitats que haurà d'implementar el paquet d'extensió per a xarxes de sensors que es proporcionarà a la plataforma *Repast*.
- **No funcionals:** Els requeriments no funcionals fan referència a les restriccions que s'imposen pel que fan a la implementació en la migració de la plataforma a *Repast*.

2.1 Requeriments funcionals

Des d'un punt de vista funcional, els requeriments llistats són els següents:

- **Migració del planificador (*scheduler*):** Encara que tant el planificador de *Repast* com el de la plataforma *SNS* són basats en events discrets, la definició d'events que utilitza cadascun d'ells és diferent. A més a més, la plataforma *SNS* incorpora un control per a considerar el temps que ha trigat l'execució d'un event per un determinat objecte de la simulació i retardar la resta d'events a partir d'aquell temps. Aquest control no es dóna en *Repast*.
- **Extensió del model de *Repast*:** El model és l'objecte que instancia i especifica els paràmetres de la simulació. Tant les simulacions de *SNS* com les de *Repast* tenen ambdues un model però amb una informació i format diferents. Per a poder aprofitar els objectes del *SNS* que ens permeten modelar xarxes de sensors s'haurà d'afegir al model de *Repast* tota la informació que no estigui continguda en el seu model i migrar la resta.
- **Taula d'events executats:** Tal i com proporcionava la plataforma *SNS* original, el paquet *SNS* per a *Repast* ha de proporcionar integrat a l'entorn gràfic una taula que mostri els events executats i la seva informació associada. Així l'usuari podrà accedir a tots els events que s'executen a la simulació i a les seves propietats.
- **Integració amb el sistema de displays de Repast:** *Repast* disposa d'un sistema de *displays* propi, oferint diversos objectes i mètodes per a poder representar gràficament els elements de la simulació. Als usuaris del paquet *SNS* per a *Repast* se'ls ha de facilitar l'accés i ús a aquestes eines gràfiques.

  La plataforma original *SNS* ja proporciona un sistema per a representar objectes gràfics a la simulació, així doncs caldrà substituir totes les classes referents a la interfície gràfica de la antiga plataforma i utilitzar-ne d'equivalents en *Repast*.
- **Facilitat d'ús en generació de les gràfiques:** *Repast* proporciona llibreries per a la generació de gràfiques durant la simulació. En la plataforma *SNS* es genera un model de dades que inclou els valors dels atributs que l'usuari de la simulació ha identificat com a destacats pel model. El paquet *SNS* per a *Repast* haurà de

proporcionar una capa que faciliti la generació de gràfiques utilitzant les llibreries d'aquest aprofitant el model de dades de la plataforma *SNS*.

2.2 Requeriments no funcionals

Haurem de tenir en compte els següents requeriments no funcionals.

- **Codi lliure:** Per a que el simulador resultant sigui accessible a tota la comunitat científica és necessari que sigui alliberat sota una llicència de *software* lliure. A més, el *Repast* original utilitza una llicència d'aquest tipus, per tant no volem afegir cap restricció.
- **Escrit en *Java*:** Tant el *Repast* original com el simulador *SNS* estan escrits en *Java*, per tant la nostra eina haurà d'estar escrita en el mateix llenguatge.
- **Documentat:** Podrem utilitzar eines com el *Javadoc* per generar una documentació de cadascuna de les classes del simulador resultant. L'objectiu és obtenir una *API* suficientment documentada per a poder utilitzar-la amb facilitat.

3 Estudi de viabilitat i planificació del projecte

Per portar a terme el nostre projecte primer hem de fer un estudi previ de viabilitat. Aquest estudi de viabilitat conté dos apartats: un estudi de si el projecte és factible en l' aspecte tècnic i una planificació temporal de les tasques previstes. La planificació temporal del projecte ens permetrà valorar si aquest és pot portar a terme dins del temps especificat.

3.1 Estudi de viabilitat

En el projecte s'han realitzat dos estudis de viabilitat: el primer sobre *Repast Simphony*, una nova versió de Repast que permet la programació de simulacions utilitzant un entorn gràfic, i el segon sobre la última versió de *Repast*. A continuació es presenten les conclusions d'aquests estudis de viabilitat i els motius pels quals finalment es va optar per migrar *SNS* a *Repast* i no a la versió *Repast Simphony*.

3.1.1 Estudi de viabilitat per a *Repast Simphony*

El *Repast Simphony* és la versió més actual alliberada per l'equip que va desenvolupar *Repast*. És una plataforma basada en els mateixos principis que aquest, també implementada en *Java*, orientada a objectes i de codi lliure. La nova característica principal de *Repast Simphony* és que en aquesta nova plataforma es permet la específicació de simulacions mitjançant un entorn gràfic. Així el usuari de la plataforma pot programar noves simulacions mitjançant eines completament gràfiques. A més a més, la nova plataforma també ofereix més entorns de visualització i integració amb eines estadístiques (*R*,*MatLab*,...).

Les proves d'integració bàsiques amb aquesta plataforma però van presentar molts problemes degut a que ha estat encara poc testejada i a que no comparteix la mateixa estructura que la versió *Repast*. Així els manuals per a la plataforma són quasi inexistents així com els exemples de simulació que proporciona. A més a més, la migració d'exemples de *Repast* a *Repast Simphony* no ha estat especificada i no se sap si serà possible migrar les existents implementacions.

Per tant degut a la seva inestabilitat, a la dificultat de les proves inicials i a que no queda clar el seu impacte en la comunitat (els desenvolupadors continuen treballant amb *Repast*) es va decidir després d'aquest estudi de viabilitat no migrar *SNS* a aquesta plataforma i realitzar l'estudi de viabilitat sobre *Repast*.

3.1.2 Estudi de viabilitat per a *Repast*

El *Repast* és una plataforma consolidada entre la comunitat científica i que té molt més temps que el *Repast Simphony*, per tant ha sigut molt testejada, existeixen un gran nombre de manuals d'usuari i un gran nombre d' exemples de simulacions.

En principi la migració a *Repast* sembla viable ja que presenta la següents característiques:

- És de codi lliure, per tant podem modificar-lo i adaptar-lo a les nostres necessitats.
- Està implementat en *Java*, al igual que el simulador *SNS*.
- Està basat en events, igual que el *SNS*.
- Està format per una arquitectura basada en un model i un planificador d'events, al igual que el simulador *SNS*.

Per a veure quin és el funcionament del mateix i com es generen les simulacions, es fan diverses proves creant exemples senzills que funcionen a la perfecció, per tant comprovem que és una eina sòlida i molt semblant al *SNS*. Per tots aquests motius entenem que, en l'aspecte tècnic, la migració a *Repast* és possible i que per tant el projecte és viable.

3.2 Planificació del projecte

En aquest apartat és llisten les tasques a realitzar per assolir els requeriments del projecte així com una estimació de les hores destinades a cada tasca. Les tasques s'han separat en tres categories: tasques d' anàlisi/estudi, les tasques de disseny/implementació i les tasques de documentació.

3.2.1 Tasques d'anàlisi i estudi

Les tasques d'anàlisi i estudi del projecte inclouen les reunions amb el tutor del projecte i l'equip del projecte IEA, així com el temps dedicat a l'estudi del funcionament del simuladors *Repast* i *SNS*.

Tasques d'anàlisi i estudi	
Reunions de seguiment	**32h**
- Reunions amb el tutor del projecte i els membres del projecte IEA.	32h
Requeriments del projecte	**3h**
- Recopilació de tots els requeriments que ha de complir el simulador resultant.	3h
Estudi del Repast *Simphony*	**25h**
- Aprende a crear exemples d'una simulació amb el Repast Simphony.	4h
- Estudi bàsic del scheduler de Repast Simphony (*IActions*, *Schedule*, etc. . .).	21h
Estudi del planificador del *SNS*	**6h**
- Estudi dels aspectes del disseny i implementació generals del *SNS* que s'han de considerar per la migració.	6h
Estudi del planificador de Repast *(scheduler)*	**24h**
- Estudiar què contenen i com funcionen les accions de Repast.	3h
- Investigar com es gestionen les cúes d'accions del planificador.	2h
- Com s'executen les accions de Repast.	1h

- Estudiar les eines que utilitza Repast per a crear una nova acció que executi l'algorisme que defineix l'usuari.	6h
- Investigar com s'adjunten les accions al planificador de Repast.	3h
- Aprendre l'algorisme de planificació d'accions de Repast. Necessari per a poder modificar-la i adaptar-la a les necessitats del nou simulador.	7h
Investigació del model del *SNS*	**6h**
- Estudiar els mètodes que utilitza el model del *SNS* per a configurar i inicialitzar la simulació (*setup(), init()*).	2h
- Què és i per a què serveixen el *field* i les *Component Factories*.	2h
- Estudiar la manera en que es fixa i guarda l'estat del model i es preparen els elements de la simulació en funció d'aquest.	2h
Estudi del model de Repast	**8h**
- La interfície *SimModel*. Qué és i quins mètodes bàsics ha d'implementar qualsevol model de simulació.	1h
- Quines propietats i mètodes conté la implementació parcial del model (*SimModelImpl*) i què fa cadascun d'ells.	4h
- Quins mètodes ha d'implementar el usuari al seu model extès de *SimModelImpl* i què ha de fer cadascun d'ells.	3h
Estudi del controlador i l'entorn gràfic de Repast	**6h**
- Investigar el controlador de Repast, que s'encarrega de controlar i inicialitzar la seva interfície gràfica.	3h
- Què és l'inicialitzador de la simulació (*SimInit*) i com funciona.	3h
Investigació del sistema de displays de Repast	**2h**
- Estudiar com es creen i quins tipus de surfaces existeixen per a Repast, així com la manera en que es representen.	0.5h
- Investigar els displays de *Repast* i la manera en que guarden i representen objectes de tipus *Drawable*.	0.5h
- Esbrinar com es creen els objectes de tipus *Drawable* i es representen als displays.	1h

Investigació del sistema de gràfiques de Repast	**2h**
- Estudiar les classes que implementen les gràfiques del Repast. És necessari per a adaptar les gràfiques per a les nostres necessitats.	1h
- Investigar com funcionen les seqüències de Repast i la manera en que es representen a les gràfiques.	1h
TOTAL	**114h**

Taula 1: Planificació del projecte

3.2.2 Tasques de disseny i implementació

Inclouen el temps dedicat al disseny, la implementació i les proves de la sol·lució final *RepastSNS*.

Tasques de disseny i implementació	
Migració del planificador (*scheduler*)	**47h**
- Incorporar la possibilitat de retrassar un event al planificador depenent del temps d'ocupació del objecte destinatari.	6h
- Afegir la possibilitat de planificar accions amb paràmetres.	18h
- Execució davant del procès d'un event SNS del mètode *process()* del destinatari amb l'event com a paràmetre.	7h
- Modificació dels mètodes per a planificar events disponibles al model per a que cridin els mètodes de *Repast*.	10h
- Realitzar les modificacions pertinents per a poder planificar una acció per a que s'executi al mateix moment en que s'afegeix al planificador.	6h
Extensió del model de Repast	**35h**
- Afegir variables i constants pròpies del model SNS.	2h
- Afegir mètodes de construcció i inicialització pels elements del model *SNS*.	6h
- Afegir crides als nous mètodes afegits des dels mètodes d'inicialització i construcció de *Repast* (*init()*, *setup()*, *build()*).	8h
- Integrar els mètodes de construcció i incialització de la simulació del nou model amb la funcionalitat existent a *Repast* de poder resetejar la simulació.	7h
- Implementar la interficie *SimulationListener* per a que el model de *Repast* pugui rebre events i processar-los.	6h
- Eliminar totes les referències del model antic a l'entorn gràfic i canviar les crides al generadors de nombres al·leatoris propis del *SNS* per les de *Repast*.	6h

Taula d'events executats	**12h**
- Aconseguir que el Repast mostri al seu entorn gràfic la taula d'events executats i que aquesta es mostri per pantalla a l'iniciar-lo.	6h
- Fer que s'enviin tots els events executats a la taula d'events i aquesta els mostri en una nova cel·la.	6h
Integració amb el sistema de displays de Repast	**24h**
- Afegir mètodes per a gestionar la incorporació de una o diverses *surfaces* al sistema de displays i per a poder afegir amb facilitat *displays* a les *surfaces* corresponents.	12h
- Dissenyar un sistema per a poder afegir, recuperar i gestionar llistes d'objectes de tipus *Drawable* d'un *display*. Mitjançant això podrem afegir i esborrar elements gràfics als displays en temps d'execució.	12h
Facilitat d'ús en generació de les gràfiques	**21h**
- Crear un model de dades similar al del *SNS* però només amb les propietats i mètodes necessaris per al nostre cas i afegir-lo al model de *Repast*.	6h
- Crear un nou tipus de seqüència que guardi l'objecte i la propietat del model de dades a la qual representa.	4h
- Incorporar un sistema per a generar i mostrar fàcilment un gràfic que representi una llista de propietats especificades per l'usuari.	6h
- Oferir una sèrie de gràfiques predefinides com per exemple els events executats cada cert interval de temps.	5h
Exemple de simulació	**14h**
- Estudiar com funciona un exemple concret del *SNS* per a poder migrar-lo.	12h
- Migració de l'exemple del simulador *SNS* per a executar-lo en la nova plataforma *RepastSNS*.	2h
TOTAL	**153h**

Taula 2: Tasques de disseny i implementació

3.2.3 Tasques de documentació

Les tasques de documentació consisteixen en redactar la memòria del projecte i els manuals d'usuari.

Tasques de documentació	
Redacció de la memòria	**115h**
- Aprendre a utilitzar LaTeX.	10h
- Estructuració i redacció de la memòria final del projecte.	105h
TOTAL	**115h**

Taula 3: Tasques de documentació

Sumant el temps invertit en les tres tasques principals del projecte, que són l'estudi, el disseny i la implementació i la redacció de la memòria del mateix, obtenim el següent resum d'hores.

Hores totals del projecte	
- Tasques d'anàlisi i estudi.	114h
- Tasques de disseny i implementació	153h
- Tasques de documentació.	115h
TOTAL	**382h**

Taula 4: Total d'hores del projecte

Un cop estimades les hores totals del projecte veiem que entren dins del temps estipulat per al mateix, per tant podem valorar que si som capaços de seguir el *planning* podem tirar cap endavant el projecte.

3.3 Cost del projecte

A partir de la planificació temporal especificada en els apartats anteriors podem calcular el cost econòmic del projecte, agrupant les tasques segons el personal que ha de realitzar-les i aplicant un cost econòmic en €/hora.

A més de les hores invertides en la realització del projecte hem de tenir en compte el cost d'altres recursos, com per exemple llicències de software. Tant el *SNS* com el *Repast* són programari lliure i el projecte es realitza en *Java* sota un sistema operatiu *GNU-Linux*, per tant la despesa en aquest aspecte és inexistent.

La estimació del cost del projecte en recursos humans la podem veure a la següent taula:

Perfil	**Hores**	**Dotació**	**Total**
Programador	300h	30 €/h	9.000 €
Analista	80h	50 €/h	4.000 €
Director	20h	70 €/h	1.400 €
Cost en recursos humans			14.400 €

Taula 5: Estimació del cost en recursos humans

Si afegim el cost en *hardware* i despeses corrents com per exemple llum, aigua, etc. . . obtenim una estimació total del cost del projecte.

Concepte	**Cost**
Recursos humans	14.400 €
Hardware	600 €
Despeses corrents	200 €
Cost total del projecte	**15.200 €**

Taula 6: Estimació del cost total del projecte

4 Estudi del simulador *Repast*

Per a acomplir els requisits plantejats haurem de fer un estudi previ del *Repast*. Un cop sabem quin és el funcionament intern del mateix podrem dissenyar una sol·lució per a adaptar-lo i donar-li les característiques desitjades.

4.1 L'arquitectura de *Repast*

El *Repast* té una arquitectura basada en el patró MVC (Model-Vista-Controlador) separant la capa de negoci de la interfície gràfica i el controlador. Això fa que l'aplicació sigui modular i reutilitzable, permetent poder executar simulacions amb o sense interfície gràfica, ja que la capa de negoci no es veu afectada al estar separada de la capa de vista.

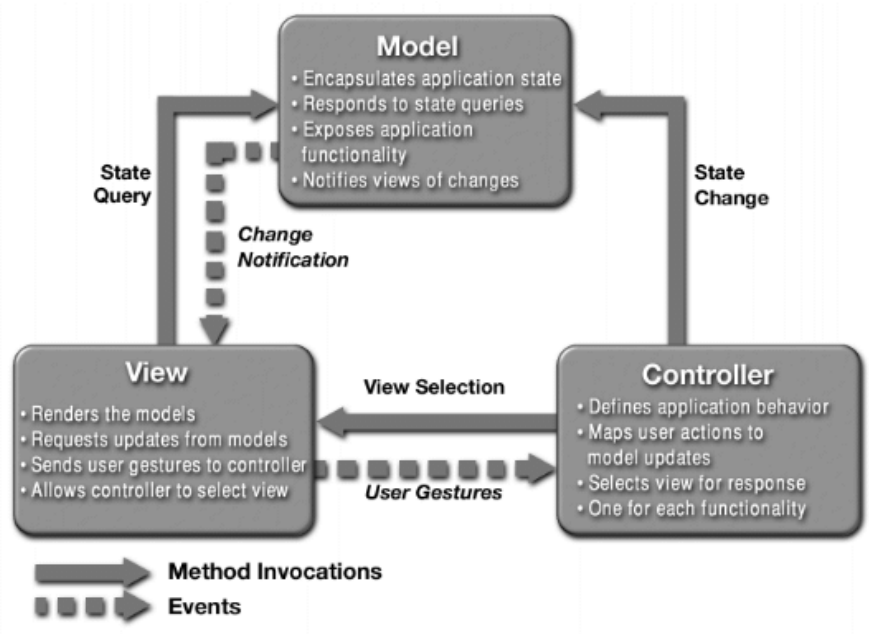

Figura 1: Patró model-vista-controlador

Per a modificar el *Repast* ens farà falta estudiar cinc nuclis bàsics:

- Planificador (*Scheduler*).
- Model.
- El controlador i la interfície gràfica de *Repast*.
- Sistema de *displays*.
- Sistema de gràfiques.

Veiem mitjançant la figura 2 com els diferents elements a estudiar es troben distribuïts a l'arquitectura del *Repast*:

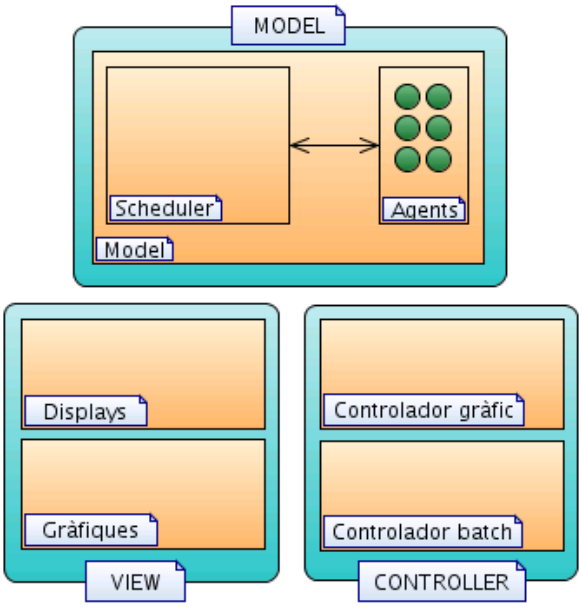

Figura 2: Distribució dels elements de *Repast* a la seva arquitectura

La part "Model" conté els elements que implementen la capa de negoci del *Repast*, aquests són el model de la simulació, que conté tots els elements de la mateixa, i el planificador, que s'encarrega d'executar les accions que modifiquen l'estat de la simulació i de marcar el temps de la mateixa. Podem veure que el model de la simulació també conté els agents, els quals poden executar accions per ordre del planificador (*scheduler*) i també poden demanar al mateix que planifiqui accions que canviaran l'estat de la simulació.

La part "View" del *Repast* conté diferents elements que permeten captar visualment l'estat actual de la simulació. Aquests són, principalment, el sistema de *displays*, que mostra gràficament els diferents objectes que formen part de la simulació, i el sistema de gràfiques, que ens permet representar dades estadístiques sobre l'estat de la mateixa.

Pel que fa a la part "Controller", el *Repast* té un controlador bàsic que porta el flux de la simulació (play, stop, pausa, etc...). D'aquest controlador bàsic extenen dos controladors, un amb interfície gràfica per poder utilitzar-lo amb botons, i un altre de tipus *"batch"* sense interfície gràfica.

A continuació passem a estudiar de cadascun dels elements de l'arquitectura de *Repast* les seves parts necessàries per a poder realitzar la migració satisfactòriament.

4.2 El planificador

A *Repast*, com a tots els simuladors basats en events, el planificador d'events o *scheduler* és l'encarregat de marcar el ritme de l'execució i controlar el temps de la simulació. Aquest s'encarrega d'executar els events corresponents per ordre del temps estipulat i controlar el temps de simulació actualizant-lo al temps del event executat. Estudiarem el planificador per a poder dissenyar una sol·lució eficaç i afegir les característiques que ens demanen als requeriments.

4.2.1 Les accions

A *Repast* els events són anomenats accions, les quals són la unitat bàsica d'execució del *scheduler*. Qualsevol canvi que volem realitzar a la simulació s'ha de fer mitjançant la construcció d'una acció que executarà un mètode d'un objecte de la simulació a un temps determinat. La classe abstracta *BasicAction* (figura 3) és la que s'ofereix en *Repast* per a implementar accions.

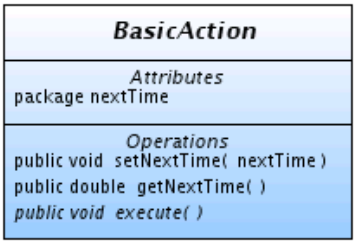

Figura 3: Classe BasicAction del Repast, representant una acció bàsica.

Tota acció ha de tenir una propietat que indiqui el moment determinat en què ha d'ésser executada. En aquest cas la propietat que ens indica el moment de la simulació en que portar a terme l'execució de l'acció és el *nextTime*, que es pot fixar i obtenir mitjançant els seus mètodes *setNextTime()* i *getNextTime()*. El mètode abstracte *execute()* és el que contindrà el codi que s'executarà amb l'acció.

4.2.2 Com planificar accions

Per a planificar accions a *Repast* ho hem de fer mitjançant el planificador o *scheduler*, que es proporciona a la classe *Schedule*. Al planificador (figura 4) es proporcionen tota una sèrie de mètodes per a poder planificar accions especificant diferents ordres, duracions, etc...

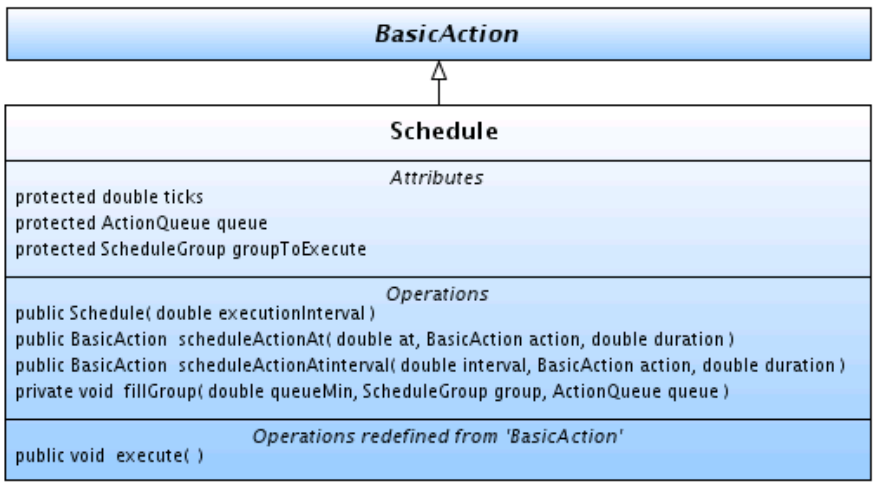

Figura 4: Classe Schedule del Repast.

Per a migrar el planificador *SNS* però els mètodes que més ens interessa estudiar són els següents:

- *scheduleActionAt(double at, BasicAction action, double duration)*: Rep per paràmetre una *BasicAction* per a planificar-la a un moment determinat de la simulació i amb una duració determinada (per defecte 0). Aquesta acció només s'executarà un cop.
- *scheduleActionAtInterval(double interval, BasicAction action, double duration)*: Rep per paràmetre una *BasicAction* per a planificar-la a un moment determinat de la simulació i amb una duració determinada (per defecte 0). En aquest cas l'acció s'executa diversos cops amb una diferència de temps equivalent a la propietat *interval*, rebuda per paràmetre

La classe té una propietat anomenada *ticks* que guarda el temps actual de la simulació, per tant podem veure que a *Repast* la unitat de temps de simulació és el **tick**, a diferència del *SNS*, on la unitat de temps és el **nanosegon**.

La propietat *actionQueue* és la cua d'accions del *scheduler* i s'encarrega de guardar les accions que es planifiquen fins al moment de la seva execució.

En cada execució el planificador executa una acció, executant el mètode *execute()*. El que es fa en aquest mètode és el següent:

1. Preparar el grup d'execució cridant al mètode *fillGroup()*. El mètode *fillGroup()* és l'encarregat d'omplir el grup d'execució (*groupToExecute*) amb les següents accions a executar en un mateix *tick*.
2. Executar el grup d'execució mitjançant la crida al mètode *groupToExecute.execute()*. El mateix grup s'encarrega d'executar cadascuna de les accions que conté cridant al seu corresponent mètode *execute()*.
3. Replanificar les accions cridant al mètode *reSchedule()* del grup d'execució. El mateix grup d'execució s'encarrega de cridar al mètode *reSchedule()* de cadascuna de les accions que conté passant-lis per paràmetre la cua d'accions del *scheduler* (*actionQueue*). Les accions, al ésser cridades al seu mètode *reSchedule()*, decideixen si es replanifiquen o no.

4.2.3 L'algorisme de planificació

Tot comença en el moment en que creem l'acció i la planifiquem mitjançant els mètodes que l'*scheduler* ens proporciona (*scheduleActionAt(), scheduleActionAtInterval()*, etc...). Una vegada creem l'acció i la enviem a planificar s'executa l'algorisme que veiem al diagrama:

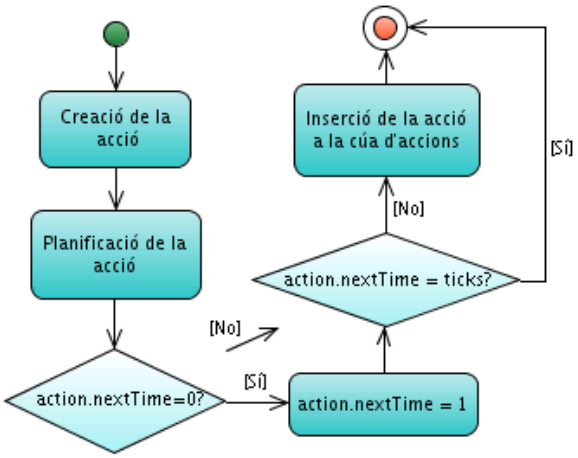

Figura 5: Diagrama de l'algorisme de planificació de Repast

1. Si l'acció es vol executar a temps 0 se li canvia la propietat *nextTime* a 1, tal i com hem comentat anteriorment, per tant s'executarà a temps 1.

2. En cas que l'acció es vulgui executar a l'instant actual es retorna la acció sense planificar-la. Posant un exemple, si l'*scheduler* té temps de simulació *ticks=2* i volem executar la nostra acció a temps 2, no es planificarà i per tant mai no serà executada.

3. En funció del mètode utilitzat per a planificar-la, si un mètode amb execució puntual (*scheduleActionAt()*) o amb interval (*scheduleActionAtInterval()*), es fixa el tipus d'acció per a definir si s'executarà un o diversos cops.

4. Finalment l'acció s'insereix a la cua d'accions del planificador. Quan arribi el moment determinat l'acció s'executarà.

4.2.4 Crear accions

A *Repast* les classes que s' utilitzen internament per a instanciar accions són bàsicament dues: *ActionUtilities* (figura 6) i *ByteCodeBuilder* (figura 7).

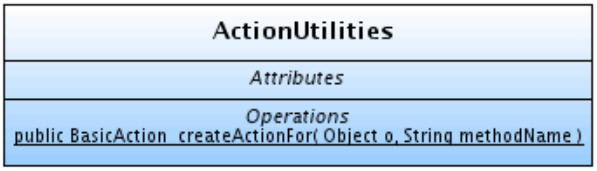

Figura 6: Classe ActionUtilities del Repast.

Aquesta classe conté un mètode anomenat *createActionFor()* al qual se li passa per paràmetre un objecte i el mètode que es vol executar del mateix, retornant una acció (*BasicAction*) que executará el mètode demanat. Imaginem que tenim un objecte de la següent classe hipotètica:

```
1  public DummyClass {
2    public void execMethod() {}
3  }
4
5  DummyClass dummy= new DummyClass();
```

Listing 1: Dummy class

Si volem que el *Repast* executi el mètode *execMethod()* del nostre objecte *dummy* haurem de crear una acció que faci això. Això ho podem fer cridant al mètode estàtic *ActionUtilities.createActionFor()* passant-li per paràmetre l'objecte *dummy* i el mètode a cridar, en aquest cas *execMethod()*.

El mètode *createActionFor()* se serveix d'altres mètodes de la classe *ByteCodeBuilder* per crear i retornar un objecte del tipus *BasicAction*. Al cridar al mètode abstracte *execute()* de l'acció retornada aquesta farà una crida al mètode *execMethod()* del nostre objecte *dummy*. Cal dir que al *Repast* original no és possible cridar a un mètode d'un objecte passant-li arguments per paràmetre.

Un usuari que vulgui crear una acció no necessita saber res més a part del que ja hem explicat sobre la classe *ActionUtilities* però en el nostre cas, que haurem de modificar la manera en que *Repast* crea les accions, hem d'estudiar la classe de recolzament que aquesta utilitza per a crear-les, la *ByteCodeBuilder* (figura 7).

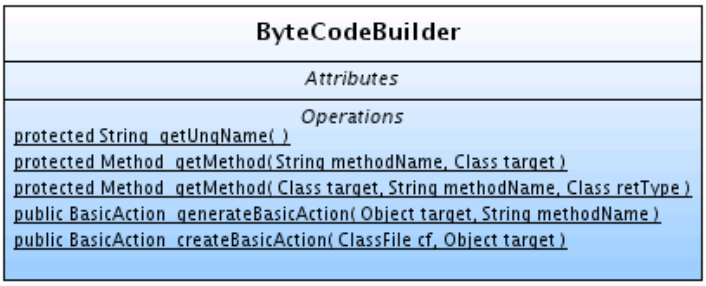

Figura 7: Classe ByteCodeBuilder del Repast.

D'aquesta classe el mètode que més ens interessa saber com funciona és el *generateBasicAction()*. Aquest mètode és cridat pel mètode *ActionUtilities.createActionFor()* per crear l'acció que executi el mètode de l'objecte desitjat, rebent per paràmetre l'objecte (*target*) i el mètode a cridar de l'objecte (*methodName*).

El que fa és, en temps d'execució, fer servir diverses eines per generar dinàmicament un objecte d'una classe que extèn de *BasicAction* i implementa el seu mètode abstracte *execute()*. A dins d'aquest mètode es realitza la crida al mètode del nostre objecte. Per exemple, al cridar al mètode passant-li per paràmetre l'objecte *dummy* i el nom del mètode a cridar, és a dir *"execMethod()"*, l'aspecte de la classe que

```
1  public class BAB_SYNTH_1 extends BasicAction {
2
3    private DummyClass target;
4
5    public BAB_SYNTH_1(DummyClass target) {
6      this.target = target;
7    }
8
9    public void execute() {
10     target.execMethod();
11   }
12 }
```

Listing 2: Acció creada dinàmicament

On l'objecte *target* que la classe rep per paràmetre al constructor és el nostre objecte *dummy*, de la classe *DummyClass*. Aquesta classe per sí mateixa no existeix com a fitxer *BAB_SYNTH_1.java*, sinó que es crea en temps d'execució per a executar el mètode desitjat i després, en cas que s'executi només un cop, es destrueix.

4.2.5 Els grups d'execució

Cada vegada que enviem una acció al planificador de *Repast* aquest la incorpora a una cua d'accions, la qual s'encarrega de guardar-la fins al moment en que ha d'ésser executada. La cua d'accions és proporcionada en una classe anomenada *ActionQueue* i no és més que una classe que conté una llista d'accions i una sèrie de mètodes per a manegar-les.

Per a executar les accions aquestes s'agrupen segons la seva propietat *nextTime*, portant-se d'un cop a executar les següents accions que tenen el mateix instant d'execució, per tant haurem de poder executar les accions en grup. Al *Repast* aquests grups d'execució es proveeixen a la classe *ScheduleGroup* (figura 8), que extèn la classe *BasicAction*.

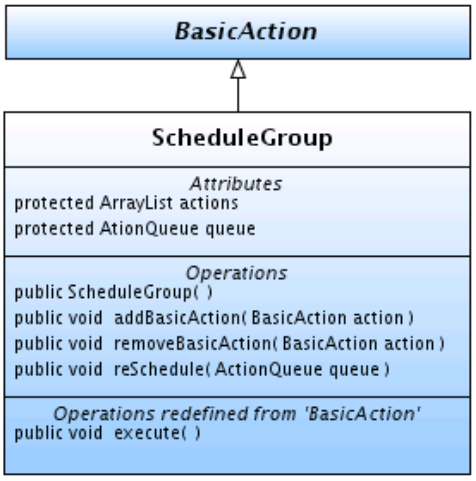

Figura 8: Classe ScheduleGroup, grup d'execució del scheduler.

Quan creem un grup d'execució li hem de passar per paràmetre una cua d'accions, com podem veure al constructor *ScheduleGroup(ActionQueue queue)*. Això és necessari per

replanificar les accions després d'executar-les en cas que s'hagin d'executar més d'un cop, és a dir adjuntar-les un altre cop a la cua d'accions del planificador de la qual provenien.

La classe *ScheduleGroup* conté un mètode anomenat *execute()* que redefineix al de la superclasse *BasicAction* i s'encarrega d'executar totes les accions que conté el grup. Després d'executar les accions del grup totes són replanificades, és a dir es crida el mètode *reSchedule()* del grup d'execució. L'acció mateixa decideix, en funció de propietats internes que han sigut fixades en el moment de la creació, si es torna a planificar o no.

4.3 El model de *Repast*

El model és el "món" virtual que encapsula tots els elements de la simulació. Per a poder fer la integració dels simuladors *SNS* i *Repast* haurem d'estudiar les característiques del model d'aquest últim. Al *Repast* el model és proporcionat en una definició (interfície *SimModel*) i una implementació parcial, (classe *SimModelImpl*).

4.3.1 La definició del model

La definició de lo que és el model de la simulació i els mètodes bàsics que ha de tenir ve donada per la interfície *SimModel* (figura 9).

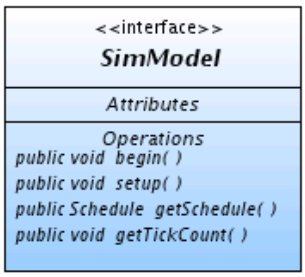

Figura 9: Interfície SimModel de Repast definint el seu model.

Qualsevol model a *Repast* ha d'implementar el mètode *setup()*. Aquest mètode és cridat la primera vegada que el model es carrega i cada vegada que l'usuari para la simulació i posteriorment pressiona el botó de *"setup"*. S'encarrega de preparar el *Repast* per a la simulació.

Un altre mètode important a comentar és el *begin()*. Aquest mètode és cridat quan l'usuari pressiona el botó d'iniciar la simulació. En aquest mètode s'ha d'implementar la inicialització i construcció del model i els *displays*. El model ha de tenir sempre un planificador o *scheduler*. Per a assegurar-se d'això, la interfície ens obliga a retornar un scheduler amb el mètode *getSchedule()*, ja que sense un planificador la simulació no és possible.

Per últim podem veure un mètode *getTickCount()*. Aquest mètode ha de retornar els *ticks* actuals, és a dir el temps actual de la simulació. Més endavant veurem que a les implementacions del model aquest mètode retorna el temps de simulació del controlador, que a la seva vegada agafa el temps de simulació del *scheduler* contingut dins el model del qual conté una referència.

4.3.2 La implementació del model

Una vegada explicada la interfície, per acabar d'explicar com funciona el model de *Repast* hem de comentar la implementació del mateix mitjançant la figura 10:

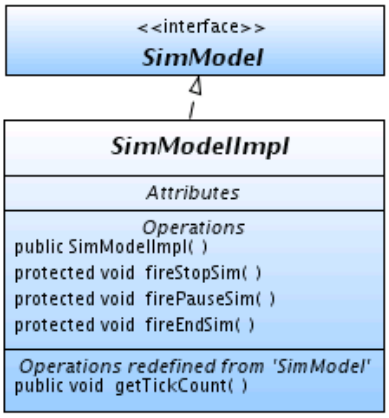

Figura 10: Implementació del model de Repast.

Aquests són els mètodes més importants que hem de saber com funcionen:

- Els mètodes *fireStopSim()*, *firePauseSim()* i *fireEndSim()* s'utilitzen per a enviar a cadascun dels *listeners* del model un event avisant de què la simulació es para, pausa o acaba respectivament. D'aquesta manera els objectes de la simulació es poden assabentar de lo que està passant en cada moment.
- Com a mètodes implementats de la interfície *SimModel* tenim el *getTickCount()*, que retorna el temps actual de la simulació.

Aquesta classe, que només realitza una implementació parcial d'alguns mètodes de la interfície *SimModel*, s'ha construït com a abstracta amb la intenció de deixar la resta de mètodes per a que els implementin els models que extenguin de la mateixa. Així doncs quan un usuari crea un nou model extenent la classe *SimModelImpl* ha d'implementar obligatòriament els següents mètodes:

- El mètode *getInitParam()*, que serveix per a dir al *Repast* quines són les variables que es mostraran a la interfície gràfica per a poder modificar-les en temps d'execució.
- El mètode *setup()*, al qual l'usuari ha de definir quines són les accions que s'han de realitzar per a preparar i inicialitzar el model.
- El mètode *begin()*, que l'usuari ha d'implementar per a definir l'algorisme a executar i els objectes a preparar cada cop que s'inicia la simulació.
- El mètode *getSchedule()*, que ha de retornar una referència al planificador o *scheduler*. Com que el planificador d'events l'ha de crear l'usuari que crea el seu propi model extès d'aquesta classe, mitjançant aquest mètode la superclasse *SimModelImpl* obté la referència al mateix.

4.4 El controlador gràfic i l'iniciador de la simulació

Com que hem d'afegir una taula d'events executats a la interfície gràfica de *Repast*, haurem d'estudiar com funciona el controlador gràfic i l'objecte que s'utilitza per a iniciar la simulació.

4.4.1 El controlador gràfic

Aquest es proveeix en la classe *Controller* (figura 11), que és una varietat que extèn d'un controlador bàsic de la simulació. Estudiem els mètodes necessaris per a poder integrar la nostra taula d'events a la interfície gràfica de *Repast*.

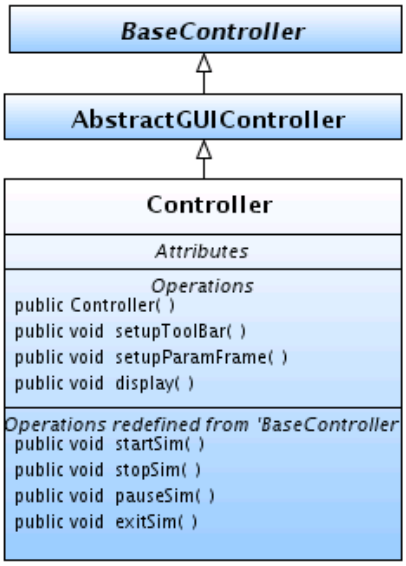

Figura 11: Classe *Controller*, que implementa el controlador de la interfície gràfica.

Aquesta classe extèn de *AbstractGUIController*, que és el controlador bàsic amb interfície gràfica, que la vegada extèn de la base per a qualsevol controlador de simulació, el *BaseController*. Hem estudiat els mètodes que més ens interessen per a poder inserir la taula d'events a l'entorn gràfic de *Repast*:

- *setupToolBar()*: Aquest mètode és cridat cada vegada que s'inicia la interfície gràfica i prepara la barra d'eines del *Repast*, que conté els botons de control de la simulació (inici, pausa, stop, ...), l'etiqueta que mostra els *ticks* de la simulació, etc...
- *setupParamFrame()*: Utilitzat per a crear i mostrar el *frame* que mostra les variables que podran ésser modificades en temps d'execució.
- *display()*: És el mètode que mostra tots els elements de la interfície gràfica de *Repast*.

Als mètodes redefinits de la superclasse *BaseController* (*startSim(), stopSim(), pauseSim()* i *exitSim()*) el controlador gràfic habilita i deshabilita els botons de la barra d'eines en funció del mètode que ha sigut cridat.

4.4.2 L'iniciador de la simulació

Per últim explicarem la manera que té *Repast* d'iniciar la simulació. La classe que implementa l'iniciador de la simulació és la *SimInit* (figura 12):

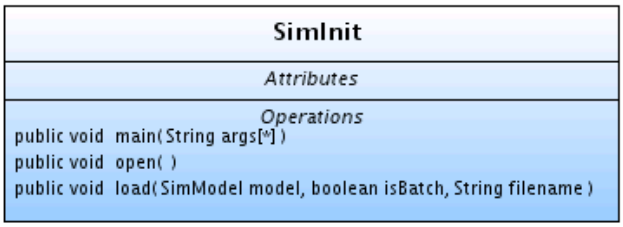

Figura 12: Classe *SimInit*, l'iniciador de la simulació de *Repast*.

El mètode *load()* és el que l'usuari crida des del seu model per a iniciar la simulació, passant-li per paràmetre una referència al mateix. Aquest mètode realitza les següents accions:

1. Crear el controlador gràfic de la simulació.
2. Fixa al model la referència del controlador de la simulació mitjançant el mètode *model.setController()*.
3. Fixa al controlador la referència del model mitjançant *controller.setModel()*. D'aquesta manera tant el model com el controlador tenen una referència a l'altre.
4. Afegeix el controlador com a *listener* del model.
5. Inicia el controlador gràfic fent un *display()* del mateix.

Si un usuari crea una simulació, per a iniciar-la haurà d'executar les següents línies des del seu model:

```
1  public class MyModel extends SimpleModel {
2
3      public static void main(String[] args){
4        SimpleModel model = new SimpleModel();
5        SimInit init  = new SimInit();
6        init .loadModel(model, null, false);
7    }
8  }
```

Listing 3: Iniciació de la simulació

4.5 El sistema de displays

Repast ofereix a la seva *web* un tutorial per a aprendre a representar objectes gràfics a la simulació. Veiem quin és el sistema de *displays* que utilitza i com ho fa per a dibuixar a pantalla els diferentes objectes gràfics de la mateixa.

Veiem mitjançant aquest diagrama en què consisteix el sistema de *displays* de *Repast*.

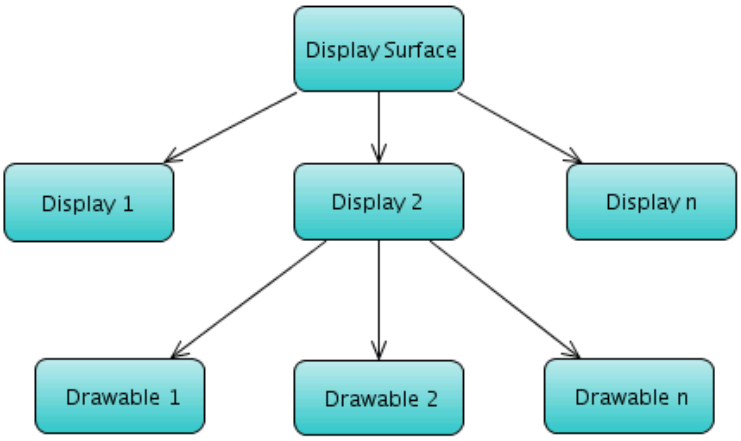

Figura 13: Sistema de displays de Repast.

L'objecte que conté tots els elements gràfics de la simulació és la classe *DisplaySurface*, que és l'equivalent a una finestra. Així doncs per cada finestra que vulguem obrir per a representar objectes haurem de crear un objecte d'aquest tipus. Aquests són els mètodes més importants de la classe:

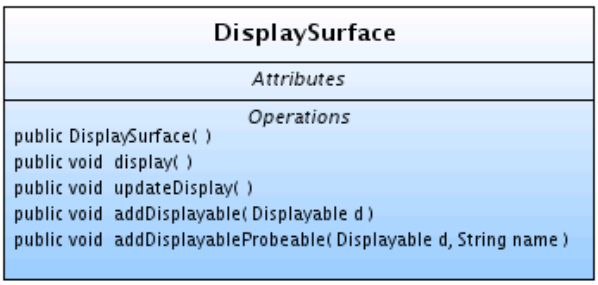

Figura 14: Classe *DisplaySurface* de Repast.

Per a dibuixar la *surface* hem de cridar al seu mètode *display()*, que la farà aparèixer a pantalla. Després, cada vegada que vulguem actualitzar-la haurem de cridar al seu mètode *updateDisplay()*, que repinta tots els *displays* que conté. Mitjançant el mètode *addDisplayable()* podem afegir un *display* a la *surface*.

Tenim que una *DisplaySurface* pot tenir diferents *displays*. Aquests objectes són l'equivalent a les "capes", és a dir en una finestra podem tenir n capes, cada una contenint els seus propis objectes. Els *displays* són proporcionats en diverses classes, un exemple d'elles és la classe *Object2DDisplay*.

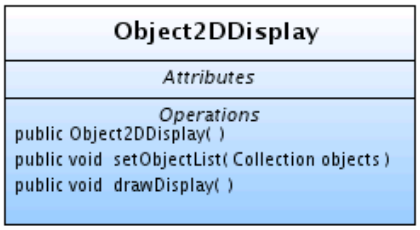

Figura 15: Classe *Object2DDisplay* de Repast.

Mitjançant el seu mètode *setObjectList()* podem fixar la llista d'objectes de tipus *Drawable* que conté el *display*, els quals seran tots dibuixats quan es cridi al mètode *drawDisplay()*. Hem de fixar-nos que la classe **no té** un mètode *getObjectList()* per a obtenir la llista d'objectes del *display*.

Per últim podem veure que cada *display* pot tenir n objectes de tipus *Drawable*. Els objectes *Drawable* són els que es poden dibuixar a la pantalla a dins d'un *display*. Qualsevol objecte que vulgui ésser representat gràficament als *displays* ha d'implementar una interfície que asseguri que implementa uns mètodes bàsics.

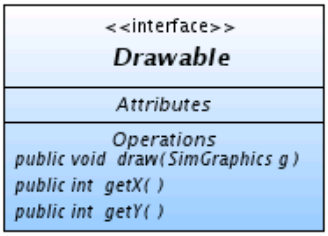

Figura 16: Interfície *Drawable* de Repast.

Per a dibuixar un objecte fan falta saber dues coses essencials:

1. Quina és la seva posició.
2. Què és el que s'haurà de dibuixar a pantalla (un cercle, un requadre, una imatge, etc...).

Podem representar als *displays* qualsevol objecte de la simulació sempre que implementi la interfície *Drawable*, que obliga a implementar:

1. Els mètodes *getX()* i *getY()*, que donen la posició (x,y) del mateix.
2. El mètode *draw()*, que pinta l'objecte. A aquest mètode se li passa una referència a l'objecte que representa els gràfics a pantalla (*SimGraphics*) i els utilitza per a dibuixar l'objecte gràfic.

Veiem un petit exemple de com ho hauriem de fer per a muntar un sistema de *displays* a *Repast* i dibuixar elements gràfics a pantalla:

```
1  public class MyModel extends SimpleModel {
2
3    public buildDisplay() {
4      DisplaySurface dSurf = new DisplaySurface(this, "MySim");
5      dSurf.setBackground(Color.WHITE);
6      registerDisplaySurface("MySim", dSurf);
7
8      ArrayList<Drawable> agentsList = new ArrayList<Drawable>();
9      for(int i=0;i<3;i++) { agentsList.add(new MyAgent()); }
10
11     Object2DDisplay agentsDisplay = new Object2DDisplay(world);
12     agentDisplay.setObjectList(agentList);
13
14     dSurf.addDisplayableProbeable(agentDisplay, "Agents");
15   }
16 }
```

Listing 4: Iniciació de la simulació

En aquest exemple estem creant una *Display Surface* anomenada "MySim" a la qual li posem de fons un color blanc (línies 4,5,6). Seguidament creem una llista d'objectes de tipus *Drawable* i li afegim tres agents que implementen la interfície *Drawable* (línies 8,9). Per acabar creem un *display* d'agents, li fixem la seva llista d'objectes dibuixables (línies 11,12) i la afegim a la *DisplaySurface* amb el nom "Agents" (línia 14).

Després d'això hauríem de dibuixar la *surface* a pantalla cridant al seu mètode *display()*. A continuació, un cop la *surface* hagi aparegut a pantalla, cada vegada que vulguem actualitzar-la hauríem de cridar al seu mètode *updateDisplay()*.

4.6 El sistema de gràfiques

A la *web* del projecte *Repast* existeix un tutorial on s'expliquen els diferents gràfics que es proporcionen i la manera en que es poden crear. L'únic tipus de gràfic que ens permet representar els valors de n propietats en un mateix gràfic és el *OpenSequenceGraph*. Cada propietat que representem al gràfic ho hem de fer mitjançant una seqüència:

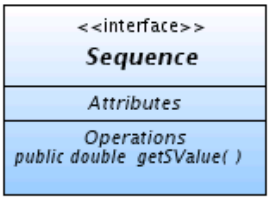

Figura 17: Interfície *Sequence*, definint una seqüència del sistema de gràfiques.

Qualsevol objecte que vulgui representar una propietat seva a un gràfic ha d'implementar la interfície *Sequence*, per tant haurà d'implementar el mètode *getSValue()*. En aquest mètode es retorna un valor de tipus *double*, que serà representat al gràfic on és contingut la seqüència.

Un exemple de com crear un gràfic i afegir-li sequències que representin les propietats seria el següent:

```
1  class DummyClass implements Sequence {
2      int value;
3
4      public DummyClass(int value) {
5          this.value = value;
6      }
7        public double getSValue() {
8          return value;
9      }
10   }
11
12 OpenSequenceGraph graph = new
13 OpenSequenceGraph("AgentStats", this);
14
15 graph.setXRange(0, 200);
16 graph.setYRange(0, 200);
17 graph.setAxisTitles("Time", "AgentAttributes");
18
19 graph.addSequence("Sequence1", new DummyClass(10));
20 graph.addSequence("Sequence2", new DummyClass(20));
```

Listing 5: Iniciació de la simulació

A l'exemple podem veure com creem la classe *DummyClass* que implementa la interfície *Sequence*, per tant proporciona el seu mètode *getSValue()*. Al constructor de la classe li passem el valor que ha de representar la seqüència.

El que fem és crear un gràfic de tipus *OpenSequenceGraph* i hi afegim dues seqüències, és a dir dos objectes de la classe *DummyClass* que implementen *Sequence*. Al mètode *getSValue()* de la classe es retornarà el valor *value* que conté la classe i que s'ha fixat mitjançant el constructor al crear-la. El resultat serà un gràfic que mostrarà dues línies, una que infinitament valdrà 10 i una altra que valdrà 20.

La manera que té aquest gràfic per a mostrar-se per pantalla és cridar al seu mètode *display()*. Després, per actualitzar contínuament el valor de les seves seqüències és executar el seu mètode *step()*. Aquest mètode crida al mètode *getSValue()* de cadascuna d'elles i dibuixa els valors que retornen. L'inconvenient que tenen aquests gràfics és que no s'actualitzen sols, sinó que es deixa a l'usuari escollir el moment en que crida al seu mètode *step()* per actualitzar-lo.

5 Disseny i implementació

Un cop fets els estudis necessaris per a poder realitzar la migració hem de dissenyar una sol·lució eficaç que acompleixi els requeriments plantejats. Per això haurem de realitzar modificacions a totes les parts estudiades, començant pel planificador.

5.1 Migració del planificador

Per a realitzar la migració del planificador haurem de realitzar canvis a les diferents parts del mateix que hem estudiat. Aquestes són les accions, els grups d'execució, les eines per a crear accions i la classe del planificador.

5.1.1 L'acció i el grup d'execució

Una de les característiques que té el simulador *SNS* és que té la capacitat de retrassar una acció en funció de la propietat *LastWorkingTime* de l'objecte destí, que és el temps que es calcula que aquest estarà ocupat.

Imaginem que estem a temps de simulació *ticks=3* i volem executar el mètode *dummy()* d'un objecte als *ticks* actuals. El planificador del *SNS* li preguntaria a l'objecte destí fins quan calcula que estarà ocupat, obtenint el seu *LastWorkingTime*. Si aquest objecte no pot atendre peticions fins a temps de simulació *ticks=6*, el planificador retrasarà la execució de l'acció fins a aquell moment. Sabem que el *Repast* no té aquesta funcionalitat, per tant la hem d'afegir. Per a implementar-la necessitem que l'acció guardi en variables internes l'objecte destí i el nom del mètode que ha d'executar, ja que el planificador ha d'obtenir de la mateixa l'objecte destí per a obtenir el seu *LastWorkingTime*.

Els elements de la simulació han de poder comunicar-se entre sí. Al simulador *SNS*, tots aquells elements que vulguin poder rebre informació sobre els events que es produeixen a la simulació han d'implementar la interfície *SimulationListener*. Aquesta interfície obliga al mateix a implementar el mètode *simulationChanged(SimulationEvent event)*, a on aquest pot rebre un event provinent d'un altre element. En funció del tipus d'event que rebi, aquest executa (si el té) el mètode que processa aquest event per a realitzar les operacions oportunes. Veiem un exemple:

1. El sensor de l'agent *A* detecta un foc i genera un event del tipus *FireDetectedEvent* per a l'agent *A*.

2. Quan el planificador entrega l'event a l'agent *A* ho fa executant el mètode *simulationChanged (SimulationEvent ev)* del agent (passant-li per paràmetre l'event *FireDetectedEvent*). Tots els events possibles extenen de la classe *SimulationEvent*.

3. El mètode *simulationChanged (SimulationEvent ev)* processa aquest event buscant un mètode de la classe tipus *process(FireDetectedEvent ev)*. Si aquest mètode existeix l' executa.

Per incorporar això al *Repast* ens farà falta poder executar mètodes d'objectes destí passant l'event per paràmetre. De l'estudi de *Repast* sabem que no té aquesta funcionalitat, per tant haurem d'incorporar una variable interna a dins de l'acció on guardi l'event i haurem d'aconseguir que al crear la mateixa s'executi el mètode de l'objecte destí passant per paràmetre aquest event. Finalment per acomplir aquests requeriments dissenyem la sol·lució de la figura 18.

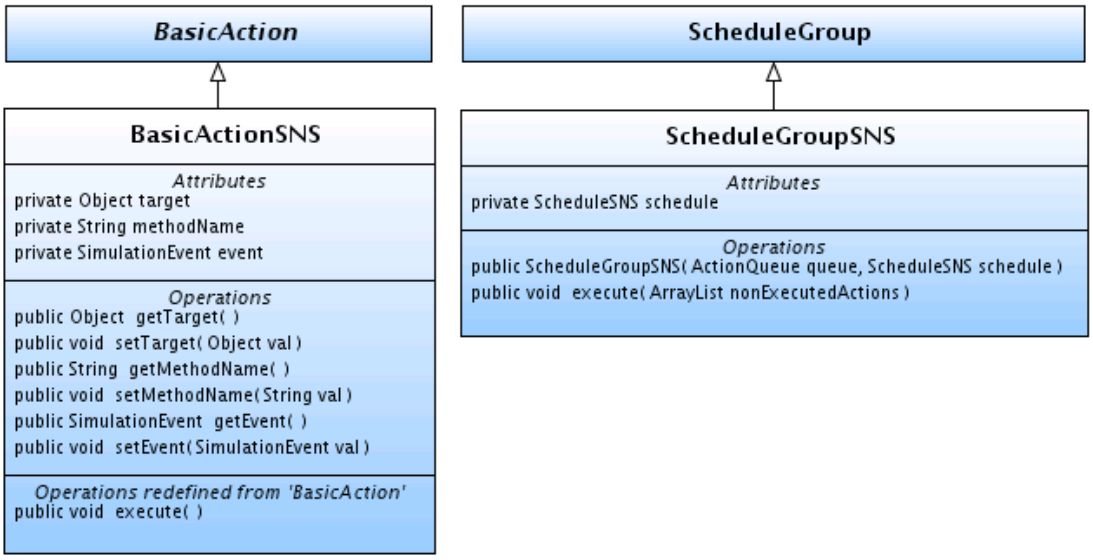

Figura 18: Classes BasicActionSNS i ScheduleGroupSNS del nou simulador.

Veiem que a la classe *BasicActionSNS*, que extén *BasicAction*, incorporem les propietats *target, methodName* i *event*, que són l'objecte destí, el mètode que volem executar d'aquest objecte i l'event que volem passar per paràmetre, respectivament. Les tres propietats tenen els seus *getters* i *setters* corresponents. El mètode *execute()*, que prové de la superclasse *BasicAction*, enlloc d'implementar-lo el seguim deixant com a abstracte per a que l'implementi l'usuari.

Creem també la classe *ScheduleGroupSNS*, que extén *ScheduleGroup*. A aquesta li afegim la propietat *scheduler*, que guarda una referència al que serà el nou planificador.

El mètode *execute()*, que prové de la superclasse *ScheduleGroup*, l'hem redefinit i hem incorporat el nostre propi algorisme d'execució. Podem veure com li passem per paràmetre una llista d'accions no executades, on es guardaran les accions que no han sigut portades a executar per algun motiu. Veiem què és el que fa:

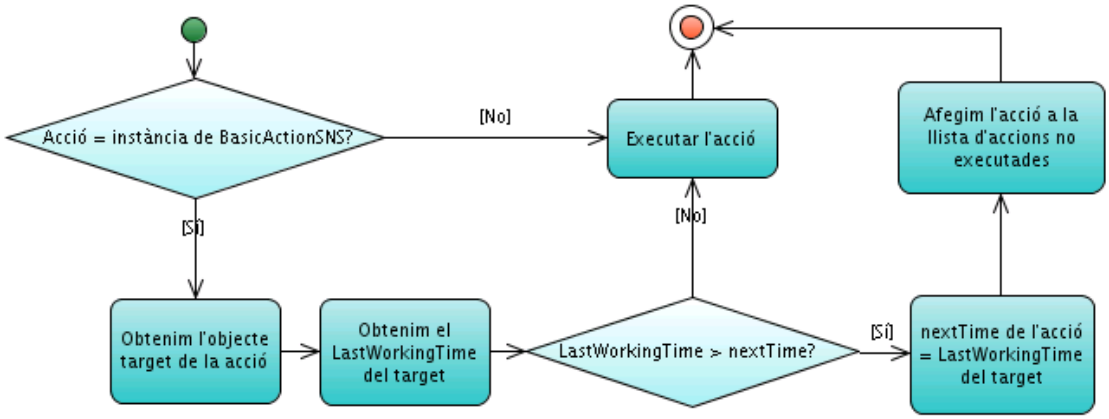

Figura 19: Algorisme d'execució d'accions del nou planificador.

Podem veure com primer de tot comprovem si l'acció és una instància del nou tipus *BasicActionSNS*. Si no ho és, significa que és una acció original de *Repast*, per tant l'executem igual que es feia abans de la migració. Això comporta que al nou planificador també podem executar accions originals de *Repast* sense cap problema, per tant no hem perdut funcionalitats.

Si l'acció, en canvi, és del tipus *BasicActionSNS*, primer de tot fem un *cast* de l'acció i obtenim l'objecte destí per a obtenir d'aquest la seva propietat *LastWorkingTime*. Mirant aquesta propietat podem saber si l'objecte destí està ocupat. Si no ho està, executem l'acció. Si en canvi està ocupat, retrassem l'execució de l'acció fins al moment en que aquest estigui lliure, posant al *nextTime* de l'acció el valor del *LastWorkingTime* de l'objecte *target*. Llavors afegim l'acció a la llista d'accions no executades. Aquesta llista d'accions és una referència de la que ens passa el planificador, per tant el mateix podrà accedir a les accions que no hagin sigut executades i fer amb elles el que sigui oportú.

5.1.2 Les eines de creació d'accions

Haviem comentat que el *Repast* utilitza les classes *ActionUtilities* i *ByteCodeBuilder* per a generar dinàmicament i en temps d'execució classes exteses de *BasicAction* que implementen el seu mètode abstracte *execute()* per a cridar al mètode de l'objecte destí. Per a que a partir d'ara les accions puguin cridar al mètode d'un objecte passant-li un event per paràmetre haurem de modificar-les. Les classes resultants són les següents:

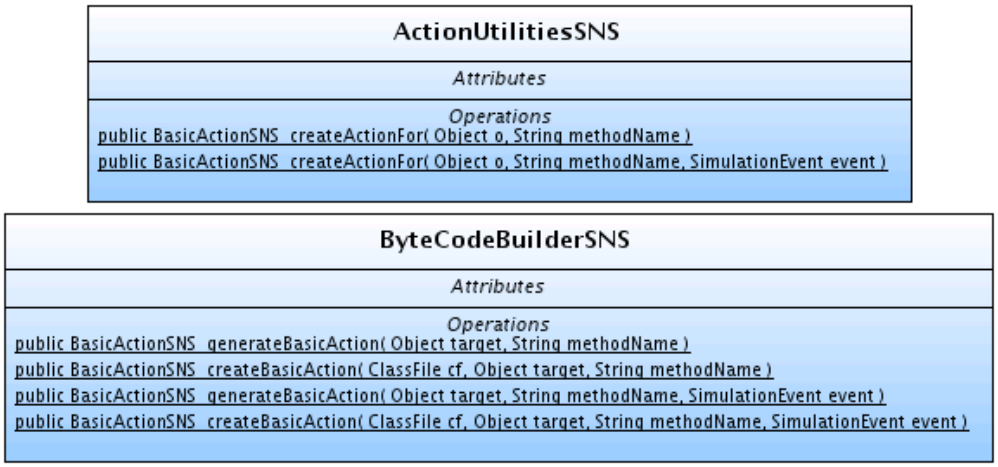

Figura 20: Classes per a crear accions al nou simulador.

Creem la classe *ActionUtilitiesSNS* amb dos mètodes estàtics que ens permeten crear accions per a:

1. Executar el mètode d'un objecte sense paràmetres.
2. Executar el mètode d'un objecte passant un event per paràmetre.

D'aquesta manera podrem executar el mètode *simulationchanged()* d'un objecte passant-li l'event per paràmetre i també seguim proporcionant la possibilitat d'executar mètodes en objectes destí com es feia abans, és a dir sense paràmetres.

Podem veure que els dos mètodes retornen una acció de tipus *BasicActionSNS*. Gràcies a que la nova acció extén de la classe *BasicAction* de *Repast* podem retornar aquest tipus i guardar la acció a una variable de tipus *BasicAction*, fent després un *cast* al nostre tipus *BasicActionSNS* quan faci falta.

La nova classe *ByteCodeBuilderSNS* proveeix els mètodes que necessita la classe *ActionUtilitiesSNS* per a crear les accions. Veiem que, per tant, conté els mètodes que necessita aquesta última per a crear accions que executin mètodes amb i sense event per paràmetre. Si cridem al mètode que crea una acció passant un event per paràmetre, la classe creada dinàmicament tindrà el següent aspecte:

```
1  public class BAB_SYNTH_1 extends BasicActionSNS {
2
3    private Object target;
4    private SimulationEvent event;
5
6    public BAB_SYNTH_1(Object target, SimulationEvent event) {
7      this.event = event;
8    }
9
10   public void execute() {
11     target.execMethod(event);
12   }
13
14   public Object getTarget() {
15     return target;
16   }
17
18   public SimulationEvent getEvent() {
19     return event;
20   }
21 }
```

Listing 6: Acció creada dinàmicament

Podem veure que la classe resultant guarda referències a l'objecte destí i l'event que ha de passar per paràmetre. Veiem també com al seu mètode *execute()* crida al mètode de l'objecte destí passant-li l'event per paràmetre.

5.1.3 El planificador

Per acabar ens queda realitzar les modificacions necessàries a la classe que proveeix el planificador, la classe *Schedule*, per a que al crear les accions utilitzi les noves classes. També haurem d'afegir alguns mètodes per a que els objectes de la simulació del simulador *SNS* puguin planificar accions al planificador de *Repast*. El disseny resultat el podem veure a la figura 21.

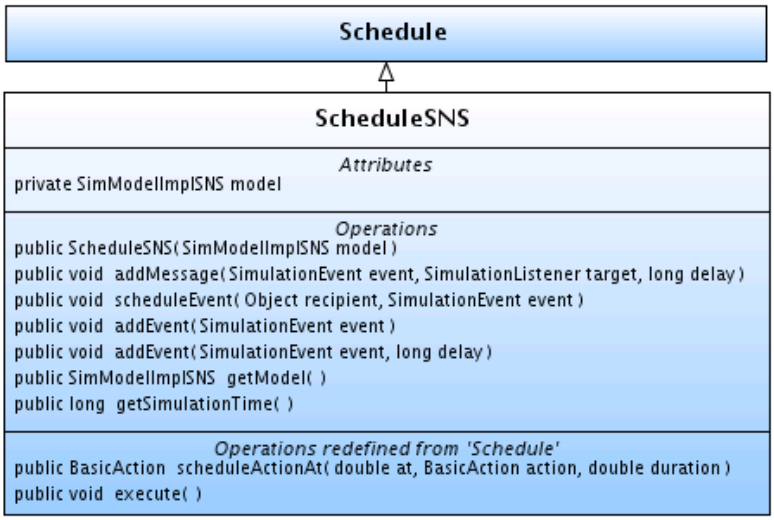

Figura 21: Classe ScheduleSNS, el planificador del nou simulador.

Podem veure que hem afegit els següents mètodes a la classe:

- *ScheduleSNS()*: Al constructor de la nostra classe creem un grup d'execució de tipus *ScheduleGroupSNS*, que és el que té les modificacions que hem fet. Llavors guardem a la variable *groupToExecute* de la superclasse *Schedule* la referència al nostre tipus de grup. Així el *Repast* treballa amb el grup d'execució mitjançant la seva classe original, *ScheduleGroup* , i quan ens interessi accedir-hi a un mètode de la nostre classe, farem un *cast* a *ScheduleGroupSNS*.

- *addMessage(SimulationEvent event, SimulationListener target, long delay)*: La manera que tenen els objectes del simulador *SNS* de planificar accions per a executar mètodes d'un altre objecte de la simulació és mitjançant aquest mètode, per tant l'hem hagut d'afegir al planificador de *Repast*. El que fem en aquest mètode és:

  - Obtenir el temps actual de la simulació mitjançant el mètode *getSimulationTime()* i sumar-li el *delay*. Si estem a *ticks=7* li sumem el *delay* de 3 i l'acció s'executarà a temps *ticks=10*.
  - Fixar el temps d'execució a dins de l'event i cridar al mètode *scheduleEvent()* passant-li l'objecte destí i l'event per paràmetre. Aquest mètode s'encarrega de crear, cridant al mètode indicat de la classe *ActionUtilitiesSNS*, l'acció de tipus *BasicActionSNS* que executarà el mètode *simulationChanged()* de l'objecte destí passant-li l'event per paràmetre.
  - Planificar l'acció cridant al mètode *scheduleActionAt(...)*

  D'aquesta manera hem aconseguit poder planificar a *Repast* accions que permetin canviar l'estat d'una simulació creada amb objectes del simulador *SNS*.

- *addEvent()*: S'encarrega de fer el mateix que el mètode *addMessage()* però, enlloc d'executar el mètode *simulationchanged()* d'un objecte passat per paràmetre, executa aquest mateix mètode al model. Per això necessitem tenir al *scheduler* una referència a aquest, la qual es fixa mitjançant el constructor (*ScheduleSNS (SimModelImplSNS)*) i es retorna mitjançant el seu *getter* corresponent (*getModel()*).

- *getSimulationTime()*: El *Repast* utilitza el mètode *getCurrentTime()*, que retorna un valor de tipus *double*, per a obtenir el temps actual de la simulació. En canvi el simulador *SNS* utilitza el mètode *getSimulationTime()* per a retornar un valor de tipus *long*, ja que aquest guarda el temps de simulació en nanosegons.

  Per a que funcionin les crides que realitzen els objectes del simulador *SNS* per a obtenir el temps de simulació, hem afegit aquest mètode. El que fa és obtenir el temps de simulació del planificador original de *Repast* cridant al mètode *getCurrentTime()* i fent un *cast* a *long*.

- *scheduleActionAt()*: Aquest mètode és l'encarregat d'inserir la acció a la cúa d'accions del planificador. Quan vam estudiar l'algorisme de planificació del *Repast* original vam veure que no és possible planificar accions al temps de simulació actual, és a dir si tenim *ticks=5* no podem planificar una acció que s'hagi d'executar a *ticks=5*.

  Al simulador *SNS* sí que es poden planificar accions al temps de simulació actual, per tant hem hagut de modificar l'algorisme de planificació per a donar aquesta possibilitat. L'algorisme resultant és el següent:

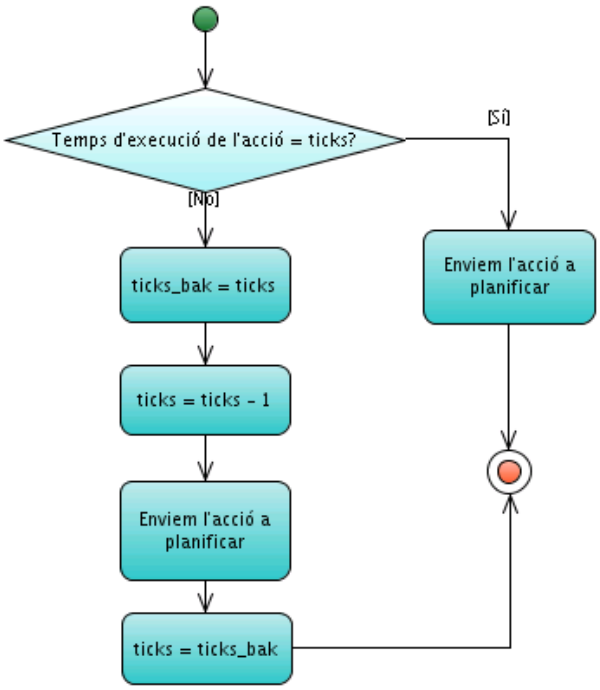

Figura 22: Representació del nou algorisme de planificació del planificador.

Podem veure que el que fem és:

  - Si l'acció es vol executar en un temps major al de l'actual de la simulació s'envia a planificar, tal i com es feia fins ara.
  - Si es vol executar al temps actual de la simulació, guardem aquest en una variable de recolzament *ticks_bak*. Llavors decrementem els *ticks* actuals restant-li 1.

  - Enviem l'acció a planificar. Com que en aquest mateix moment els *ticks* són 1 unitat inferiors al temps d'execució de l'acció, el planificador deixarà inserir-la a la cua.
  - Per acabar restaurem els *ticks* per a restaurar el temps de la simulació.

- *execute()*: Ja hem estudiat que aquest mètode serveix per a portar a execució les següents accions que han d'ésser executades. També hem hagut de modificar l'algorisme que executa per adaptar-lo a les nostres necessitats. L'algorisme resultant és el següent:

  - Omplir el grup d'execució amb les accions que s'han d'executar al següent *tick*.
  - Executar el grup d'execució cridant al seu mètode *execute()*, passant-li una llista d'accions no executades per paràmetre.
  - El grup d'execució executa les accions i inserir a la llista que li hem passat per paràmetre les accions que no han sigut executades, per exemple perquè l'objecte destí estava ocupat i s'ha retrassat l'execució de la mateixa.
  - Re-inserir a la cua d'accions del planificador les accions que el grup d'execució ha retornat a la llista d'accions no executades.

  D'aquesta manera tanquem l'algorisme que ens permet retrassar l'execució d'una acció en funció de la propietat *LastWorkingTime* de l'agent destí.

Mitjançant tots aquests canvis hem aconseguit migrar el planificador original de *Repast* per a que crei, planifiqui i executi accions que permetin córrer una simulació basada en els objectes propis del simulador *SNS*.

5.2 Migració del model

Com que el model de *Repast* es proporciona en una definició i una implementació, haurem de modificar les dues classes per a afegir les funcionalitats requerides.

5.2.1 Definició del nou model

Per a la migració de la definició del model hem hagut de modificar la interfície *SimModel*. Veiem quines modificacions hem fet:

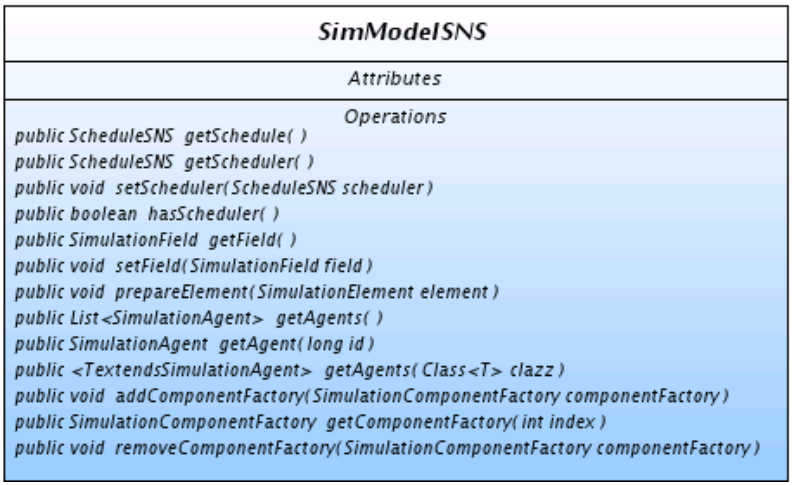

Figura 23: Classe *SimModelSNS*, definició del nou model.

El nou model crearà un planificador de tipus *ScheduleSNS*, per tant hem d'afegir els *getters* i *setters* corresponents. Per a obtenir la referència al planificador d'events, els objectes del simulador *SNS* utilitzen el mètode *getScheduler()*, a diferència dels objectes del *Repast*, que utilitzen el mètode *getSchedule()*, per tant l'hem hagut d'afegir. També hem afegit el mètode necessari per a fixar aquesta referència, el *setScheduler(SchedulerSNS scheduler)*, i un altre mètode *hasScheduler()* que retorna *true* o *false* en funció de si la referència al mateix existeix o no.

Al simulador *SNS* els *phenomenas* són fenòmens que poden ocórrer al camp de la simulació, podent ser des de la humitat o la temperatura d'una zona de la mateixa fins a un agent, ja que aquests últims també són un element físic i poden ser detectats per altres agents. El camp que conté els fenòmens és el *field*, i ha d'existir-hi una referència al nou model. Mitjançant la incorporació dels mètodes *getField()* i *setField()* permetem als objectes del simulador *SNS* obtenir i fixar la referència a aquest.

Els agents, *phenomenas*, sensors, bateries, i tot allò que existeix dins la simulació són elements de la mateixa. Al *SNS*, per tant, tots els objectes que puguin existir a ella han d'implementar, fins i tot el model, la interfície *SimulationElement*, que defineix l'element. Aquesta interfície obliga als elements de la simulació a implementar els mètodes *setup()* i *init()*, que els preparen i inicialitzen respectivament. Per a afegir nous elements a la simulació s'han de preparar i inicialitzar. Per a fer això el model del nostre simulador ha d'implementar el mètode *prepareElement()*, que rep l'element de la simulació a preparar i crida, en funció de l'estat de la simulació, als mètodes *setup()* o *init()* del mateix.

Els agents del simulador *SNS* es defineixen mitjançant la interfície *SimulationAgent*. El model ha de saber quants i quins agents existeixen a la simulació, per tant hem d'afegir els mètodes necessaris per a manegar la llista d'agents, que són els *addAgent(SimulationAgent agent), getAgents()* i *getAgent(long id)* per a afegir un agent a la simulació, obtenir una llista dels que conté el model i obtenir un agent per *id*, respectivament. Objectes de classes diferents poden ser agents mentre implementin la interfície *SimulationAgent*, així que també hem afegit el mètode *getAgent(Class T clazz)*, que ens permet obtenir una llista de tots els objectes que implementin aquesta interfície i siguin d'una classe determinada.

El simulador *SNS* proporciona també un component que s'encarrega de generar nous elements durant el transcurs d'una simulació, com per exemple agents o fenòmens. Aquest és la *Component Factory*, que es proveeix en la interfície *SimulationComponentFactory*. El nou model ha de poder guardar *n Component Factories*, per tant afegim els mètodes *addComponentFactory(), getComponentFactory(int index)* i *removeComponentFactory()* per a afegir, obtenir i eliminar *factories* al model.

5.2.2 Implementació del nou model

A continuació mostrem la sol·lució proposada com a model del nou simulador mitjançant la figura 24.

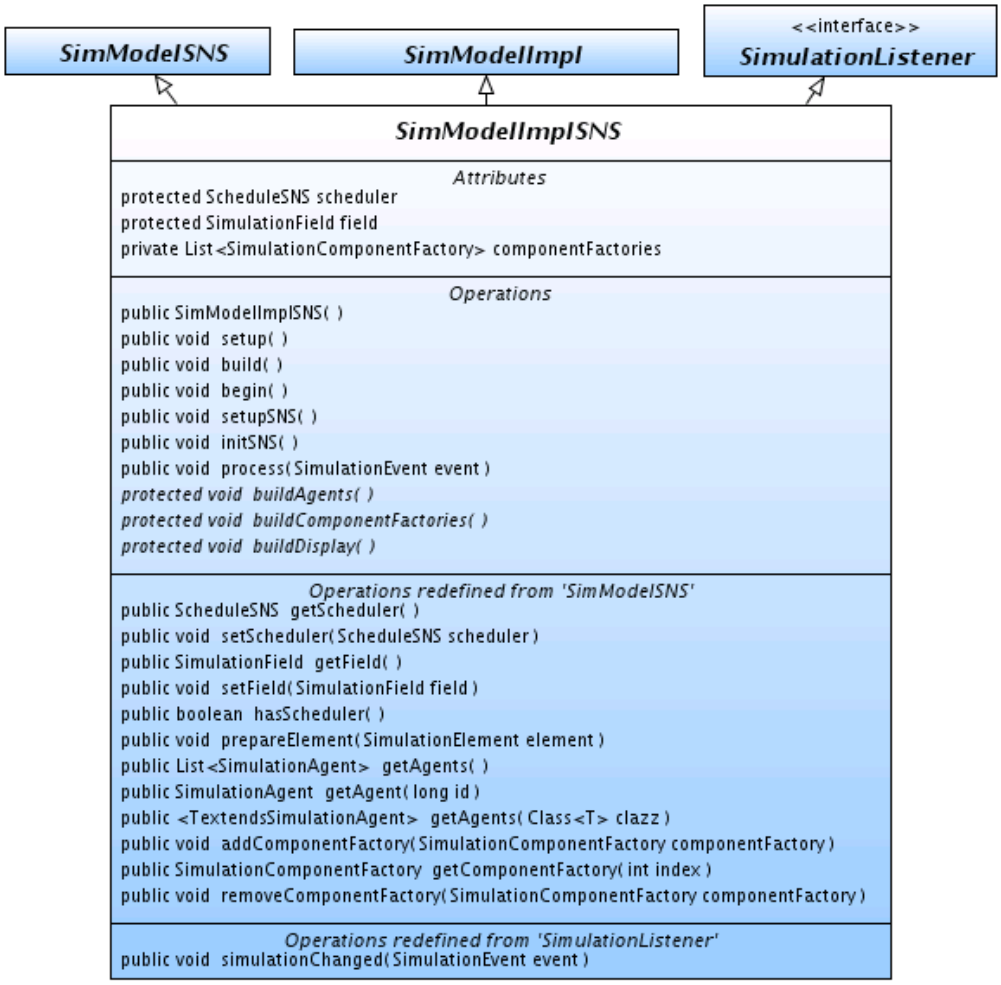

Figura 24: Classe *SimModelImplSNS*, implementació del nou model.

El nostre model implementa la interfície *SimulationListener*, per tant ha d'implementar el seu mètode *simulationChanged(SimulationEvent event)*. Això el converteix en un receptor d'events que serà capaç de rebre informació de la simulació i processar-la si té un mètode *process()* específic per a tractar-la.

Veiem també que aquest extén el model original de *Repast*, el *SimModelImpl*, per tant conserva tota la funcionalitat del mateix. Com que també implementa la interfície *Sim-*

ModelSNS, haurem d'afegir-hi els mètodes que proporciona el model del simulador *SNS*. D'aquesta manera haurem obtingut un model amb les funcionalitats dels simuladors *Repast* i *SNS*.

Les propietats que hem afegit al model són:

- Un planificador *scheduler* de tipus *ScheduleSNS*.
- Un objecte *field* de tipus *SimulationField* que serà el camp que contingui els fenòmens.
- Una llista *componentFactories*, que guarda referències a les factories d'events.

A continuació veiem els mètodes que implementa el model:

- Tres mètodes abstractes *buildAgents(), buildComponentFactories()* i *buildDisplay()* als quals l'usuari ha de crear al seu model els agents, les *component factories* i els *displays* respectivament. La implementació d'aquests mètodes es deixa al usuari ja que en aquests tres casos els elements a construir depenen de cada cas concret de simulació.
- *setup()*: En aquest mètode, implementat de la interfície *SimModel*, es fan les configuracions inicials del model, com inicialització de variables internes i la configuració del *RandomProvider*, que proporciona la possibilitat de crear valors aleatoris de diferents tipus.
- *begin()*: Aquest mètode, com haviem comentat, és cridat pel controlador cada cop que s'inicia una nova simulació. S'encarrega de cridar als mètodes *build(), setupSNS()* i *initSNS()*.
- *build()*: S'encarrega de construir tots els objectes de la simulació. Primer crea el planificador (*scheduler*) i el *field*, i a continuació crida als mètodes abstractes de la classe, *buildAgents(), buildComponentFactories()* i *buildDisplay()* per a construir els agents, les factories de components i el sistema de *displays*, respectivament.
- *setupSNS()*: Aquest mètode realitza la configuració dels elements pertanyents al simulador *SNS*, executant el mateix algorisme que realitzava al mateix. Al simulador origen s'anomenava *setup()*, però per a diferenciar-lo del mètode *setup()* original del model de *Repast* se li ha anomenat així.
- *initSNS()*: Executa el mateix algorisme que executava el mètode *init()* del model del *SNS*, inicialitzant el *field*, les *component factories*, i els agents, deixant-los preparats per a iniciar la simulació.

Tant el model de *Repast* com el model del *SNS* utilitzen eines per a crear valors aleatoris de diferents tipus (*integer, long, boolean,* etc...), encara que cadascun d'ells utilitza metodologies diferents. Per a la total integració hem hagut de canviar les crides al generadors de nombres al·leatoris propis del simulador *SNS* per les de *Repast*.

Per acabar podem veure com el nostre model també implementa els mètodes de les interfícies *SimModelSNS* i *SimulationListener*, els quals ja hem explicat a les definicions de les mateixes.

5.3 La taula d'events executats

El simulador *SNS* proporciona a la seva interfície gràfica una taula on es mostren tots els events executats a la simulació. El següent pas a la migració és la incorporació d'aquesta taula d'events a la interfície gràfica de *Repast*.

La taula d'events es proporciona a les classes *SimulationEventTableView* i *SimulationEventTableRenderer*. La primera és la classe principal que implementa la taula, i la segona és una classe que utilitza la primera com a recolzament.

La taula implementa la interfície *SimulationListener*, per tant proporciona el mètode *simulationChanged(SimulationEvent event)*. Mitjançant aquest mètode podem enviar un event a la taula i aquesta la adjunta i la mostra per pantalla, essent aquesta la manera que utilitzarem per a mostrar en ella tots els events executats. Necessitarem realitzar dues modificacions al *Repast* per a mostrar la taula i enviar-hi events:

- Adjuntar la taula a la interfície gràfica.
- Modificar el planificador per a que hi envii tots els events que s'envien entre elements de la simulació.

5.3.1 Incorporació a la interfície gràfica

Sabem que el que s'ocupa d'iniciar i controlar la interfície gràfica de *Repast* és el controlador gràfic, proporcionat a la classe *Controller*. Per a aconseguir mostrar la taula d'events per pantalla hem de realitzar a la mateixa les modificacions pertinents. També sabem que l'iniciador de la simulació, la classe *SimInit*, és l'encarregat de crear el controlador, per tant també la hem de modificar per a que a partir d'ara crei un controlador del nou tipus. Les dues classes resultants són les que podem veure a la figura 25.

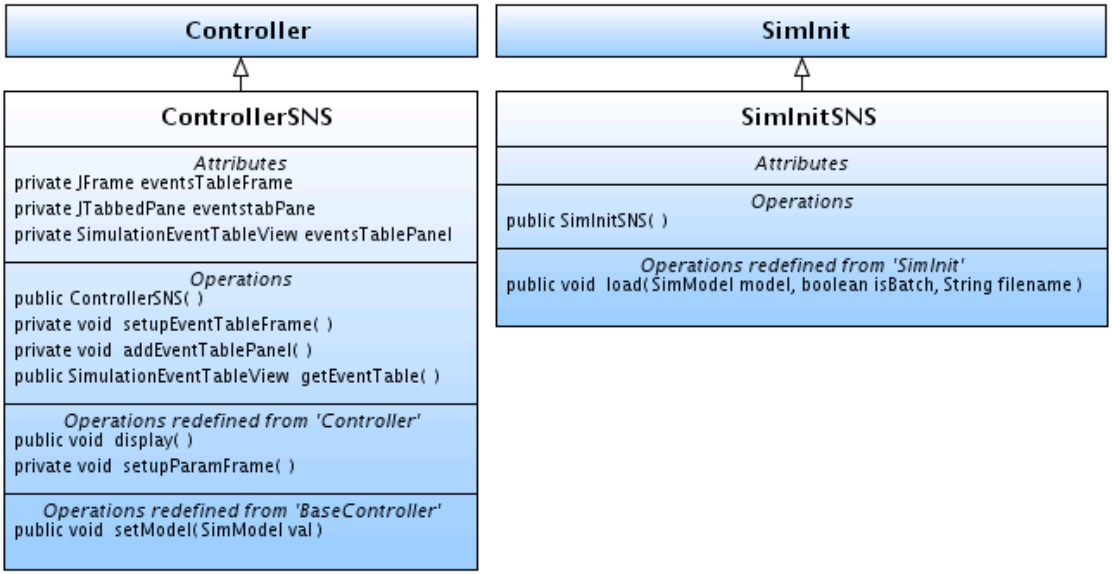

Figura 25: Classes *ControllerSNS* i *SimIniSNS*, el controlador gràfic i l'iniciador.

La nova classe *ControllerSNS* extén del controlador gràfic original de *Repast*, el *Controller*. D'aquesta manera només hem d'afegir els mètodes necessaris per a mostrar la taula

d'events i seguim conservant tota la funcionalitat de l'antic controlador. Les modificacions que hem realitzar són les següents:

- Afegim les següents propietats:
  - *eventsTablePanel*: És un objecte del tipus *SimulationEventTableView*. La taula d'events extén de la classe *JSplitPane*, per tant és un panell que es pot representar a una interfície gràfica.
  - *eventsTableFrame*: Un panell s'ha de representar dins d'un *frame*, per tant necessitem aquest objecte de tipus *JFrame* que contindrà el panell de la nostra taula d'events.
  - *eventsTabPane*: Aquest panell serà el contingut a dins del *frame* de la nostra taula. A dins d'aquest estarà contingut l'objecte *eventsTablePanel*, d'aquesta manera veurem la nostra taula representada a dins d'una pestanya.
- Creem els següents mètodes:
  - *ControllerSNS()*: Després de cridar al constructor de la superclasse s'encarrega de crear el panell de la taula d'events (*eventsTablePanel*).
  - *display()*: Sabem que aquest mètode s'encarrega de construir i mostrar la interfície gràfica, per tant afegim una crida al nostre mètode *setupEventTableFrame()*, que s'encarrega de crear i mostrar el *frame* que conté la taula d'events.
  - *setupEventTableFrame()*: Crea el *frame* de la taula d'events, fixa el títol de la taula i la seva icona, afegeix el panell de tipus "pestanya" i crida al mètode *addEventTablePanel()*, que insereix el panell de la nostra taula d'events a dins de la pestanya creada. Per últim configura la posició (x,y) de la taula i la fa visible.
  - *addEventTablePanel()*: Inserta el panell de la taula d'events a dins de la pestanya *eventsTabPane*.
  - *getEventTable()*: Sabem que el model conté referències al controlador gràfic i el planificador d'events (*scheduler*), per tant afegint aquest mètode el model tindrà accés a la taula d'events. Si des del model li passem una referència de la taula al *scheduler*, aquest podrà enviar els events executats a la mateixa.

A la classe *SimInit* només hem hagut de realitzat modificacions al mètode *load()*. Aquestes són:

- Substituïm la creació del controlador antic per la creació del nou controlador *ControllerSNS*.
- Com que a aquest mètode se li passa una referència al model, fixem a aquest una referència a la taula d'events. Per a fer això s'afegeix al model el *setter* corresponent, el qual veurem en la següent secció.

Gracies a això al iniciar el *Repast* s'iniciarà el nostre controlador gràfic, que mostrarà per pantalla la nostra taula d'events.

5.3.2 L'enviament d'events a la taula

Un cop mostrada la taula a la interfície gràfica ens queda realitzar les modificacions necessàries per a que el planificador li envii tots els events que s'enviin entre elements de la simulació. Les accions, que són les que contenen els events que s'envien d'uns elements de la simulació als altres, s'executen al mètode *execute()* del grup d'execució (*ScheduleGroupSNS*).

Per a poder enviar aquests events a la *EventTable* necessitem poder accedir-hi a una referència de la taula des del grup d'execució, per tant hem afegit els següents mètodes a les següents classes (figura 26):

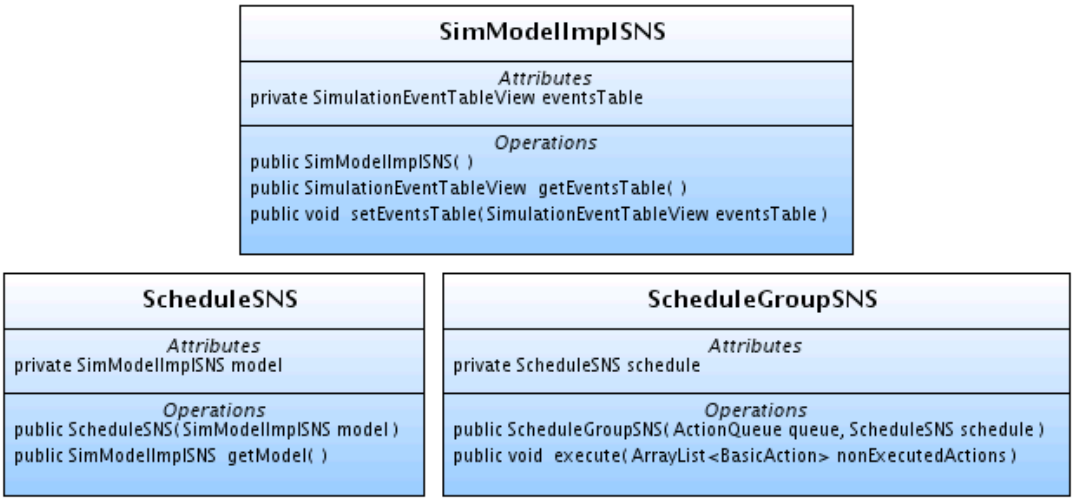

Figura 26: Propietats i mètodes afegits a les classes *SimModelImplSNS*, *ScheduleSNS* i *ScheduleGroupSNS* per a mostrar els events a la taula.

Al model (classe *SimModelImplSNS*) hem afegit el mètode *setEventsTable()* per a guardar-hi a la variable local *eventsTable* la referència a la taula d'events. Aquest mètode és el que hem comentat que crida l'iniciador *SimInitSNS* quan inicia la simulació. També incorporem el mètode *getEventsTable()* per a obtenir la referència a la taula.

Com que el *scheduler* és creat pel model, hem modificat el constructor de la classe *ScheduleSNS* afegint-li per paràmetre una referència a aquest. Així quan es crea el planificador aquest guarda la referència al model a la seva variable local i privada *model*. Per a poder obtenir-la hem afegit també el mètode *getModel()*.

El grup d'execució es crea a la classe *ScheduleSNS*, per tant hem modificat el constructor de la classe *ScheduleGroupSNS* i hem afegit una referència al *scheduler*, que és guardada al mateix a la variable local *schedule*. Podem veure llavors que la referència de la taula d'events passa d'una classe a l'altra de la següent manera:

- *SimInitSNS* fixa la referència de la taula al model amb *model.setEventsTable (SimulationEventsTablesView event)*.
- El model, al crear el planificador, li passa la referència a sí mateix mitjançant el constructor (*scheduler = SchedulerSNS(this)*).

- Quan el planificador és creat, al seu constructor crea el grup d'execució que utilitzarà durant la simulació, passant-li una referència a sí mateix mitjançant *groupToExecute = new ScheduleGroupSNS(this).*

D'aquesta manera des del grup d'execució ja podem accedir a la referència a la taula d'events. Per a enviar els events a la taula o fem mitjançant el següent algorisme:

```
1  if (actions.get(i) instanceof BasicActionSNS)  {
2       ...
3
4       if(schedule.getModel().getEventsTable() != null) {
5            SimulationEventTableView eventsTable;
6            eventsTable = schedule.getModel().getEventsTable();
7            eventsTable.simulationChanged(((BasicActionSNS)actions.get(i)).getEvent());
8       }
9
10      ...
11 }
```

Listing 7: Enviament d'events a la taula des-del grup d'execució

Si l'acció és del tipus *BasicActionSNS* executem tot l'algorisme d'execució de l'acció i al final obtenim la referència a la taula d'events. Si aquesta no és nul·la li enviem l'event fent un cast de l'acció a *BasicActionSNS* i obtenint-lo mitjançant el seu mètode *getEvent()*. Per acabar enviem l'event a la taula d'events mitjançant el seu mètode *simulationChanged()*.

Aquesta és la sol·lució implementada per a mostrar i fer funcionar la taula d'events del simulador *SNS* a la interfície gràfica del *Repast*.

5.4 Millora del sistema de displays

Repast ja incorpora un sistema de *displays*, però volem millorar-lo per a oferir una manera més còmoda de representar objectes gràfics a pantalla. L'objectiu és que qualsevol objecte que tingui una referència al model pugui crear *surfaces* i *displays* i adjuntar-hi objectes de tipus *drawable* sense tenir referències a les mateixes, sinó només pel nom dels objectes. També volem guardar al model referències a tots els elements que formen part del sistema de *displays*.

Un dels principals inconvenients que té el sistema de displays de *Repast* és que, com vam veure al seu estudi, podem fixar una llista d'objectes de tipus *Drawable* a un *display* mitjançant el mètode *setObjectList()* on li passem una llista dels *drawables* que han d'ésser pintats, però no existeix un mètode *getObjectList()* per a obtenir aquesta llista.

El simulador *SNS* ofereix la possibilitat d'afegir i eliminar objectes gràfics en temps d'execució, per tant a *Repast* hem de dissenyar una manera de poder guardar i obtenir la llista d'objectes de tipus *Drawable* de cada *display* per a poder modificar-la.

El principal problema que es planteja és que per a pintar els objectes d'un *display* s'utilitza un fil d'execució (*thread*) diferent del que s'encarrega de l'algorisme de la simulació. Això implica que si guardem referències a les llistes d'objectes per a modificar-les en temps d'execució (afegir i eliminar objectes) poden donar-se errors de modificació concurrent, ja

que mentre modifiquem la llista d'objectes d'un *display* aquest mateix pot estar-hi accedint per a pintar els objectes que conté. Per a la millora del nostre sistema de *displays* utilitzarem el tipus *Object2DDisplay*, per tant per a sol·lucionar aquest problema dissenyem la següent classe:

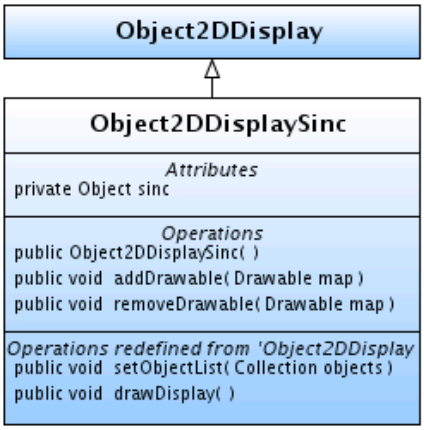

Figura 27: Classe *Object2DDisplaySinc*, el *display* amb sincronisme

La nova classe extén del *display* original de *Repast* i proporciona els mètodes necessaris per a manegar amb sincronisme la seva llista d'objectes de tipus *Drawable*. Podem, en temps d'execució, afegir un objecte *Drawable* a la seva llista mitjançant el mètode *addDrawable()* o eliminar-lo mitjançant *removeDrawable()*. Aquests dos mètodes accedeixen a la llista d'objectes sincronitzant sobre l'objecte *sinc*, cosa que els permet accedir-hi en concurrència. També hem redefinit els mètodes *setObjectList()* i *drawDisplay()* per a que tots els accessos a la llista d'objectes la facin sincronitzant sobre l'objecte *sinc*.

D'aquesta manera sol·lucionem el problema d'accés en concurrència a la llista d'objectes d'un *display*, així doncs l'usuari haurà d'utilitzar a les seves simulacions el *display* de tipus *Object2DDisplaySinc*.

Un cop fetes aquestes modificacions al *display* passem a explicar les propietats i mètodes que hem afegit al model:

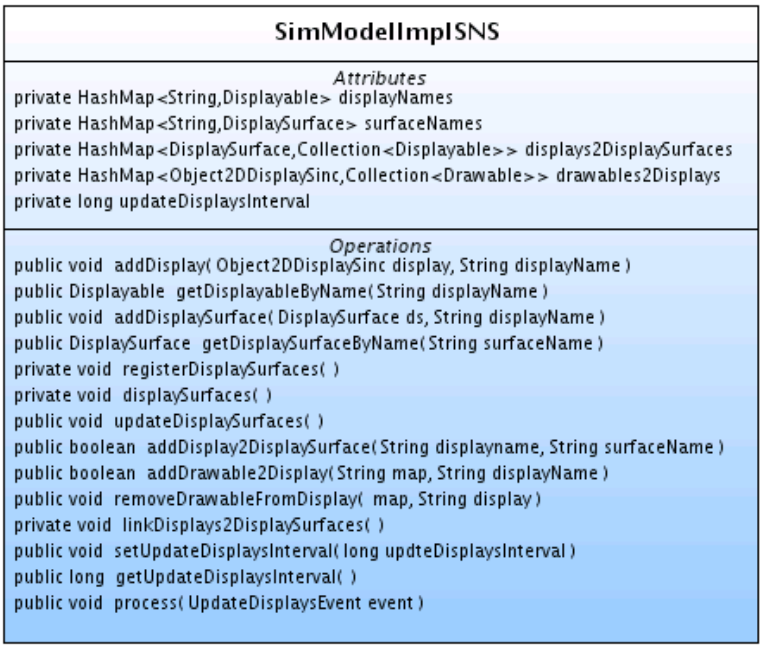

Figura 28: Propietats i mètodes afegits a la classe *SimModelImplSNS* per a millorar el sistema de *displays*.

Lo primer que hem afegit al model és una sèrie de *HashMaps* que ens permeten guardar referències a les *surfaces, displays* i *drawables* que s'adjunten al mateix. A les propietats *surfaceNames* i *displayNames* guardem, respectivament, les *surfaces* i els *displays* que s'adjunten al model i els relacionem amb el seu nom.

Com ja hem estudiat, una *surface* pot tenir *n displays* i un *display* pot tenir *n drawables*. Per a guardar aquesta relació hem creat els *HashMaps displays2DisplaySurfaces* i *drawables2Displays*, que guarden els *displays* que conté una *surface* i els *drawables* que conté un *display* respectivament. Amb aquest últim *HashMap* veiem que podem guardar referències a tots els objectes de tipus *Drawable* que conté un *display* determinat, solucionant així el principal problema que ens plantejava la integració dels objectes gràfics del simulador *SNS* amb el sistema de *displays* de *Repast*.

Per a gestionar aquests *HashMaps* hem creat els següents mètodes:

- *addDisplaySurface(), getDisplaySurfaceByName()*: Amb aquests dos mètodes podem afegir una *surface* al model i associar-la a un nom i obtenir després aquesta *surface* mitjançant el nom que li hem donat al moment d'adjuntar-la.

- *addDisplay(), getDisplayByName()*: Per a afegir *displays* al model i associar-les a un nom i per a obtenir el mateix *display* mitjançant el seu nom.

- *addDisplay2DisplaySurface()*: Un cop hem afegit una *surface* i un *display* al model, podem afegir aquest últim a la llista de *surfaces* mitjançant aquest mètode. Com que els dos tipus d'objectes els hem afegit al model relacionant-los amb un nom (*String*), només farà falta donar el nom del *display* i el de la *surface* a la qual el volem afegir. L'algorisme del mètode s'ocupa d'obtenir els objectes guardats als seus corresponents *HashMaps* i relacionar-los. Si un dels dos objectes no ha sigut encara

afegit al model es retorna *false*, en cas contrari es retorna *true*.

- *addDrawable2Display()*: Permet afegir un objecte de tipus *Drawable* a un *display* donant l'objecte *Drawable* i el nom del *display*. El *display* ha d'haver sigut afegit prèviament al model o sinó el mètode retornarà *false*. En cas contrari retornarà *true*.

  Com que els elements gràfics han de poder aparèixer i desaparèixer en temps d'execució, també hem creat el mètode *removeDrawableFromDisplay()* per a eliminar un objecte *drawable* d'un *display*.

- *registerDisplaySurfaces()*: Un cop tenim la nostra llista de *surfaces* hem d'afegir-la i registrar-la al sistema de displays de *Repast* per a que les pugui mostrar per pantalla, per això hem creat aquest mètode, que realitza aquest registre.

- *displaySurfaces()*: Per a mostrar per pantalla les *surfaces* s'ha de fer un *display()* de cadascuna d'elles, per tant hem creat també aquest mètode que s'encarrega d'aquesta tasca.

- *updateDisplaySurfaces()*: Quan una *surface* es mostra per pantalla hem de poder actualitzar la seva informació per a que repinti tots els elements que conté. Mitjançant aquest mètode podem actualitzar totes les *surfaces* que conté el model, les quals actualitzaran els *displays* que contenen i aquests últims els seus objectes *drawables*.

Un altre inconvenient del sistema de *displays* de *Repast* és que l'actualització dels *displays* la ha de realitzar l'usuari en el moment que vol executant el mètode *update()* de cada *surface*. Per a realitzar aquesta tasca automàticament hem afegit al model lo següent:

- Una propietat *updateDisplaysInterval* que indica cada quant temps s'hauran d'actualitzar les *surfaces* i els seus corresponents mètodes *setUpdateDisplaysInterval()* i *getUpdateDisplaysInterval()*.

- Un mètode *process(UpdateDisplaysEvent event)* que rep l'event que indica que s'han d'actualitzar les *surfaces*. Aquest crida al mètode *updateDisplaySurfaces()*, que s'encarrega d'actualitzar-les, i per últim llença un altre event de tipus *UpdateDisplaysEvent* que ell mateix rebrà un altre cop amb una diferència de temps equivalent a la propietat *updateDisplaysInterval*. Així aconseguim que el sistema de *displays* s'actualitzi automàticament cada n interval de temps.

  El primer event d'aquest tipus és llençat al mètode *begin()*, que s'executa cada cop que s'inicia una simulació.

Gracies a aquest disseny podem afegir i eliminar còmodament objectes gràfics al sistema de *displays* del nou simulador mitjançant mètodes senzills i sense haver de preocupar-nos de la seva gestió. Per últim mostrem un exemple de com hauríem de crear un sistema de *displays* amb els següents elements:

1. Una *surface* de nom *"MySimulation"* que mostrarà una finestra.

2. Un *display* anomenat *"SensorsDisplay"* que contindrà els sensors i un *display* anomenat *"AnimalsDisplay"* que contindrà agents que representaran animals.

3. Un objecte de tipus *Sensor* que implementa *Drawable* i existirà dins del *display* anomenat *"SensorsDisplay"* i un objecte de tipus *AnimalAgent* que estarà contingut dins del *display* anomenat *"AnimalsDisplay"*.

```
1  DisplaySurface mySimSurface = new DisplaySurface(this, "MySimulation");
2  mySimSurface.setBackground(Color.WHITE);
3
4  Object2DTorus sensorsWorld = new Object2DTorus(800,600);
5  Object2DTorus animalsWorld = new Object2DTorus(800,600);
6  Object2DDisplaySinc sensorsDisplay = new Object2DDisplaySinc(sensorsWorld);
7  Object2DDisplaySinc animalsDisplay = new Object2DDisplaySinc(animalsWorld);
8
9  Sensor sensor = new Sensor();
10 AnimalAgent animalAgent = new AnimalAgent();
11
12 addDisplaySurface(mySimSurface, "MySimSurface");
13
14 addDisplay(sensorsDisplay, "SensorsDisplay");
15 addDisplay(animalsDisplay, "AnimalsDisplay");
16
17 addDisplay2DisplaySurface("SensorsDisplay", "MySimSurface");
18 addDisplay2DisplaySurface("AnimalsDisplay", "MySimSurface");
19
20 addDrawable2Display(sensor, "SensorsDisplay");
21 addDrawable2Display(animalAgent, "AnimalsDisplay");
```

Listing 8: Creació d'un sistema de displays amb el nou simulador

Podem comprovar la facilitat amb la que podem crear un sistema de displays amb les noves eines que hem proporcionat al nostre simulador.

5.5 Facilitat d'ús amb el sistema de gràfiques

Per a crear un sistema de fàcil creació de gràfiques que representin el valor de propietats de diferents objectes de la simulació, decidim crear:

1. Un model de dades que guardi els valors actualitzats de les propietats que l'usuari especifiqui.

2. Mètodes per a crear gràfics que representin el valor de propietats que existeixin al model de dades.

5.5.1 El model de dades

Per a crear el model de dades decidim implementar les següents classes (figura 29):

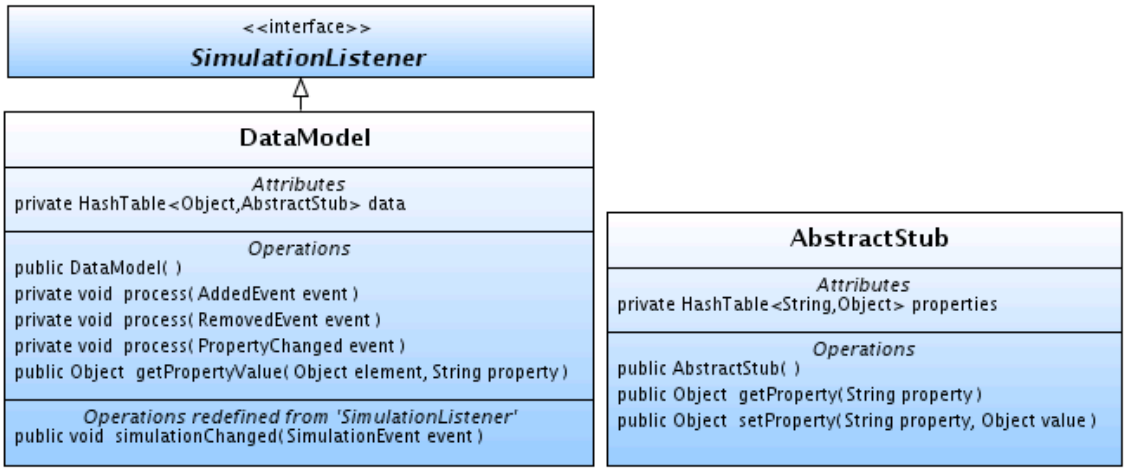

Figura 29: Classes que implementen el model de dades del nostre simulador

El model de dades serà proporcionat per la classe *DataModel*. Veiem que conté la propietat *data*, que és una *HashTable* que guarda la relació d'un objecte i la seva llista de propietats que volem mantenir al model de dades.

Aquesta llista de propietats es proporciona en la classe *AbstractStub*, que conté una *HashTable* que relaciona propietats i el valor de les mateixes. Per a fixar i obtenir el valor d'aquestes propietats la classe conté els mètodes *setProperty()* i *getProperty()*. D'aquesta manera podem guardar al model de dades els n valors de les n propietats de m objectes.

La classe *DataModel* implementa la interfície *SimulationListener*, per tant proporciona el mètode *simulationChanged()*, cosa que ens permet enviar events al mateix. Com que una propietat pot ser afegida, esborrada o actualitzada, decidim implementar els següents tipus d'events:

- *AddedEvent*: Indica que un objecte vol ésser afegit al model de dades. Conté l'objecte a afegir, que es pot obtenir mitjançant el mètode *getAddedObject()*.
- *RemovedEvent*: Indica que un objecte vol ésser eliminat del model de dades. Conté l'objecte a eliminar i es pot obtenir mitjançant el mètode *getRemovedObject()*.
- *PropertyChangedEvent*: Indica que una propietat d'un objecte ha canviat el seu valor. Conté l'objecte, la propietat a actualitzar i el nou valor d'aquesta, que es poden obtenir mitjançant els mètodes *getSource()*, *getProperty()* i *getNewValue()*.

Per a que això funcioni el mateix usuari ha de ser el que llenci aquests events. Per exemple, un lloc ideal on llençar els events *PropertyChangedEvent* seria al *setter* de la propietat, on es canvia el seu valor, així estaria sempre actualitzat. Per a processar els tres tipus d'events, incorporem a la classe els mètodes *process()* que els tractaran:

- *process(AddedEvent event)*: Obté de l'event l'objecte a afegir i l'afegeix a la *HashTable data*. A continuació crea una llista buida de propietats i la relaciona amb el mateix.
- *process(RemovedEvent event)*: Obté de l'event l'objecte a eliminar del model de dades i l'elimina de la *HashTable data*.

- *process(PropertyChangedEvent event)*: Primer de tot obté l'objecte *source*, i d'aquest obté la seva llista de propietats. Si la llista no existeix en crea una de nova i hi afegeix la propietat. Per últim obté la propietat i fixa el seu nou valor. Així doncs, aquest event serveix per a afegir valors de propietats d'objectes i també actualitzar-los.

El mètode *simulationChanged()* és el que rep els events al *DataModel*, i en funció del seu tipus l'envia al corresponent mètode *process()*. Per a que això funcioni el model de dades ha de rebre tots els events que rebi el model, per tant al model afegim:

1. Afegim la propietat *private DataModel dataModel*.
2. Afegim el seu corresponent mètode *getDataModel()*.
3. Modifiquem el mètode *simulationChanged()* i afegim una crida al mètode *simulationChanged()* del model de dades per a cada event que el model rebi.

D'aquesta manera el model de dades rep tots els events que rep el model i, en funció de si tenen relació amb propietats que estiguin mapejades al *DataModel*, és a dir si són events tipus *AddedEvent, RemovedEvent* o *PropertyChangedEvent*, els tracta o no.

Mitjançant això podem tenir en temps d'execució un model de dades que guarda el valor actualitzar de les propietats de diferents objectes especificats per l'usuari.

5.5.2 El sistema de muntatge de gràfiques

Un cop creat el model de dades podem aprofitar-lo per a obtenir els valors de les seves propietats i representar-los mitjançant gràfiques. La sol·lució final per a implementar això la podem veure a la figura 30.

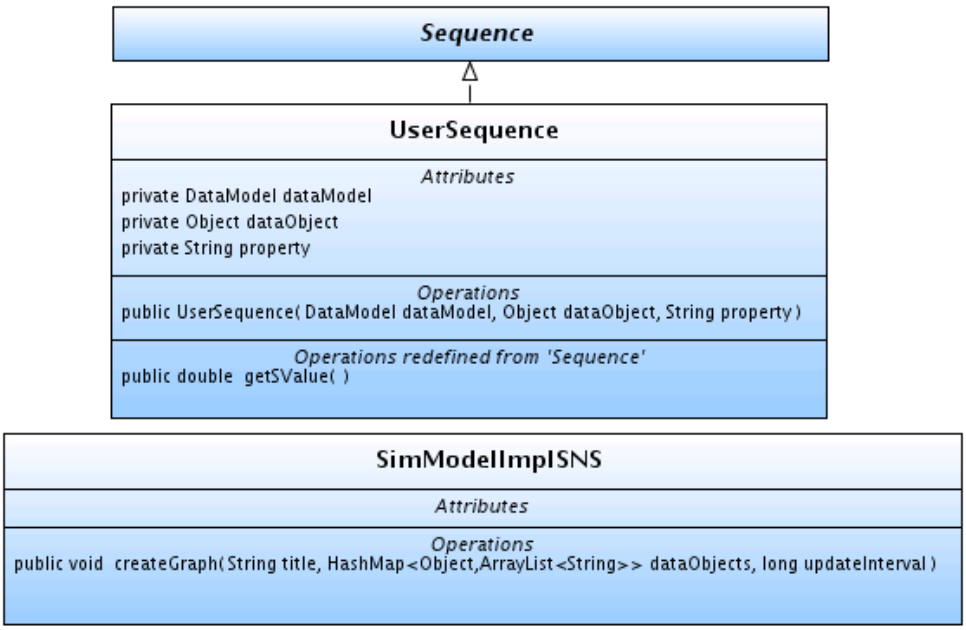

Figura 30: Sol·lució per a les gràfiques

Hem creat la classe *UserSequence*, que s'encarrega de representar a un gràfic el valor d'una propietat del model de dades, per tant al construir una seqüència d'aquest tipus li

hem de passar referències al model de dades, l'objecte que conté la propietat i la propietat que s'ha de representar, ja que una seqüència representa el valor d'una i només una propietat d'un objecte. Aquestes referències es guarden a les variables locals corresponents.

Aquesta classe implementa la interfície *Sequence*, per tant proporciona el seu mètode *getSValue()* que retorna el valor a dibuixar al gràfic. Com que tenim la referència al model de dades, podem obtenir el valor de la propietat del mateix. Per tant, a aquest mètode s'obté el valor actualitzat de la propietat en el moment en que és cridat.

Per a crear fàcilment un gràfic amb una o diverses propietats d'un o diversos objectes hem afegit al model (classe *SimModelImplSNS*) el mètode *createGraph()*. A aquest mètode se li ha de passar per paràmetre un objecte *HashMap* que relaciona n objectes, cadascun amb la seva llista de propietats que volen ésser representades. A més li hem de passar un valor *updateInterval* que fixarà l'interval d'actualització del gràfic.

Un cop creat el gràfic i obtingudes les propietats que volem representar al mateix, el mètode *createGraph()* crea una seqüència (*UserSequence*) per a cadascuna d'elles. A continuació es mostra el gràfic fent un *display()* del mateix i planifica una acció que executi el mètode *step()* del gràfic infinitament en un interval de temps equivalent a *updateInterval* per a actualitzar-lo.

Un cop explicada la manera de representar gràfiques, veiem un exemple de com ho faríem per a representar les següents propietats:

- De l'objecte *ObjectOne*, les propietats x i y.
- De l'objecte *ObjectTwo*, les propietats *isReady*, x i y.

```
1  private HashMap<Object,ArrayList<String>> properties
2  properties = new HashMap<Object,ArrayList<String>>();
3
4  private ArrayList<String> ObjectOneProperties = new ArrayList<String>();
5  private ArrayList<String> ObjectTwoProperties = new ArrayList<String>();
6
7  ObjectOneProperties.add("x");
8  ObjectOneProperties.add("y");
9  properties.put(ObjectOne, ObjectOneProperties);
10
11 ObjectTwoProperties.add("isReady");
12 ObjectTwoProperties.add("x");
13 ObjectTwoProperties.add("y");
14 properties.put(ObjectTwo, ObjectTwoProperties);
15
16 createGraph("MySim", properties, 1000000000);
```

Listing 9: Representació de propietats a gràfiques amb el nou simulador

En aquest exemple el gràfic s'actualitzarà cada segon de la simulació (no cada segon real), és a dir cada 10^9 nanosegons.

El resultat és que hem creat un sistema per a mapejar els valors de diverses propietats a un model de dades i poder representar els seus valors a les gràfiques que especifiqui l'usuari.

6 Proves: L'exemple *CoverageProblem*

Per a realitzar les proves necessàries del simulador resultant decidim migrar un exemple prèviament implementat per al simulador *SNS*. Un cop migrat, podrem provar el correcte funcionament de totes les característiques que hem incorporat a *Repast*. L'exemple que utilitzarem per a la fase de proves serà el *Coverage Problem*, que és el més elaborat que es va implementar per al simulador *SNS*, per tant serà un exemple ideal per a provar el nou simulador.

Aquest exemple s'enfoca a l'estudi de la cobertura d'un àrea determinada considerant el consum d'energia i aplicat a la detecció de focs. El que farem és agafar l'exemple original implementat per al *SNS* i realitzar els canvis necessaris per a que funcioni per al nou simulador *Repast-SNS*. La migració del simulador la hem realitzada pensant en que qualsevol exemple realitzat prèviament per al *SNS* es pugui migrar fàcilment i amb les mínimes modificacions possibles, per tant les úniques parts que necessitarem modificar són:

- El model de la simulació.
- Els elements de la simulació que siguin representats gràficament.

A més també haurem d'afegir proves per a comprovar el correcte funcionament del model de dades i el sistema de gràfiques.

6.1 El model de la simulació

El que fem és agafar el model original de l'exemple, que extén del model original del *SNS*, i canviar la seva superclasse per a que extengui del model *SimModelImplSNS*. Veiem el resultat:

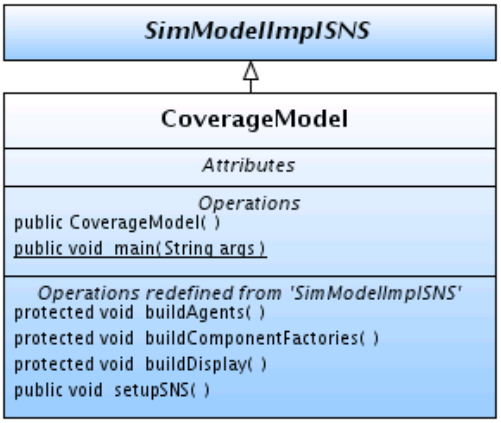

Figura 31: Model de l'exemple *Coverage Problem*.

La superclasse *SimModelImplSNS* ens obliga a implementar els seus mètodes abstractes, que són:

- *buildDisplay()*: S'encarrega de crear el sistema de *displays* que s'utilitzarà per a representar gràficament els elements de la simulació. En aquest exemple creem:

  - Una *surface* anomenada *"displayField"* que contindrà tots els *displays*.
  - Un *display* anomenat *"agentField"* que contindrà tots els agents de la simulació i està contingut dins del *"displayField"*.
  - Un *display* anomenat *"animalField"* que contindrà tots els animals de la simulació i està contingut dins del *"displayField"*.

- *buildAgents()*: Aquest mètode ja existia al model original del *SNS* aquest exemple ja l'estava implementant, per tant no hem de modificar l'algorisme de creació d'agents. En aquest cas l'exemple crea 10 agents de tipus *"Animal"* i 4 agents senzills que representen els sensors de la simulació.

  Per a representar gràficament els agents també afegim cadascún d'ells als *displays* corresponents. Així doncs els 10 agents de tipus *"Animal"* els afegim al *display "animalField"* i els 4 sensors els afegim al *display "agentField"*.

- *buildComponentFactories()*: Aquest mètode també existia al model original del simulador *SNS*, per tant no hem hagut de realitzar cap modificació. En ell es crea una *FirePhenomenonFactory*, que crea contínuament *phenomenas* de tipus foc per a provar la seva detecció per part dels sensors.

Per acabar implementem els mètodes per a configurar i iniciar la simulació, que són:

- *setupSNS()*: Equival al mètode *setup()* del model del simulador *SNS*, per tant copiem el seu algorisme. El que fa és configurar la distribució dels sensors al *field*.

- *main()*: En aquest mètode hem hagut de realitzar les modificacions pertinents per a iniciar la simulació amb el nou *framework*, per tant és el que més diferent ha quedat en relació amb el del exemple implementat per al *SNS*. El que fem és:

  - Crear un iniciador de la simulació (*SimInitSNS*).
  - Crear un nou model (*CoverageModel*).
  - Iniciar la simulació mitjançant el mètode *load(...)* de l'iniciador.

La resta de mètodes que té l'exemple original del *SNS*, com per exemple *getters* i *setters* de diverses propietats que utilitza per a la simulació, es queden tal i com estaven ja que no fa falta realitzar cap modificació.

Podem veure com les modificacions que hem hagut de realitzar al model han sigut mínimes, ja que els mètodes a implementar són pràcticament els mateixos que implementava l'exemple original. Això ha sigut possible gràcies a que el nou model proporciona les funcionalitats dels models de *Repast* i del *SNS*.

6.2 Els elements gràfics

Com que el sistema de *displays* del simulador *SNS* i del *Repast* són diferents hem hagut de modificar la manera en que els elements de l'exemple es representen a pantalla. Com ja sabem, tots els elements que es vulguin dibuixar han d'implementar la interfície *Drawable*, per tant modifiquem els següents elements que han d'ésser representats:

- Agents i sensors (*SimulationAgent, CoverageSensor*).
- Animals (*AnimalAgent*).
- Fenòmens foc (*FirePhenomenon*).

Per a mostrar com s'ha realitzat la migració posarem com exemple els agents de tipus *"Animal"*. En aquest cas el que hem fet és modificar la classe *AnimalAgent*, que és la que representa un animal a la simulació, per a que implementi la interfície *Drawable*, que obliga a la classe a implementar els mètodes *getX(), getY()* i *draw()*. Aquests mètodes permetran a l'objecte retornar la seva posició *(x,y)* i especificar què és el que ha de dibuixar a pantalla, en aquest cas la imatge d'una zebra, una hiena o un elefant, depenent d'un valor aleatori entre 0 i 2.

Amb unes mínimes modificacions hem pogut adaptar els objectes d'un exemple originalment generat per al simulador *SNS* per a poder representar-los gràficament al simulador *RepastSNS*. L'aspecte de la simulació resultant és el següent:

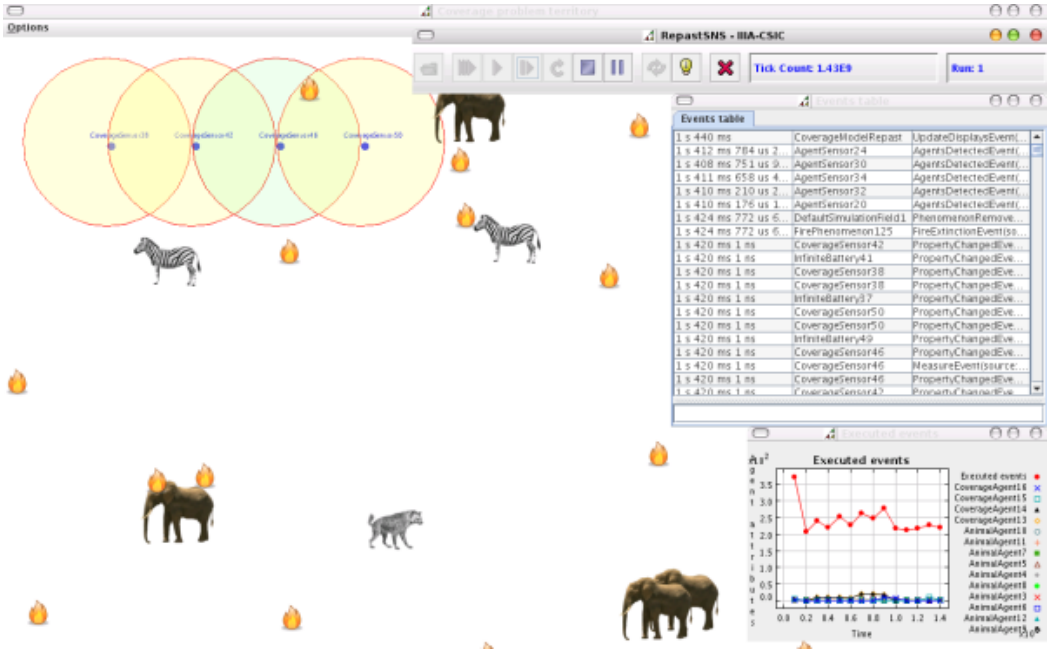

Figura 32: Exemple *Coverage Problem*, migrat per al nou simulador.

Veiem també que a la simulació es mostra per defecte la taula d'events executats, que en tot moment està mostrant la informació actualitzada dels events que s'executen contínuament a la simulació.

6.3 El model de dades i les gràfiques

Per a provar aquesta part de la migració necessitem afegir dues coses al codi del nostre exemple:

1. Mapejar al model de dades una propietat qualsevol d'un element de la simulació.
2. Generar una gràfica amb aquesta propietat.

Com que per a realitzar les proves podem utilitzar qualsevol propietat, decidim mapejar la propietat *"y"* de la classe *AnimalAgent*. L'agent de tipus *"AnimalAgent"* només canvia la seva posició cada vegada que rep un event de tipus *MoveEvent*, que indica que s'ha de moure. Així doncs modifiquem el mètode *process(MoveEvent event)* per a que cada vegada que l'objecte es mogui i la seva propietat *"y"* canvii aquest llenci un event de tipus *PropertyChangedEvent*, que indicarà al model de dades el nou valor de la mateixa. D'aquesta manera el valor de la propietat estarà sempre actualitzat al model de dades i el valor que es representarà a la gràfica serà el real.

Finalment, per a mostrar una gràfica que representi la propietat especificada modifiquem el mètode *setup()* de la classe *AnimalAgent*. El que fem és cridar al mètode *createGraph()* del model passant-li per paràmetre el nom de la classe, el temps d'actualització del gràfic i un *HashMap* que conté:

- Una referència a l'objecte (*this*).
- Una llista que conté el nom de la propietat a representar de l'objecte, en aquest cas la propietat *"y"*.

El resultat és que mostra per pantalla gràfiques com les següents, una per a cada objecte de tipus *"AnimalAgent"* que existeixi a la simulació.

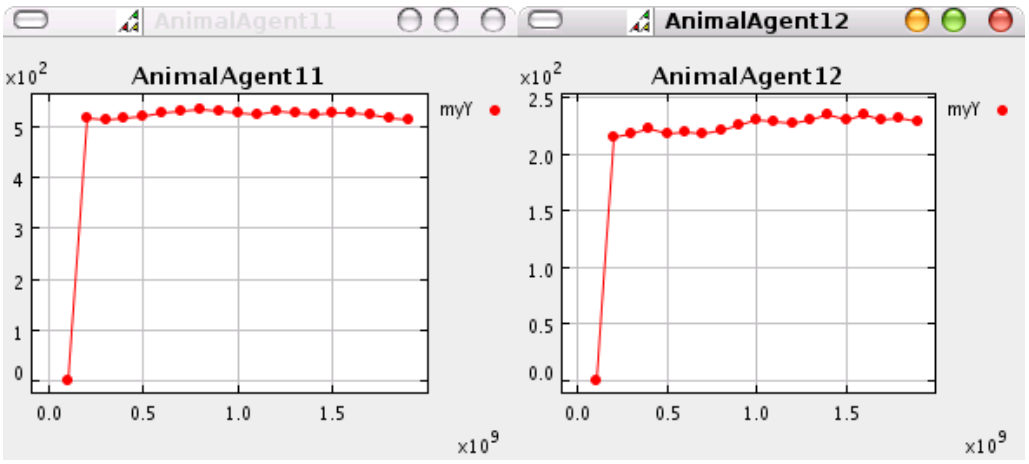

Figura 33: Gràfiques del nou simulador.

Veiem a la figura dues gràfiques representant el valor de la propietat *"y"* de dos objectes de tipus *"AnimalAgent"*.

Aquest exemple conté les funcionalitats essencials per a provar els diferents nuclis que hem modificat:

- Execució d'events i enviament d'informació entre agents (Planificador).
- Configuració de la simulació, a més de la creació dels agents, *phenomenas*, etc... (Model).
- Mostratge dels events executats per pantalla (Taula d'events executats).
- Representació gràfica d'objectes. (Sistema de *displays*).
- Gràfiques de propietats d'objectes (Model de dades i sistema de gràfiques).

Per tant, un cop migrat l'exemple i veient els seu correcte funcionament podem concloure que la migració ha sigut satisfactòria.

7 Resultats i conclusions

Com a resultat d'aquest projecte hem obtingut una plataforma de simulació basada en events per a xarxes de sensors que proporciona les funcionalitats de les dues plataformes originals que s'han utilitzat per a realitzar la migració, el *Repast* i el *SNS*.

La plataforma proporciona, entre d'altres coses:

- Eines de modelatge de xarxes de sensors com a sistemes multi-agent.
- Un sistema millorat de *displays* que permet representar gràficament a pantalla els objectes de la simulació que especifiqui l'usuari.
- Un model de dades que pot guardar fàcilment informació actualitzada de diverses propietats pertanyents a objectes de la simulació.
- Un còmode sistema de creació de gràfiques a partir de propietats del model de dades.

A més de totes les característiques originals que ja proporcionava la plataforma *Repast* (eines de modelatge de xarxes socials, suport per a sistemes d'informació geogràfica, llibreries d'algorismes genètics, etc...).

La nova plataforma serà accessible tant a la comunitat científica que utilitzava *Repast* com la que utilitzava *SNS*, per tant s'allibera sota la llicència *GPL* per a que aquesta se'n pugui beneficiar. Aquesta es pot descarregar des del següent enllaç:

`http://www.maia.ub.es/~cerquide/RepastSNS`

Per a introduir a l'usuari en l'ús de la plataforma aquesta memòria pot servir per a comprendre l'eina resultant i la seva arquitectura, així com la manera de crear exemples senzills.

7.1 Línies futures

A continuació es detallen les possibles línies futures que pot seguir el projecte:

- **Migració a *Repast Simphony***: Donat que a l'estudi de viabilitat es va decidir no migrar a aquesta plataforma perquè encara era molt nova i donaria molts problemes, seria desitjable tornar a intentar la migració a *Repast Simphony* un cop passat un temps prudencial que permeti consolidar la plataforma.
- **Protocols d'encaminament**: La plataforma resultant no és capaç d'optimitzar les comunicacions calculant protocols d'encaminament, per tant la seva incorporació milloraria molt l'acceptació per part de la comunitat.
- **Coordenades món a pantalla i viceversa**: Ara mateix la plataforma no representa els objectes gràfics adaptant les coordenades de món a pantalla i pantalla a món, per tant seria interessant poder afegir aquesta funcionalitat.
- **Manuals d'usuari i exemples**: Aquesta memòria pot servir com a manual d'usuari per a fer servir la plataforma, però seria necessària la implementació de manuals d'usuari, exemples i tutorials en anglès per a millorar l'acollida de la plataforma per part de la comunitat científica, ja que no ha sigut possible realitzar-los per falta de temps al projecte.

Referències

[1] M.Vinyals, J.A.Rodriguez Aguilar, and J. Cerquides. A survey on sensor networks from a multi-agent perspective. *2th International Workshop on Agent Technology for Sensor Networks (ATSN@AAMAS)*, 2008.

[2] D. Cruller, D. Estrin, and M. Srivastava. Overview of sensor networks. *Computer(Long Beach, CA)*, 37(8):41–49, 2004.

[3] Marc Pujol Gonzàlez. Plataforma per a la simulació de xarxes de sensors, February 2008. Projecte de final de carrera d'Enginyeria Informàtica. UAB.

[4] Repast Organization for Architecture and Development (ROAD). Repast project, an agents-based modeling toolkit, 2004.