UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

Deseño dunha linguaxe orientada a partidas de rol en liña

Autor:
Martín Coego Pérez

Director:
Paulo Félix Lamas

Grao en Enxeñaría Informática

Febreiro 2017

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

D. Paulo Félix Lamas, Profesor do Departamento de Electrónica e Computación da Universidade de Santiago de Compostela, e **D. Tomás Teijeiro Campo**, Profesor do Departamento de Electrónica e Computación da Universidade de Santiago de Compostela,

INFORMAN:

Que a presente memoria, titulada *Deseño dunha linguaxe orientada a partidas de rol en liña*, presentada por **D. Martín Coego Pérez** para superar os créditos correspondentes ao Traballo de Fin de Grao da titulación de Grao en Enxeñaría Informática, realizouse baixo a nosa dirección no Departamento de Electrónica e Computación da Universidade de Santiago de Compostela.

E para que así conste aos efectos oportunos, expiden o presente informe en Santiago de Compostela, a (Data):

O director, O codirector, O alumno,

Paulo Félix Lamas Tomás Teijeiro Campo Martín Coego Pérez

Índice xeral

Índice de figuras

Capítulo 1

Introdución

1.1. Os xogos de rol

Un xogo de rol é un xogo de interpretación no que o conxunto de xogadores vai desenvolvendo unha narrativa común, interpretando cada un deles a un personaxe ficticio que inflúe na historia coas súas accións. Un dos participantes asume o papel de mestre ou director de xogo, que en esencia é quen ten a labor de deseñar a historia e fiar os acontecementos, mentres o resto de xogadores toman decisións en base aos seus personaxes. Cando un xogador pretende facer algunha acción con risco de fracasar, o éxito ou fracaso da súa acción determínase mediante unha tirada de dados. Por tanto, podería dicirse que os xogos de rol no seu concepto son un termo medio entre o teatro e os contacontos.

Debido á poca homoxeneidade dos xogos de rol a súa orixe concreta é discutida, mais con frecuencia asúmese que o concepto xurde a partir dos wargames de miniaturas, xogos de mesa nos que se simulan batallas empregando figuriñas[6]. Co tempo, este tipo de xogos foron dándolle un maior peso ás características individuais de cada miniatura e ao seu transfondo, o que levou á aparición dos xogos de rol tal e como os coñecemos hoxe en día. Un fito importante no mundo dos xogos de rol é a aparición de Dungeons and Dragons (publicado orixinalmente en 1974), unha das sagas máis influíntes e de maior éxito comercial, xa que moitos dos xogos que saíron posteriormente estaban moi influídos por esta.

Non pasou moito tempo ata que este fenómeno se extendeu aos ordenadores, especialmente na época de internet, o que propiciou que se comezaran a organizar partidas de rol mediante correo electrónico e posteriormente a través de foros, o que se pasou a denominar como *play-by-post*. Este sistema de rol é asíncrono, é dicir, os xogadores podían participar en partidas a longo prazo tendo horarios totalmente distintos: os xogadores enviaban mensaxes explicando o que pretendían que os seus personaxes fixeran, e o director de xogo respondía co resultado das súas accións en base a tiradas de dados e outros factores. Este estilo de xogo aínda existe a día de hoxe, xa que é o que ofrece unha maior flexibilidade horaria

e tamén unha maior flexibilidade de cara á xogabilidade, posto que os xogadores non están limitados por ningunha norma do sistema: eles mesmos poden expoñer coas súas palabras o que queren facer no xogo.

Por outra banda, no mundo do software enseguida comezaron a desenvolverse programas deseñados especificamente para xogar ao rol, no que non era necesario un director de xogo xa que o propio sistema era o que xestionaba toda a partida; un exemplo disto é o software MUD (Multi-User Dungeon) e todos os seus descendentes, sendo o MUD1 (ano 1978) o primeiro mundo virtual existente[1]. Isto tiña o inconveniente de que o xogador só podía tomar decisións prefixadas no propio sistema e seguir historias totalmente predeseñadas, sen apenas capacidade de improvisación real; a diferencia dos foros de rol, nos que se mantén a figura tradicional do director de xogo que "leva" a partida.

Finalmente, os xogos de rol serviron de gran fonte de inspiración para o desenvolvemento de videoxogos, aparecendo o xénero RPG (*Role-playing game*) que conta con numerosas características propias dos xogos de rol, sendo as máis frecuentes: a creación de personaxes personalizados, a escolla de habilidades e clases personalizadas, e os sistemas de atributos para mellorar as capacidades do personaxe no xogo.

A pesar de todo isto, os xogos de rol clásicos de lapis e papel seguen tendo unha importante popularidade en certos círculos, especialmente entre a xente nova, e a día de hoxe seguen aparecendo e medrando diversos sistemas de xogo.

1.2. Ferramentas informáticas existentes

Como xa se comentou no apartado anterior, existen diferentes variantes dentro dos xogos de rol e para cada unha delas hai bastantes ferramentas dispoñibles. Nomearemos aquí algúns exemplos.

1.2.1. Xogo de rol tradicional

Existen numerosos sistemas de xogo que aínda teñen bastante popularidade a día de hoxe, pero se se inclúe este apartado aquí é pola existencia de ferramentas que simulan unha experiencia tradicional de xogo por ordenador. O exemplo máis paradigmático a día de hoxe é Roll20 [3], unha plataforma de rol que permite realizar partidas de distintos sistemas de xogo en liña, tales como Dungeons and Dragons, World of Darkness e mais. Caracterízase por simular unha partida real en persoa, empregando para isto un mapa, fichas de personaxe e a posibilidade de empregar webcam. O obxectivo dos creadores é poder xogar ao rol tendo unha experiencia o máis próxima posible a xogar en persoa con xogadores reais.

1.2.2. Xogo de rol baseado en texto

Os diversos forks e clons do MUD que foron saíndo co tempo son incontables, mais é especialmente interesante unha versión de MUD orientada a obxectos coñecida como LambdaMOO [5]. Esta ferramenta é moi semellante ao MUD, pero engadindo o paradigma orientado a obxectos: todo o que existe no mundo de xogo son obxectos que interactúan entre si. Á marxe disto, funciona como calquera outro MUD: primeiro o administrador crea o mundo e as súas relacións, e posteriormente os xogadores conéctanse ao servidor para interactuar con el pola súa conta.

1.2.3. Xogo de rol play-by-post

Os xogos de rol do tipo play-by-post, nos que os xogadores se conectan con diferentes horarios sen unha ferramenta informática que os guíe, xeralmente empregan para as súas partidas programas non deseñados para tal fin: grupos de correo, foros, blogues, wikis e redes sociais. Normalmente hai un director de xogo que dirixe a partida, igual que nos xogos tradicionais, pero pode non habelo.

1.2.4. Videoxogo RPG

O xénero RPG dos videoxogos ten unha prolífica colección de títulos, tanto xogos de un só xogador (Dragon Age, Vampire The Masquerade: Bloodlines), como xogos multixogador masivos (World of Warcraft), caracterizados por integrar nun mesmo mundo de xogo a miles de xogadores simultaneamente.

1.3. Xustificación do proxecto

Como se puido ver na sección previa, os xogos de rol ofrecen unha variada oferta de ferramentas e opcións para todos os gustos. Non obstante, analizando o panorama en detalle pódese apreciar unha notable carencia: non hai ferramentas informáticas deseñadas para os xogos de rol play-by-post, isto é, os xogos de rol habitualmente realizados a través de foros. Habitualmente, nestes casos ao que se tende é a empregar un director de xogo que organice a partida e administre a rede na que se realiza a mesma, que pola súa parte terá que ter na súa cabeza o estado actual da mesma, e empregar recursos tales como documentos ou dados físicos para realizar a súa labor.

Este proxecto xurde coa idea de crear unha plataforma orientada a xestionar este tipo de partidas e mellorar a súa experiencia de xogo. O que se pretende é ofrecer a directores de xogo un programa que manteña os datos do mundo de xogo e garanta unha consistencia interna, de forma semellante a como fan os MUDs; pero mantendo o estilo de xogo play-by-post, deixando que a parte software sirva como apoio na partida e non como base da mesma. Neste sentido,

os xogadores seguirían escribindo coas súas palabras as decisións dos seus personaxes, a diferencia do MUD onde os xogadores executan código directamente; e sería o director de xogo o encargado de executar o código pertinente e redactar a resolución das accións dos xogadores para que estes podan lela.

Á marxe do uso deste sistema como ferramenta de ocio, tamén pode ter outras implicacións positivas, tales como o seu uso con fins didácticos. Neste sentido podemos ver dúas posibles aplicacións do sistema: facilitar o achegamento dos xogos de rol na educación, como xa se probou na práctica [4] polo seu fomento do traballo en equipo e a mellora de habilidades sociais e resolución de problemas; ou achegar o mundo da programación informática aos usuarios de rol, debido ao contacto cunha linguaxe de script fácil de entender.

1.4. Obxectivos xerais

O obxectivo deste traballo é o de deseñar e implementar unha ferramenta software que facilite a dirección de partidas de rol asíncrono en liña, empregando para isto unha linguaxe de script propia, é dicir:

- Definir unha linguaxe formal que permita ao usuario crear un mundo virtual mediante a especificación de obxectos distintos cos seus atributos, entre eles os obxectos que representen aos propios personaxes dos xogadores, ademais de permitir definir interaccións preprogramadas entre os propios obxectos. Esta linguaxe debe estar orientada ao xogo de rol, polo que terá en conta elementos clave neste contexto: escenarios (habitacións), visibilidade ou ocultación das accións e sucesos, cálculos para comprobar o resultado das accións, etc.
- Implementar dita linguaxe formal mediante un intérprete de script.
- Deseñar unha base de datos que dote de persistencia ao mundo virtual.
- Deseñar unha interface web para interactuar co sistema de tal modo que o director de xogo poda empregar a linguaxe de script directamente; os xogadores pola contra só precisan dunha caixa de texto para explicarlle ao director as accións que desexan realizar, quedando nas mans deste a interacción co mundo interno.

1.5. Resumo da memoria

O proxecto seguirá unha metodoloxía de desenvolvemento áxil, polo que a maior parte da documentación obterase a partir de comentarios no propio código. A maiores incluiranse documentación adicional como diagramas de clases ou diagramas entidade-relación. O documento estará dividido nos seguintes capítulos (o capítulo 1 é esta propia introdución):

- O capítulo 2 contén a planificación do proxecto, tanto os recursos necesarios como a propia planificación temporal.
- O capítulo 3 é o destinado á especificación de requisitos: definicións, casos de uso, e a lista de requisitos obtidos.
- O capítulo 4 referirase ao deseño do sistema, dende a interrelación dos distintos compoñentes á descrición en detalle do seu funcionamento.

Capítulo 2

Xestión do proxecto

A xestión de proxectos conleva unha serie de metodoloxías e técnicas encamiñadas a planificar temporalmente o proxecto para acadar o seu alcance no prazo previsto, así como garantir a calidade do produto final. Neste capítulo definirase a metodoloxía de desenvolvemento empregada no proxecto, e farase unha estimación de prazos e custos.

2.1. Metodoloxía de desenvolvemento

Unha das decisións máis importantes que se deben tomar ao comezo do proxecto é a metodoloxía de desenvolvemento a empregar. Esta metodoloxía debe permitirnos estruturar o traballo para garantir a súa finalización cuns mínimos de calidade aceptables.

A escolla da metodoloxía fundaméntase nas características do proxecto. neste caso concreto, o noso programa ten unha temática pouco explorada no mercado, e empregará diversas tecnoloxías e recursos relativamente novos para o desenvolvedor. A maiores, este ten unha experiencia reducida na xestión de proxectos grandes. Tendo todas estas cuestións en conta, parece apropiado optar por unha metodoloxía de desenvolvemento áxil e descartar o uso de metodoloxías menos tolerantes ao cambio.

2.1.1. Scrum

Optouse polo emprego de *Scrum* para o desenvolvemento do software. En Scrum, o desenvolvemento realízase de forma incremental mediante iteracións: as tarefas distribúense temporalmente en sprints de unha ou dúas semanas por norma xeral. Poténciase especialmente a interacción co cliente e márcanse uns prazos fixos para cumprir obxectivos, para o cal é preciso priorizar os requisitos regularmente.

Actividades

No modelo de Scrum as reunións xiran en torno aos sprints, e podemos atopar as seguintes actividades:

- Daily Scrum ou Stand-up Meeting: Reunións diarias nas que se revisa o o estado do proxecto por parte do equipo. Neste caso concreto non son relevantes por ser un proxecto dunha soa persoa.
- Sprint Planning Meeting: Reunión de planificación do sprint. Realízase ao inicio do sprint, e nela estipúlase o traballo que se realizará no mesmo.
- Sprint Review Meeting: Reunión de revisión do sprint. Faise ao final do sprint, e revísase o traballo completado e non completado.
- Sprint Retrospective: Na retrospectiva do sprint, os membros do equipo dan as súas impresións sobre o mesmo. Isto ten a finalidade de mellorar o proceso de forma continuada.

2.1.2. Ferramentas de xestión empregadas

Hai numerosas ferramentas dispoñibles para xestionar proxectos de tipo Scrum. Neste proxecto empregarase Acunote, que permite levar o control da lista de requisitos e dos sprints coas súas tarefas.

Acunote

Trátase dunha ferramenta web que ofrece distintas funcionalidades enfocadas na xestión de proxectos con metodoloxías Scrum, tales como a creación e mantemento dunha lista de requisitos priorizada (ou *product backlog*) para o proxecto, xestión de sprints (con nome, data de inicio e fin, e tarefas) e, xa dentro de cada sprint, o control do estado de cada tarefa, incluído o tempo investido nela e o tempo total requerido. A maiores, Acunote ofrece gráficas e estatísticas do proceso xeradas automaticamente para levar o control do mesmo.

2.2. Planificación temporal

A planificación temporal desenvolveuse inicialmente en base ao anteproxecto existente. A metodoloxía Scrum non ofrece unha estruturación concreta do proxecto en fases de vida debido ao seu carácter iterativo, mais pódese facer unha aproximación dunha división de etapas do mesmo segundo o tipo de traballo desenvolvido en cada unha delas.

2.2.1. Fase inicial

Nesta primeira fase desenvolveuse unha descrición extensa do proxecto e das funcionalidades que o produto resultante debe ofrecer. Dividiuse o programa en compoñentes e determinouse a forma de interactuar destes entre si. Ademais disto, tomáronse as decisións necesarias sobre a organización do traballo e a xestión da documentación e de versións.

2.2.2. Fase de análise

A principal tarefa desta fase foi a de realizar unha análise de requisitos a partir de distintas historias de usuario. Definíronse os conceptos chave do sistema de forma máis precisa, e marcáronse formalmente os requisitos e as limitacións do software. Tamén se decidiron as principais tecnoloxías a empregar no software.

2.2.3. Fase de definición da linguaxe

Esta fase estivo centrada no desenvolvemento dunha definición formal da linguaxe do xogo, a linguaxe *COE*, e a súa implementación en Java. Non se inclúe o desenvolvemento do analizador semántico, tarefa que forma parte da seguinte fase.

2.2.4. Fase de desenvolvemento

Na fase de desenvolvemento implementáronse os distintos compoñentes software do proxecto coas súas distintas funcionalidades. Trátase da fase máis longa, e a súa realización documéntase en maior detalle no capítulo pertinente deste documento (capítulo 6).

2.2.5. Fase de probas

Nesta última fase fanse probas unitarias do sistema. A maiores, o sistema instálase nun servidor web para realizar probas con usuarios que poden ser útiles para extraer posibles melloras en usabilidade ou detectar erros.

2.3. Xestión da configuración

Os elementos de configuración son os distintos elementos cos que traballamos no proxecto (en calquera das súas fases) dos cales precisamos manter un control sobre os cambios que se poidan realizar neles, xa que poden afectar sensiblemente ao desenvolvemento do propio proxecto.

Xa que estamos a falar dun proxecto exclusivamente de desenvolvemento de software, será o propio código fonte o que consideraremos elemento de configuración. Para manter a súa integridade e realizar o control de cambios empregaremos a ferramenta Git.

2.3.1. Git

Git [8] é un software de control de versións que nos permite estabelecer repositorios de software e controlar os cambios que se realizan no código. Con Git podemos crear un repositorio para cada elemento independente do proxecto e manter un control dende calquera ordenador no que traballemos. Permite tamén visualizar os cambios realizados en cada *submit* e desfacelos en caso de ser preciso.

GitHub

GitHub é unha ferramenta web para desenvolvedores que ofrece aloxamento gratuíto para repositorios Git. Deste modo, é posible manter o repositorio en liña e traballar co mesmo dende distintos equipos sen necesidade de configurar un servidor web.

2.4. Análise de custos

Os custos deste proxecto recaen principalmente na man de obra, os equipos informáticos e os servidores. Non hai custos adicionais en adquirir paquetes de software, xa que un dos requisitos especificados dende o principio é o uso exclusivo de ferramentas libres.

Adicionalmente, consideramos un 20 % de custos indirectos.

Item	Cantidade	Custo unitario	Custo total
PC	1	1000,00 €	1000,00 €
Persoal	412,5 horas	15,00 €	6187,50 €
VPS	2 meses	20 €	40 €
		Subtotal:	7227,50 €
		Custos indirectos:	+20 %
		Total:	8673 €

Capítulo 3

Análise

Antes de comezar a desenvolver o software deberase facer unha especificación dos requisitos que este debe cumprir. Habitualmente, nas metodoloxías tradicionais de desenvolvemento de software, faise unha busca exhaustiva de requisitos no comezo do proxecto, coa intención de realizar o menor número de cambios posible aos mesmos. En Scrum, por ser unha metodoloxía áxil esta formulación cambia radicalmente: os requisitos introdúcense nun *Backlog*, a lista priorizada de requisitos propia desta metodoloxía, e vanse modificando segundo sexa pertinente por cada iteración.

Polo xeral, os requisitos en Scrum só se analizan en profundidade antes de realizar o sprint no que serán cumpridos. Debido a isto, a interacción co cliente é importante e é habitual que os requisitos muden varias veces ao longo do proxecto, sen por iso ter que realizar custosos procesos de reaxuste.

3.1. Definicións

Antes de comezar cos requisitos será necesario definir correctamente todos os termos que se empregarán neste capítulo e tamén en capítulos posteriores. Xa que os termos fan referencia a conceptos moi concretos dos xogos de rol, unha boa definición é indispensable para poder entender correctamente o documento.

Partida

Cada unha das instancias independentes do software na que poden participar os usuarios. A partida é por tanto a suma dun mundo de xogo máis os usuarios que participan nel. Cada partida ten inicialmente un director de xogo e varios xogadores. Non se contempla a comunicación entre distintas partidas, cada unha funciona de xeito independente. Unha mesma execución do motor de xogo pode manter varias partidas funcionando simultaneamente.

Mundo de xogo

O mundo de xogo (ou simplemente mundo) é a abstracción de todos os obxectos e clases correspondentes a unha partida. Pódese distinguir entre a definición do mundo e o seu estado:

- Definición: O conxunto das clases descritas polos directores de xogo.
- Estado: O conxunto dos obxectos instanciados a partir das clases.

Xogador

Usuario que está asociado a un determinado personaxe e toma decisións en base a este. Os xogadores non poden modificar o mundo directamente, senón que as súas interaccións co mundo realízanse a través dos personaxes.

Director de xogo (DX)

Usuario con privilexios que pode alterar o mundo de forma directa, e resolver as decisións dos xogadores. Poden tanto alterar a definición do mundo (creando e modificando clases) como cambiar o estado do mesmo (instanciando obxectos e modificándoos). Os directores de xogo non precisan ter un personaxe propio.

Clase

No ámbito do mundo de xogo, abstracción dun conxunto de obxectos con atributos e métodos comúns, seguindo o paradigma da programación orientada a obxectos. Os directores de xogo poden crear e modificar estas clases.

Un exemplo de clase sería *espada*, *porta*, *cabalo* ou *guerreiro*.

Obxecto

No ámbito do mundo de xogo, cada unha das instancias dunha clase concreta. Os obxectos *existen* no mundo, e sitúanse nun determinado escenario. É labor dos directores de xogo crealos, modificalos e destruílos. Todos os obxectos posúen un identificador global numérico dentro do mundo.

Un exemplo de obxecto sería *a segunda mesa da posada*, *o personaxe dun dos xogadores* ou *o can que acompaña ao grupo de personaxes*.

Personaxe

Obxecto asociado a un determinado xogador. Toda a información percibida polo personaxe será posta en coñecemento do xogador correspondente, e toda decisión tomada polo xogador determinará as accións do personaxe (isto último baixo a supervisión do DX).

Decisión

Descrición verbal dun xogador explicando a acción que pretende que realice o seu personaxe. As decisións están redactadas en linguaxe informal, pero con suficiente información como para que o director de xogo poda entender o que o xogador pretende. É labor do director de xogo xerar as accións a partir das decisións dos xogadores.

Un exemplo de decisión sería: *"O meu personaxe achégase á espada incrustada na pedra, di en voz alta 'alá vou' e trata de arrincala."*

Acción

Conxunto de liñas de código que alterarán o mundo de xogo nun determinado escenario. O código da acción é escrito por un director de xogo segundo o que interprete lendo as decisións dos xogadores que se atopen no escenario correspondente (ou pola súa propia iniciativa se non hai xogadores).

Narración

Texto redactado polo DX asociado a unha determinada acción. Os xogadores cuxos personaxes se atopen no escenario no momento da narración terán acceso a esta. Esta é a principal vía que teñen os xogadores de recibir información do mundo.

Rexistro de personaxe

Todo o que perciba un personaxe determinado (as narracións ás que teña acceso) formará un rexistro. O xogador ten acceso a este rexistro, polo que poderá consultar en todo momento todo o que o seu personaxe puido oír, ver ou percibir de calquera outro xeito.

Escenario

Cada un dos recintos pechados nos que se divide o mundo de xogo. Tamén coñecidos como *habitacións* ou *estancias* noutros sistemas semellantes. Todos os obxectos están situados nun escenario, incluídos os personaxes, e en calquera momento o DX pode cambialos de escenario mediante o código. Os escenarios pódense agrupar en *rexións* mediante nomes compostos.

Exemplos de escenarios poderían ser */Capital/mercado*, */montañas/minaabandonada/terceiraseccion* ou */casteloantigo/segundaplanta/habitacion1*.

Tirada

Resultado aleatorio que representa unha tirada de dados. A tirada especifica o número de dados tirados e o número de caras de cada dado. Este resultado

emprégase xeralmente para determinar o éxito ou o fracaso das accións dos personaxes ou doutros obxectos.

Inventario

Os obxectos teñen a capacidade de almacenar outros obxectos. Para iso existe o concepto do inventario: un inventario é en esencia unha localización especial de obxectos (semellante aos escenarios) vinculada a un obxecto específico. Deste modo, un obxecto A pode dispoñer dun inventario que contén aos obxectos B e C. Á hora da xogabilidade, os personaxes contidos nun inventario considéranse situados no *escenario real* do obxecto que os contén. Un exemplo de uso de inventarios é *un cofre do tesouro*, *un barril* ou un *xogador co seu inventario*. Un exemplo de inventarios diferenciados para o mesmo obxecto pode ser *un xogador no que se diferencien os obxectos que leva nas mans dos que leva nas costas*.

3.2. Historias de usuario

As historias de usuario son características das metodoloxías áxiles. Trátase de breves descricións do que os distintos usuarios queren que o sistema lles permita facer, e axudan á hora de extraer requisitos para os sprints.

3.2.1. Descrición de roles

Inicialmente teremos que definir os roles cos que interactuar co sistema. Identificamos dous roles fundamentais: o director de xogo e o xogador.

- **Director de xogo:** Este actor correspóndese coa definición descrita previamente; o seu cometido por tanto será dirixir a partida, recibindo decisións dos xogadores e transcribíndoas como accións dos seus personaxes.
- **Xogador:** Calquera usuario que participe na partida cun personaxe. Redacta decisións e envíaas ao DX.

3.2.2. Lista de historias de usuario

As seguintes historias de usuario elaboráronse a partir de entrevistas con usuarios potenciais. Son situacións típicas que se darán durante a execución do software.

HU-01: Alta de xogadores

Como xogador, quero poder rexistrarme no sistema coa finalidade de poder acceder a unha partida.

HU-02: Definición dunha nova clase

Como director de xogo, quero poder definir unha clase nova no mundo de xogo, para poder crear obxectos da mesma no futuro.

HU-03: Creación de personaxe

Como director de xogo, quero poder asociar un obxecto a un xogador concreto, para que poda dispoñer dun personaxe propio.

HU-04: Envío de decisións

Como xogador, quero poder enviarlle as miñas decisións ao DX, para que este as teña en conta á hora de resolver accións.

HU-05: Resolución de accións

Como director de xogo, quero poder ler as decisións dos xogadores, para deste modo saber como resolver as accións.

HU-06: Creación dun escenario

Como director de xogo, quero crear escenarios para poder introducir obxectos neles no futuro.

HU-07: Cambio de escenario dun obxecto

Como director de xogo, quero mover obxectos dun escenario a outro, para poder desprazar aos xogadores polo mundo de xogo.

3.3. Requisitos funcionais

Requisito FN.01

Título: Alta de usuarios
Descrición: A aplicación debe permitir que se creen novas contas de usuario.
Importancia: Esencial

Requisito FN.02

Título: Modificación de usuarios
Descrición: A aplicación debe permitir que se modifiquen os datos (enderezo

electrónico, contrasinal) de contas de usuario existentes.
Importancia: Esencial

Requisito FN.03

Título: Identificación de usuarios
Descrición: A aplicación debe permitir que os usuarios se identifiquen correctamente, e se recoñeza o seu rol dentro do sistema.
Importancia: Esencial

Requisito FN.04

Título: Creación de clases
Descrición: A aplicación debe permitir ao director de xogo crear clases novas no mundo.
Importancia: Esencial

Requisito FN.05

Título: Modificación de clases **Descrición:** A aplicación debe permitir ao director de xogo modificar clases existentes.
Importancia: Esencial

Requisito FN.06

Título: Eliminación de clases
Descrición: A aplicación debe permitir ao director de xogo eliminar clases existentes. A aplicación debe garantir que non existan obxectos non asociados a ningunha clase.
Importancia: Esencial

Requisito FN.04

Título: Herdanza de clases
Descrición: A aplicación debe permitir que unhas clases sexan fillas de outras, herdando deste xeito os seus métodos e campos.
Importancia: Esencial

Requisito FN.07

Título: Creación de obxecto
Descrición: A aplicación debe permitir crear novos obxectos dende as clases xa existentes. Estes obxectos créanse no contexto dun escenario determinado.
Importancia: Esencial

Requisito FN.08

Título: Modificación de obxecto
Descrición: A aplicación debe permitir modificar o valor dos campos de obxectos existentes.
Importancia: Esencial

Requisito FN.09

Título: Eliminación de obxecto
Descrición: A aplicación debe permitir eliminar obxectos nos escenarios.
Importancia: Esencial

Requisito FN.10

Título: Ligazón de obxecto personaxe a xogador
Descrición: A aplicación debe permitir que un determinado obxecto se asocie a un xogador, para deste xeito permitir ao xogador ver as narracións das súas accións e enviar as súas decisións á estancia axeitada.
Importancia: Esencial

Requisito FN.11

Título: Creación dun escenario baleiro
Descrición: A aplicación debe permitir ao director de xogo crear un novo escenario de cero que non conteña obxectos.
Importancia: Esencial

Requisito FN.12

Título: Eliminación dun escenario
Descrición: A aplicación debe permitir ao director de xogo eliminar un escenario

do mundo. Todos os obxectos que conteña o escenario serán eliminados.
Importancia: Opcional

Requisito FN.13

Título: Creación dun modelo de escenario
Descrición: A aplicación debe permitir ao director de xogo deseñar un modelo de escenario. Este modelo conterá código que se executará no momento de crear a estancia.
Importancia: Opcional

Requisito FN.14

Título: Creación dun escenario dende modelo **Descrición:** A aplicación debe permitir ao director de xogo crear un novo escenario empregando un modelo de escenario, executando o código que este modelo contén.
Importancia: Opcional

Requisito FN.15

Título: Cambio de escenario dun obxecto
Descrición: A aplicación debe permitir ao director de xogo mover un obxecto dun escenario a outro. O identificador global do obxecto permanece igual.
Importancia: Esencial

Requisito FN.16

Título: Envío de decisións
Descrición: A aplicación debe permitir aos xogadores enviar as súas decisións para que o DX poda tratalas.
Importancia: Esencial

Requisito FN.17

Título: Visualización de narracións
Descrición: A aplicación debe permitir que os xogadores podan ver os resultados das accións que involucren aos seus personaxes, sexan accións causadas polas súas decisións ou non.
Importancia: Esencial

3.4. Restricións de deseño

Requisito RD.01

Título: Uso de ferramentas libres
Descrición: A aplicación debe realizarse empregando unicamente ferramentas libres, como unha forma non só de reducir custos, senón tamén de permitir unha licencia libre do propio software resultante.
Importancia: Esencial

3.5. Requisitos non funcionais

Requisito NF.01

Título: Linguaxe: Lóxica de escenario
Descrición: A linguaxe debe proporcionar ao DX a capacidade de crear e eliminar escenarios, mover obxectos dun escenario a outro, obter a lista de personaxes nun escenario, e asignar variables locais para nomear aos obxectos ou almacenar datos de tipos primitivos.O código executado nun escenario non debe afectar nun principio a outros escenarios diferentes, salvo no movemento de obxectos entre eles.
Importancia: Esencial

Requisito NF.02

Título: Linguaxe: Lóxica de visibilidade e ocultación
Descrición: A linguaxe debe proporcionar a capacidade de determinar que obxectos son visibles por un personaxe e cales non mediante o uso de condicións. Isto non ten un efecto real na xogabilidade (un personaxe pode interactuar cun obxecto cuxa existencia descoñece se así o determina o DX no seu código), mais é relevante á hora de ofrecer información aos xogadores.
Importancia: Opcional

Requisito NF.03

Título: Linguaxe: Lóxica de inventario
Descrición: A linguaxe debe permitir que uns obxectos se conteñan aos outros. Na práctica isto ten varias consecuencias, sendo a máis importante o feito de que o obxecto dependente deba estar forzosamente no mesmo escenario co obxecto recipiente. Tamén ten o seu efecto na visibilidade, xa que se un personaxe non

pode ver ao recipiente nunca poderá ver os obxectos que contén. A linguaxe debe ofrecer a capacidade de especificar as clases que conteñen inventarios, métodos para introducir uns obxectos noutros, e tamén proporcionará a capacidade de recibir a lista de obxectos dun inventario.
Importancia: Esencial

Requisito NF.04

Título: Linguaxe: Tipos de datos básicos
Descrición: A linguaxe debe implementar como tipos primitivos os números (enteiros ou en punto flotante) e as cadeas de texto. Tamén deben implementarse os arrays, sexan de números, cadeas de texto ou obxectos.
Importancia: Esencial

Requisito NF.05

Título: Linguaxe: Lóxica de eventos
Descrición: A linguaxe debe ofrecer a opción de definir código de clase que se executa automaticamente ante determinados eventos, tales como o cambio de rolda, a chegada de novos obxectos ao escenario ou o cambio de escenario do propio obxecto.
Importancia: Opcional

Requisito NF.06

Título: Linguaxe: Condicionais e tiradas de dados
Descrición: A linguaxe debe ofrecer a capacidade de empregar condicións lóxicas para alterar o fluxo do código. Polo outro lado, a linguaxe debe prover dun método de obter valores aleatorios para simular as tiradas de dados, nas que o DX poda especificar o número de dados lanzados e o número de caras de cada dado.
Importancia: Esencial

Requisito NF.07

Título: Linguaxe: Iteracións
Descrición: A linguaxe debe prover de funcións que permitan traballar en listas de elementos de forma iterativa, tratando cada elemento por separado.
Importancia: Esencial

Requisito NF.08

Título: Linguaxe: Operacións lóxicas e matemáticas
Descrición: A linguaxe debe ofrecer as operacións lóxicas e matemáticas básicas, tales como comparacións, sumas e multiplicacións. Tamén debe permitir a operación en arrays, sexa entre dous arrays ou entre un array e un número (elemento a elemento).
Importancia: Esencial

Requisito NF.09

Título: Linguaxe: Prioridade de decisións
Descrición: A linguaxe debe permitir definir a orde na que se resolverán as decisións dos xogadores mediante funcións de comparación. No momento de resolver as decisións, estas aparecerán ante o DX ordenadas como corresponda.
Importancia: Opcional

Requisito NF.10

Título: Resultado textual: Seccións comúns e particulares
Descrición: No momento de mostrar o texto resultante dunha acción, o DX debe ter a opción de escribir unha narración común en 3ª persoa para todos os personaxes, e narracións personalizadas en 2ª persoa para cada xogador por separado.
Importancia: Esencial

Requisito NF.11

Título: Resultado textual: Condicións
Descrición: No momento de mostrar a narración, o DX deberá ter a opción de definir condicións en determinados bloques de texto. Os xogadores que non cumplan tales condicións non poderán ler eses bloques de texto.
Importancia: Esencial

Requisito NF.12

Título: Resultado textual: Formato especial
Descrición: No momento de mostrar o texto resultante dunha acción, o DX poderá empregar formato especial, tal como letra en negriña ou cursiva, ou o uso

de imaxes e enlaces web.
Importancia: Opcional

Requisito NF.13

Título: Resultado textual: Anonimato
Descrición: No momento de mostrar o texto resultante dunha acción, os xogadores non lerán directamente o nome dos personaxes mencionados, senón que verán o nome co que eles coñezan a tales personaxes.
Importancia: Opcional

3.6. Requisitos de proxecto

Requisito PR.01

Título: Data límite do proxecto
Descrición: O proxecto deberá rematarse antes do 8 de febreiro do 2017, data establecida no regulamento de traballos fin de grao para GREI da USC.
Importancia: Esencial

3.7. Requisitos de calidade

Requisito CA.01

Título: Internacionalización
Descrición: O software deberá estar deseñado de tal xeito que se poda traducir facilmente a distintos idiomas sen realizar cambios no código.
Importancia: Opcional

Requisito CA.02

Título: Manual de usuario
Descrición: O software resultante deberá ir acompañado dun manual de usuario que explique o funcionamento do mesmo, incidindo especialmente no uso da linguaxe propia.
Importancia: Esencial

Requisito CA.03

Título: Protección contra ataques
Descrición: O software resultante deberá estar protexido contra os principais ataques, como por exemplo a inxección de código SQL e a introdución de código HTML e Javascript non intencionado.
Importancia: Esencial

3.8. Requisitos de almacenamento

Requisito AL.01

Título: Persistencia da información
Descrición: A aplicación debe garantir que a información do mundo (clases, obxectos, escenarios) se conservará cando a aplicación remate e volva a poñerse en funcionamento.
Importancia: Esencial

Capítulo 4

Arquitectura e ferramentas

Neste capítulo describirase a arquitectura do sistema, detallando os distintos compoñentes que a forman e como interactúan entre eles. Ademais diso, tamén se realizará unha análise das tecnoloxías empregadas no proxecto e os motivos da súa escolla entre as ferramentas dispoñibles no mercado.

4.1. Arquitectura do sistema

A arquitectura do proxecto xa foi parcialmente descrita no anteproxecto, e o seu aspecto máis importante é a separación do motor de xogo e a interface gráfica en dous elementos diferenciados.

O sistema está orientado a realizar partidas de rol por internet. Tal e como a interacción co usuario foi formulada previamente, o usuario (neste caso o director de xogo) interactúa co sistema mediante o uso de scripts, mais non é a única forma: a interface debe proporcionar facilidades para realizar as súas tarefas sen deixar todo en mans da linguaxe. Este programa diferénciase dos MUDs tradicionais entre outras cousas na forma de entender a interacción persoa-ordenador, evitando interfaces áridas para deste modo poder chegar a unha maior masa de xogadores que poden ter coñecementos limitados ou nulos de programación informática.

Debido a isto, é fundamental asumir a existencia de dous elementos de software executable diferenciados: o motor de xogo, que chamaremos **Demiurgo**, e a interface de usuario. Ambos elementos describiranse máis en detalle na sección 4.2.

Cabe destacar que se asumiu en todo momento o motor de xogo como a prioridade do proxecto, e dentro deste, a elaboración da linguaxe de script para o usuario, polo que foi o compoñente ao que se destinou unha maior carga de traballo.

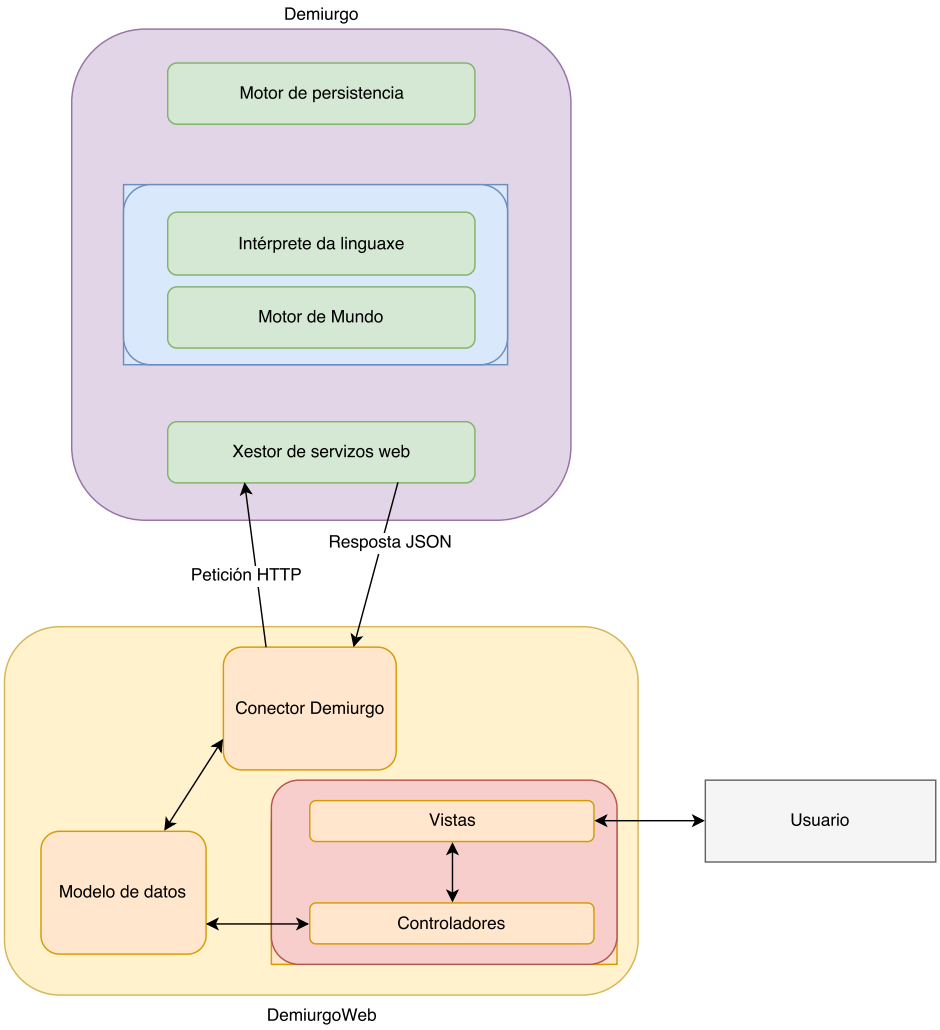

Figura 4.1: Diagrama da arquitectura do sistema.

4.2. Bloques da arquitectura

Como xa se mencionou, o sistema está inicialmente dividido en dous grandes bloques independentes: o motor de xogo e a interface. A maiores, cada un destes compoñentes ten unha arquitectura interna que será descrita a continuación.

4.2.1. Motor de xogo: Demiurgo

O motor de xogo é o elemento central do sistema. Foi bautizado co nome de **Demiurgo**. As súas principais funcións son manter o mundo (ou mundos) de xogo en memoria, mantendo ademais a súa coherencia e integridade; realizar o procesamento dos scripts para modificar o seu estado; e por último, xestionar os usuarios e o acceso á información. Podemos distinguir os seguintes compoñentes diferenciados dentro do motor de xogo:

Motor de Mundo

O motor de Mundo mantén a consistencia dos mundos das distintas partidas activas. Garda unha instancia activa de cada clase de xogo, obxecto, escenario e demais elementos propios da partida, e ofrece métodos para poder modificar o seu estado sen comprometer a súa coherencia interna. As clases empregadas por este Motor de Mundo correspóndese co *modelo* do sistema dentro dun patrón modelo-vista-controlador.

Intérprete da linguaxe

O intérprete realiza a análise léxica, sintáctica e semántica do código script enviado polo director de xogo, para deste modo extraer as súas instrucións e alterar o mundo de xogo segundo lle sexa indicado. O intérprete por tanto ten acceso ao Motor do Mundo, co que se comunica durante a lectura do código continuamente. No capítulo 5 especifícase con máis detalle o funcionamento deste intérprete.

Xestor de servizos web

Ofrece un *Endpoint* á interface web sobre o que facer peticións HTTP para estabelecer o diálogo co motor de xogo. Conta con todos os métodos necesarios para poder enviar e recibir información co Demiurgo, dende o envío de instrucións e código script ata as distintas peticións de información sobre o estado do mundo. As peticións devolven por norma xeral un obxecto JSON, e poden recibir como argumentos campos de texto (cando se emprega o método *GET*) ou un obxecto JSON (cando se emprega o método *POST*).

Motor de persistencia

O motor de persistencia comunícase do xestor da base de datos para manter no disco duro unha copia do modelo. A súa función por tanto é transformar os obxectos do Motor de Mundo a linguaxe relacional para deste modo almacenar o estado do mundo cando o sistema se detén, e facer o proceso contrario para recuperar tal estado cando o sistema volve a poñerse en funcionamento.

4.2.2. Interface web: DemiurgoWeb

Os usuarios do sistema precisan un medio para comunicarse co Motor de Xogo dunha forma cómoda e fácil de entender. Neste proxecto asumiuse a creación dunha aplicación web para suplir esta función, á que se denominou DemiurgoWeb, que permitirá tanto a directores de xogo como a xogadores interactuar co sistema. Esta interface web non realizará traballos complexos nin levará o control dos privilexios de usuarios: a súa función é exclusivamente facer de ponte co Demiurgo enviando peticións HTTP e mostrando resultados por pantalla aos usuarios.

Demiurgo e DemiurgoWeb deberán desenvolverse como dous programas totalmente diferentes e desacoplados, de tal xeito que poidan crearse outras vías para acceder ao motor de xogo, tales como apps de móbil, por exemplo.

O DemiurgoWeb implementarase en base ao patrón modelo-vista-controlador, polo que se deben analizar estas tres partes por separado. A maiores, conta cun compoñente adicional chamado *conector Demiurgo*, que se encarga de realizar todas as peticións ao Demiurgo.

Modelo de datos

O modelo de datos representa todos os tipos de datos que o sistema manexa: clases de xogo, obxectos, escenarios, etcétera. No DemiurgoWeb estas clases deberán ser directamente transformables en obxectos JSON e viceversa, xa que obterá todos os seus datos a través do conector Demiurgo neste formato.

Vistas

As vistas son elementos deseñados para mostrar unha representación visual da información aos usuarios. Non modifican os datos, a súa única función é expresalos de forma clara e facilitar a interacción con eles.

Controladores

Os controladores, seguindo co patrón, serán as clases intermedias que obteñan os obxectos do modelo para proporcionarllos ás vistas a partir das accións do usuario.

Conector Demiurgo

O conector Demiurgo está fóra do patrón modelo-vista-controlador. A súa única función é realizar as peticións ao motor de xogo Demiurgo (concretamente, ao seu xestor de servizos web) dunha forma centralizada cando os controladores llo requiran. A URL do motor de xogo e o porto deberán serlle especificados ao conector mediante ficheiros de configuración.

4.3. Linguaxe: COE

Un pilar fundamental deste proxecto, tanto pola súa carga de traballo como pola súa relevancia dentro do sistema, é a elaboración dunha linguaxe de script nova para manexar o mundo de xogo por parte do DX. Esta foi a tarefa de maior prioridade, polo que se definiu e implementou antes de comezar co resto de compoñentes.

Inicialmente valorouse a posibilidade de empregar algunha linguaxe xa existente para evitar o esforzo de crear unha nova. Isto tería o beneficio de aforrar moitísimo tempo e traballo, permitindo dedicar máis tempo a outros aspectos do proxecto como a interface web. A pesar diso, a decisión final foi a de crear unha linguaxe de cero, xa que o concepto do programa non ten precedentes e ningunha das linguaxes dispoñibles cubriría correctamente os distintos aspectos requeridos sen sacrificar facilidade de uso.

A linguaxe recibiu o nome de **COE**, acrónimo de *Código de Obxectos e Escenarios*. No apéndice B deste documento pódese atopar un manual desta linguaxe orientado ao seu uso por parte de directores de xogo.

COE conta cunha gramática de tipo LL [9], que é analizada por un analizador sintáctico descendente de esquerda a dereita.

A sintaxe da linguaxe COE está fortemente inspirada na de linguaxes amplamente difundidas como C ou Java. Está deseñada tendo en mente a súa finalidade e buscando a maior simplicidade posible. É importante resaltar que, no programa, o código sempre se executa nun de dous posibles contextos:

- No contexto de definición dunha clase, no que o script se empregará exclusivamente en definir os campos e métodos dunha clase determinada.
- No contexto dunha acción nun escenario, no que o script actuará sobre os obxectos e variables do propio escenario onde se executa.

4.4. Análise tecnolóxica

Antes de comezar con este proxecto foi fundamental realizar unha busca exhaustiva para coñecer as diferentes ferramentas dispoñibles no mercado. Non obstante, isto non se detivo en ningún momento do desenvolvemento do mesmo,

xa que a medida que o proxecto avanzaba era inevitable que apareceran cambios nos requisitos, ou simplemente descubrir novas tecnoloxías dispoñibles, o que obrigaba a tomar decisións sobre a incorporación de novas ferramentas.

4.4.1. Linguaxe de programación: Java SE

Antes de comezar co desenvolvemento, unha das primeiras decisións que era necesario tomar era a escolla dunha linguaxe de programación concreta para os distintos compoñentes. Valoráronse distintas posibilidades tanto para o motor de xogo como para a interface web.

No caso do motor de xogo, era necesaria unha linguaxe potente e que ofrecera un rendemento aceptable. Unha prioridade era que tivera dispoñibles librarías opensource (un requisito do proxecto) para poder xerar linguaxes propias, xa que era imprescindible para a elaboración da linguaxe COE.

Considerouse no seu momento momento a posibilidade de empregar unha linguaxe funcional como Scala. Esta idea foi finalmente descartada debido a que a falta de experiencia con este tipo de linguaxes e a menor documentación provocarían dificultades e retrasos nun proxecto xa de por si complexo.

Para o DemiurgoWeb valorouse empregar PHP como linguaxe, xa que a experiencia do programador con esta linguaxe é significativa, e resulta máis sinxelo e económico montar un servidor Apache para un programa en PHP que un servidor Tomcat para Java. Descartouse tamén pola dificultade de manter o código dun proxecto grande en PHP en comparación cun servidor en Java.

Finalmente a linguaxe escollida foi Java. Os principais argumentos a favor para realizar esta escolla foron os seguintes:

- Java é unha linguaxe amplamente difundida, cunha extensa documentación na rede. Ademais disto, o programador deste proxecto está relativamente familiarizado coa súa sintaxe.

- Dispón de numerosas librarías que permiten desenvolver os distintos compoñentes do proxecto. Dispón en particular de ferramentas para desenvolver analizadores léxico-sintácticos, e de librarías para montar sistemas web.

- É compatible coa maioría de entornos de desenvolvemento, nomeadamente Eclipse, entorno que se empregará neste proxecto.

4.4.2. Entorno de desenvolvemento: Eclipse

Aínda que non forme parte do entregable final, é importante comentar a elección do entorno do desenvolvemento polo seu impacto crítico no desenvolvemento do proxecto. A decisión final estaba entre Eclipse e Netbeans, xa que son as dúas linguaxes máis próximas á linguaxe Java e que ofrecen máis ferramentas útiles.

Finalmente optouse por Eclipse, xa que conta con numerosos plugins para facilitar o desenvolvemento dos distintos compoñentes do sistema, a linguaxe COE entre eles.

4.4.3. Xerador de parsers: ANTLR

Como xa se comentou previamente, a definición e implementación da linguaxe COE é un dos aspectos críticos deste proxecto, requerindo a maior parte da carga de traballo do mesmo. Debido a isto, foi necesario buscar entre as distintas ferramentas do mercado para atopar unha que facilitara a interpretación do código, e evitar deste modo a necesidade de codificar un analizador léxico-sintáctico manualmente (co esforzo e tempo desmesurados que iso requeriría).

Escolleuse ANTLR como ferramenta para deseñar a linguaxe e xerar os correspondentes analizadores léxico e sintáctico. ANTLR trátase dunha ferramenta que a partir dunha gramática LL(1) elabora un analizador sintáctico descendente; grazas a isto podemos centrarnos no deseño da gramática para que satisfaga os requisitos do proxecto, e deixar en mans de ANTLR o proceso de implementación do *parser* (recoñecedor de linguaxe).

O principal motivo polo que se escolleu ANTLR sobre outras ferramentas foi a súa capacidade de xerar código en Java de forma directa. Hai dispoñible un manual de usuario de ANTLR 4 (a versión empregada neste proxecto) [**?**].

4.4.4. Apache Maven

Tanto no Demiurgo como no DemiurgoWeb, decidiuse empregar Maven para xestionar os diversos paquetes e dependencias, así como os distintos elementos software de ambos compoñentes. As principais vantaxes disto son:

- Ofrece unha estrutura de directorios estandarizada que facilita o mantemento do código de cara ao futuro do proxecto.
- Simplifica a xestión de dependencias, descargando de forma automática os paquetes necesarios e aforrando tempo neste aspecto.
- Simplifica a execución de tests no código.

4.4.5. Servizos REST

Un aspecto importante do sistema era a forma na que a interface web (e outras posibles interfaces futuras non contempladas no alcance deste proxecto) se comunicaría co motor de xogo. Unha primeira aproximación suxeriu o uso de Java RMI para comunicar ambos compoñentes, aproveitando o feito de que ambos corrían sobre JVM. Non obstante, optouse finalmente por descartar esta opción e empregar servizos web de tipo REST en base ás seguintes cuestións:

- Java RMI só funciona entre programas que corran sobre JVM. A pesar de non ser este un problema no alcance actual deste proxecto, limita a evolución do sistema no futuro; empregando servizos web ábrese a porta a comunicarse co servidor de xogo mediante outras aplicacións desenvolvidas noutras tecnoloxías.

- Java RMI é unha tecnoloxía menos flexible cós servizos web, de mantemento máis complexo e que pode xerar dificultades cando os dous servidores se atopan en distintas ubicacións físicas. Pola contra, os servizos web só requiren dunha conexión vía HTTP entre ambos servidores, polo que son máis fáciles de empregar.

- Con Java RMI é necesario compartir código entre ambos servidores a través de librarías comúns para poder compartir POJOs (*Plain Old Java Objects*). Mediante os servizos web, en cambio, a comunicación pode realizarse mediante obxectos JSON, totalmente independentes do entorno.

Os servizos REST esencialmente fundaméntanse nos propios métodos de HTTP: GET, POST, PUT e DELETE. As peticións web son simples peticións por HTTP que conteñen obxectos JSON no corpo da mensaxe. Para este proxecto, todos os métodos son de tipo POST ou GET, pero poderían empregarse métodos distintos se se considerase necesario; a elección do método baséase no seguimento duns estándares, mais non afecta realmente ao funcionamento.

4.4.6. Servidor web do motor de xogo: Grizzly

O motor de xogo precisa de ofrecer un *endpoint* ao que se lle poidan facer peticións web, polo que foi necesario buscar unha solución para isto que permitira manter a funcionalidade do motor intacta. A escolla dos servizos REST sobre Java RMI realizouse nun momento no que o Demiurgo xa tiña un certo nivel de profundidade, polo que un requisito para a ferramenta a escoller era que non requerira modificar severamente o código.

Grizzly é unha ferramenta que permite que un executable Jar normal poida montar un servidor web sen necesidade de usar contenedores como Tomcat. Nese sentido, escolleuse por tratarse da opción máis sinxela para resolver o problema. Ademais disto, outro factor determinante foi a facilidade para empregar Grizzly en conxunto con Jersey, unha ferramenta para crear servizos REST.

4.4.7. Xestor de servizos REST: Jersey

Jersey e Grizzly son dúas librarías que foron escollidas simultaneamente xa que, como se comentou na subsección de Grizzly, son librarías que ofrecen gran facilidade para empregarse xuntas, e hai numerosos manuais en internet para crear servidores con estas dúas ferramentas. A finalidade de Jersey neste sentido

é a de crear *endpoints* nos que implementar os servizos REST que precisemos, os cales se encargará Grizzly de publicar nun servidor para recibir peticións.

4.4.8. Spring Framework

Spring é un Framework de Java que facilita o desenvolvemento de aplicacións grandes. A principal característica de Spring é a inxección de dependencias, que simplifica o control do ciclo de vida dos obxectos: crear os distintos compoñentes, inicializalos e manter as referencias entre eles. A maiores, dispón de módulos útiles para todo tipo de tarefas, tales como montar servidores web, empregar o patrón modelo-vista-controlador ou executar tests de JUnit.

Neste proxecto empregaremos Spring no DemiurgoWeb. Debido a isto, será necesario invertir algo de tempo en familiarizarse co funcionamento de Spring, pero unha vez superado isto axilizará enormemente a implementación do compoñente web e o seu mantemento.

Pódese atopar máis información sobre Spring na súa páxina web [11].

Spring Boot

Spring Boot é un módulo de Spring que permite crear ficheiros Jar executables sen necesidade de empregar Tomcat ou ferramentas similares; Spring Boot introduce Tomcat no interior do ficheiro Jar para obter un efecto semellante ao que conseguimos no Demiurgo con Grizzly. Ademais disto, é posible configuralo para empregar ficheiros WAR que funcionen como executables e tamén para montar en contenedores web, ofrecendo maior versatilidade. Deste modo, aforramos tempo ao evitar os erros que se puideran cometer na configuración de contenedores web.

Spring Web

Spring ofrece numerosas ferramentas para o desenvolvemento de aplicacións web. No canto de precisar servlets para as distintas webs, Spring ofrece un servlet propio que xestiona as distintas peticións e as redirixe aos controladores e vistas correspondentes, sen necesidade de ficheiros de configuración adicionais. Este módulo, e o Framework en conxunto, facilitan moito a división do código en modelo, vista e controlador.

Thymeleaf

Thymeleaf é un motor que nos permite enriquecer o código HTML das vistas de Spring para dotalo de funcionalidade, e deste modo mostrar os datos do modelo que o controlador lle proporciona. Inicialmente considerouse empregar JSP para este fin, pero a opción final foi Thymeleaf por varios motivos:

- Thymeleaf é unha linguaxe máis nova e con maior potencia que JSP.

- Thymeleaf é plenamente compatible con Spring e a interacción entre ambos é moi natural.

- Thymeleaf emprega ficheiros HTML e non precisa de etiquetas adicionais, a diferencia de JSP, que emprega ficheiros JSP coas súas propias etiquetas; isto permite que os ficheiros HTML poidan ser visualizados directamente nun navegador con datos de proba sen necesidade de montar o servidor.

4.4.9. Javascript

No apartado web empregouse Javascript para ofrecer contidos dinámicos nas distintas páxinas, e deste modo ofrecer unha interface máis amigable de cara ao usuario. A día de hoxe a inmensa maioría de usuarios empregan Javascript na súa navegación habitual.

JQuery

JQuery é unha libraría de Javascript que ofrece numerosas facilidades para facer accións complexas dentro da páxina. A maiores, moitos plugins que engaden funcionalidades avanzadas empregan JQuery como dependencia.

JQuery UI: JQuery UI é un plugin de JQuery que facilita o uso de *drag-and-drop* na páxina (arrastrar e soltar). Empregouse en determinados puntos da web para facilitar a elaboración de scripts por parte do DX.

CodeMirror

CodeMirror é un plugin baseado en Javascript nativo que ofrece unha área de texto destinada á escritura de código con funcionalidades que o *textarea* de HTML non ofrece, como resaltar a sintaxe do código ou acceder a posicións do texto mediante liña e columna, por exemplo. Moi útil á hora de redactar código COE.

4.4.10. Base de datos: MariaDB

Un dos requisitos do proxecto era a persistencia dos datos unha vez o sistema deixa de funcionar. Para iso é necesario o uso de bases de datos, e con elas, de sistemas xestores de bases de datos.

MariaDB é un xestor de bases de datos derivado de MySQL. Optouse polo uso de MariaDB pola súa plena compatibilidade con MySQL e pola súa alta velocidade.

4.4.11. Control de versións: Git

Git é unha ferramenta para desenvolvedores que xestiona os cambios realizados no código, mantendo un control de versións e facilitando o desenvolvemento de software.

Neste proxecto empregaremos repositorios Git para os distintos compoñentes, xa que é unha ferramenta coa que o programador está familiarizado, e pola posibilidade de manter repositorios públicos na web GitHub de forma gratuíta.

Capítulo 5

Deseño

Neste capítulo falaremos sobre o deseño dos distintos compoñentes do sistema. Para iso, vaise dividir en tres seccións:

- Nunha primeira sección, analizaremos o modelo de datos, é dicir, as distintas clases que o sistema emprega para manter o estado dos distintos mundos de xogo.
- A segunda sección irá destinada a todo o relativo á linguaxe COE e o código ofrecido por ANTLR, e analizarase especialmente a implementación do patrón Visitor, crucial para a análise semántica da linguaxe.
- Finalmente, a terceira sección estará dedicada a DemiurgoWeb, a forma de funcionar de Spring en xeral e o deseño da web.

5.1. Modelo de datos

O modelo dentro do patrón modelo-vista-controlador é a parte do código que fai referencia aos distintos datos que o programa manexa. No caso do Demiurgo podemos atopalo nas clases contidas en dous paquetes: *Universe* e *Values*.

En Universe podemos atopar todas as clases relacionadas cos distintos conceptos manexados nun mundo de xogo: **Action, ClassMethod, DemiurgoClass, DemiurgoLocation, DemiurgoObject, DemiurgoRoom, Inventory, RoomPath, RootObjectClass, User, Witness e World**. Máis adiante explicarase en detalle o funcionamento de cada un deles.

En Values están todas as clases que fan referencia a distintos valores que pode conter unha variable no xogo. Todas estas clases implementan unha interface chamada **ValueInterface**, que é a empregada no patrón Visitor como tipo a devolver por cada visita (explicarase isto na subsección 5.2.3). Estas clases son: **FloatValue, IntegerValue, ListValue, StringValue, LocationValue, RoomValue, InventoryValue, ObjectValue**.

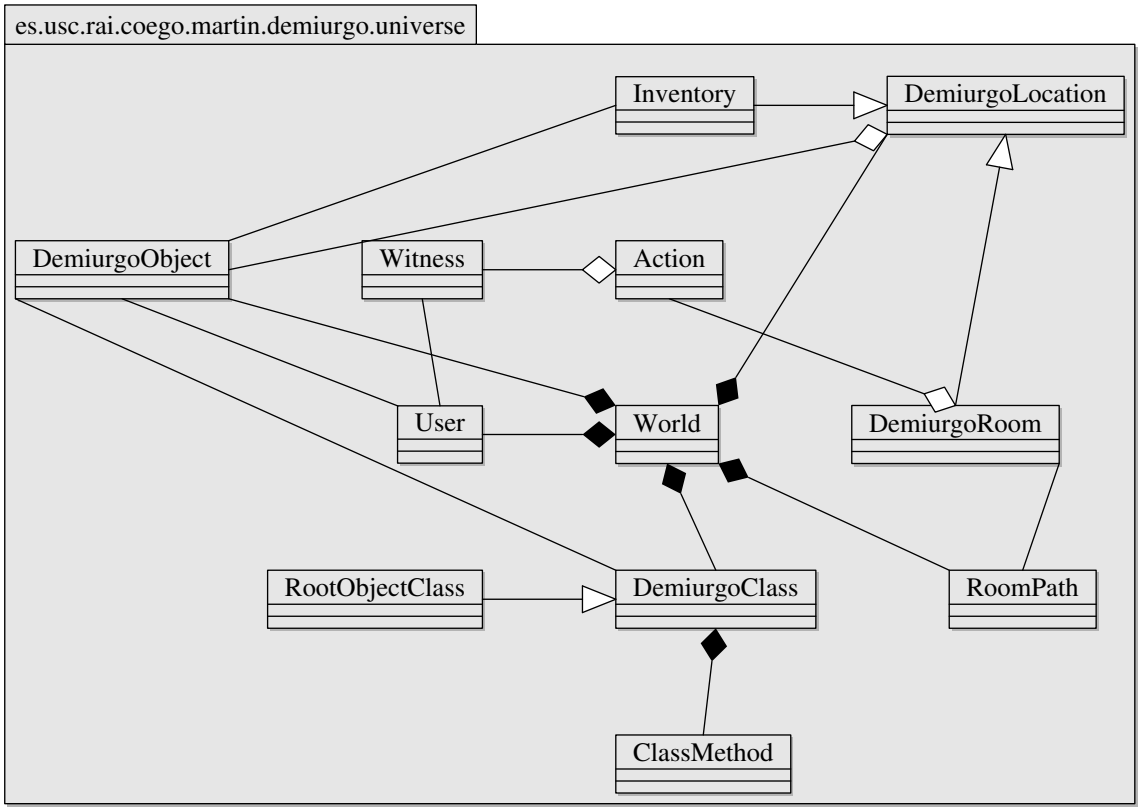

Figura 5.1: Diagrama de clases do paquete Universe (relacións entre clases).

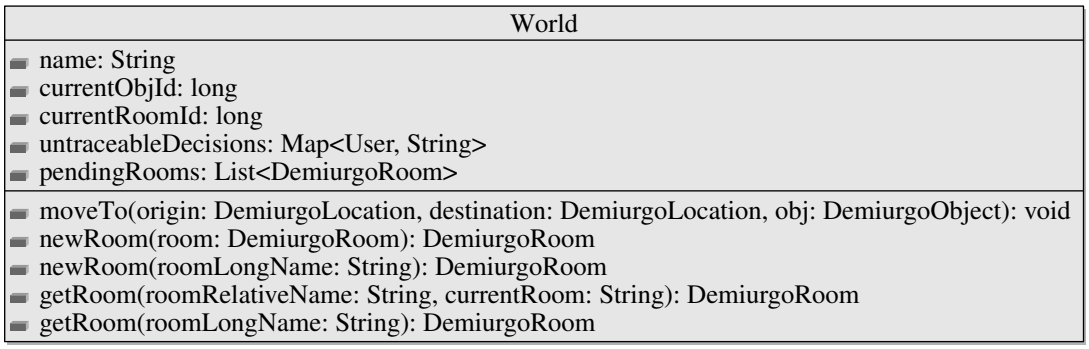

Figura 5.2: Clase World.

5.1.1. World

A clase que representa un mundo de xogo concreto. Existe un obxecto de clase World por cada partida que aloxe o Demiurgo. Esta clase contén as referencias a todos os obxectos, clases e escenarios do mundo, así como unha lista dos usuarios na partida. Todos estes elementos están debidamente referenciados en *maps* diferenciados, identificados polo seu identificador correspondente (un número na maioría dos casos, un nome nos usuarios).

Ademais de manter referencias, o obxecto World ofrece métodos útiles para xestionar as relacións entre os distintos compoñentes do mundo, como un método *setObjectUser* para asignar usuarios a obxectos.

RoomPath

RoomPath é unha clase que ten como finalidade organizar os escenarios do xogo segundo o seu nome ou ruta. A clase RoomPath emprega unha variante do patrón **Composite**: cada RoomPath contén unha lista de fillos, e un ou ningún escenario asociado (ver figura 5.3). Deste modo, almacenando un obxecto raíz equivalente ao escenario /, despregando os fillos deste obxecto atoparemos a árbore de escenarios. O funcionamento por tanto é moi semellante ao dunha árbore de directorios.

5.1.2. DemiurgoObject

A clase DemiurgoObject emprégase para os distintos obxectos do xogo, entendendo por *obxecto* a definición de xogo (non un obxecto de Java). Todos os obxectos conteñen un identificador (un número de tipo *long*) e unha lista de campos de tipo ValueInterface indexada por nome. A maiores, a clase DemiurgoObject ten referencias á *clase* (entendida segundo a definición de xogo) do obxecto en cuestión, a súa localización (un escenario ou un inventario) e o usuario relacionado con el en caso de ter algún.

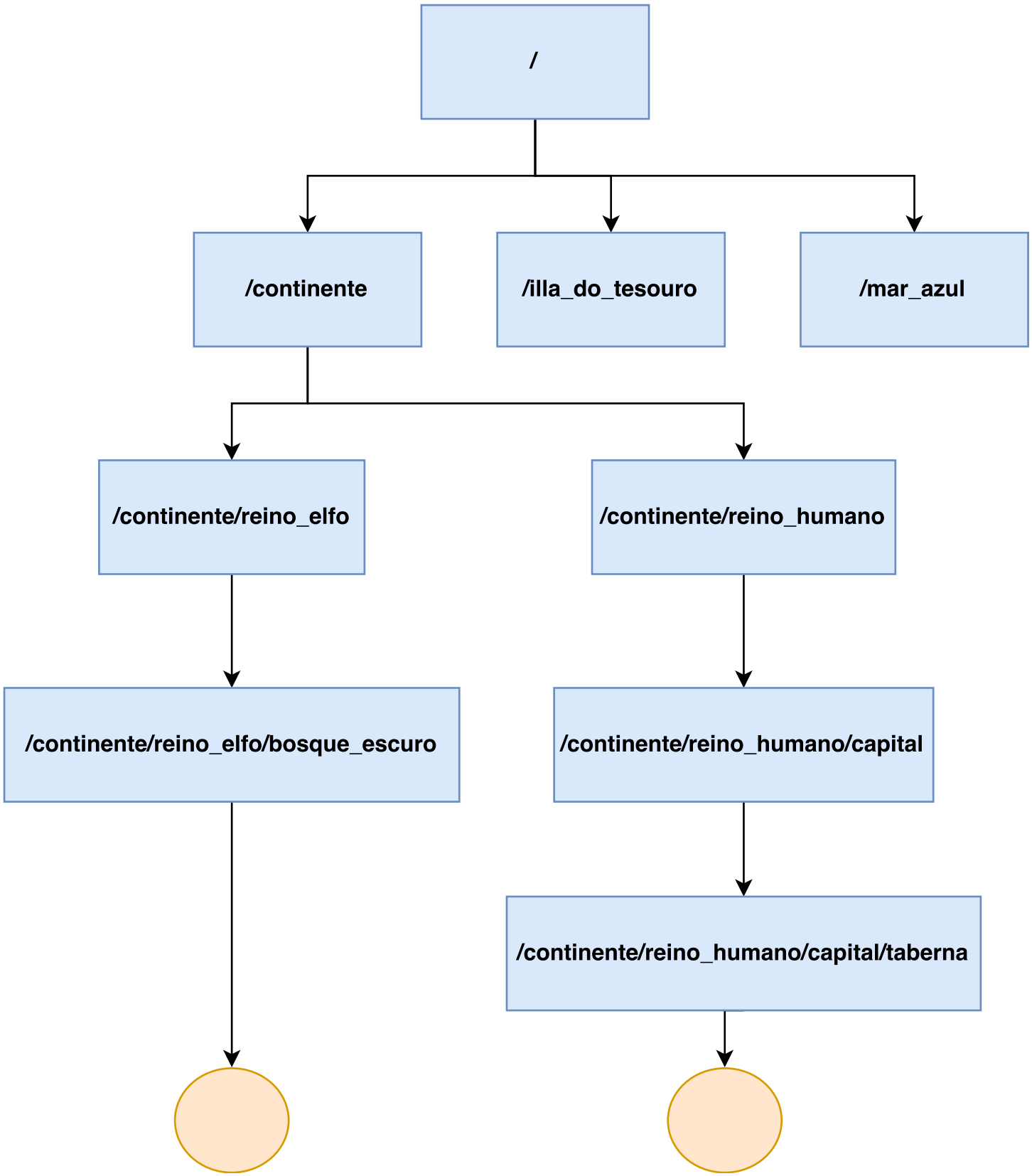

Figura 5.3: Exemplo de estrutura de escenarios. As caixas azuis representan obxectos RoomPath, e os círculos laranxas obxectos DemiurgoRoom.

5.1.3. DemiurgoClass

A clase DemiurgoClass fai referencia ás clases do xogo (no sentido da súa definición de xogo). Cada clase contén un nome, unha lista de campos (indexada por nome) e unha referencia á clase pai. Tamén conteñen unha lista de métodos.

ClassMethod

Os métodos de clase son fragmentos de código asociados a unha clase concreta cun nome determinado. Todo método dispón dunha lista de 0 ou máis argumentos (que son obxectos tipo ValueInterface), dos cales 0 ou máis son de saída e 0 ou 1 de entrada. Ademais disto, contén un obxecto de tipo NodeTree (propio da libraría ANTLR) que contén o código do método.

A clase ClassMethod é filla de DemiurgoMethod, unha clase abstracta que fai referencia a calquera tipo de método que poida ser executado no Demiurgo (tanto métodos de clases como funcións internas).

RootObjectClass

RootObjectClass é unha clase especial que herda de DemiurgoClass. Fai referencia á clase *Object* do mundo de xogo, é dicir, a clase raíz da que herdan todas as clases.

5.1.4. DemiurgoLocation

DemiurgoLocation é unha clase abstracta que fai referencia a calquera localización capaz de conter obxectos. As dúas clases fillas que herdan desta son por tanto DemiurgoRoom (escenarios) e Inventory (inventarios). Teñen en común a posesión da lista de obxectos que "conteñen".

As localizacións dispoñen de dous métodos útiles para o uso de inventarios: un é *isInsideOf(DemiurgoLocation)*, que devolve *false* no caso dos escenarios, pero nos inventarios devolve *true* se o obxecto que o contén se atopa dentro da localización especificada. O outro método é *getRealLocation()*, moi relacionado co anterior; se é invocado por un escenario devolverá o propio obxecto, pero se é invocado por un inventario, chamará ao método getRealLocation() da localización do obxecto que o contén.

5.1.5. DemiurgoRoom

DemiurgoRoom é a clase que representa un escenario no mundo. Herda de DemiurgoLocation e contén o seu nome de escenario completo (ou *ruta*), unha lista de variables de escenario (de tipo ValueInterface) e a *"prenarración"* do escenario (é dicir, o texto obtido mediante *echos* antes de redactar a narración definitiva; máis información no manual no apéndice B).

5.1.6. Inventory

O inventario é unha localización especial vinculada a un campo dun obxecto concreto. Os inventarios créanse automaticamente cando se crean obxectos que conteñen campos de tipo "inventario", e destrúense cando o obxecto é destruído (ou cando perde dito campo debido a unha modificación posterior da clase). Herda de DemiurgoLocation e a maiores ten unha referencia ao obxecto que o contén, así como o nome do campo asociado.

5.1.7. User

A clase User contén a información dun usuario: nome, enderezo email, decisión actual (unha cadea de texto ou *null*), e o seu rol. O seu contrasinal non se almacena no programa, polo que será necesario acceder á base de datos para realizar a operación de *login*.

UserRole

UserRole é unha enumeración que acepta dous valores: *GM* e *USER*. GM é o rol dos directores de xogo, mentres que USER é o rol dos xogadores comúns.

5.1.8. Action

Action contén toda a información necesaria dunha acción no mundo de xogo. Ten un identificador numérico (de tipo *long*), está asociada a un escenario concreto, e garda a narración da acción, a data e un booleano indicando se a acción está publicada ou non.

Witness

Cada acción garda unha lista de *testemuñas*, que basicamente son os usuarios que se atopaban no escenario no momento da acción. Cada testemuña contén o usuario en cuestión, e a *decisión* do usuario nesa acción.

5.1.9. Valores

A maiores do paquete Universe, tamén convén resaltar o paquete Values, que contén clases que fan referencia aos diferentes tipos de datos que manexa Demiurgo. Cada clase deste paquete ten definidas as diferentes operacións que pode realizar o seu tipo correspondente. Todas estas clases implementan a interface ValueInterface e herdan da clase abstracta AbstractValue, que ofrece operacións por defecto (polo xeral devolver unha excepción de tipo *IllegalOperationException*). Os valores teñen asignado un valor da enumeración *ReturnValueTypes* que identifica o tipo de valor.

Valores primitivos

As clases IntegerValue, FloatValue e StringValue representan os valores primitivos de números enteiros, flotantes e cadeas de texto. O seu contido está asociado ao obxecto ValueInterface correspondente, polo que ao usalo como argumento pásase por valor (non por referencia).

ListValue

A clase ListValue representa listas de outros elementos. Ademais de gardar o seu tipo ("LIST"), almacena un tipo diferente chamado *innerType*, que fai referencia ao tipo de elementos que contén. Tamén garda a profundidade da lista (*depth*). Estes dous valores son necesarios para definir totalmente o tipo dunha lista: non se pode asignar unha lista a unha variable lista con tipos internos diferentes ou con profundidades distinta.

ObjectValue

Contén unha referencia a un obxecto concreto. Dous ou máis obxectos ObjectValue poden facer referencia ao mesmo obxecto.

RoomValue

Clase filla de LocationValue, contén unha referencia a un escenario.

InventoryValue

Clase filla de LocationValue, contén unha referencia a un inventario. A diferencia de RoomValue, non pode xerarse de forma normal no código, nin enviarse como argumento; tampouco pode asignárselle un valor distinto ao que xa posúe. Isto é para evitar que os campos de inventario dun obxecto poidan ser reasignados e se perda a referencia orixinal ao inventario orixinal.

5.2. Deseño da linguaxe COE

Como xa se explicou en capítulos anteriores, o intérprete da linguaxe COE foi desenvolvido empregando o software ANTLR, un programa deseñado para ler gramáticas e xerar analizadores léxico-sintácticos a partir delas. Por tanto, o primeiro paso necesario para a elaboración deste intérprete foi unha definición formal da gramática de forma que ANTLR puidera entendela, mediante un ficheiro .g4.

5.2.1. Gramática

O ficheiro coa gramática pode atoparse no código fonte, na ubicación /src/-main/antlr/COE.g4 (seguindo a estrutura propia de Maven). O ficheiro está composto por unha serie de regras, que como podemos ver no manual de ANTLR[10] teñen a seguinte forma:

```
nomeDaRegra : <<alternativa1>> | ... | <<alternativaN>> ;
```

As alternativas, á súa vez, poden tratarse de outras regras ou de tokens léxicos (tales como números ou cadeas de texto concretas), os cales están definidos ao final do ficheiro; distínguense polo feito de ter as regras o nome en maiúsculas. Deste modo, a partir da gramática, o analizador léxico-sintáctico xerado por ANTLR pode analizar calquera código introducido e devolver unha árbore sintáctica segundo as regras definidas, ou devolver unha excepción se o código non ten equivalencia posible; implicando este caso que o código é incorrecto.

No caso da gramática de COE, a regra *"s"* é a regra inicial pola que comeza o proceso de análise. O primeiro que se pode observar é que ofrece dúas alternativas (se non temos en conta o texto baleiro): definición de clase, ou código. O primeiro caso implica definir unha nova clase para o mundo, mentres que o segundo correspóndese coa execución de código nun escenario concreto. Estas dúas regras por tanto correspóndense con dous contextos distintos do programa; no caso de empregalos incorrectamente (por exemplo, introducindo código corrente no canto dunha definición de clase) o sistema devolverá un erro; non obstante, iso definirase na análise semántica, e resulta irrelevante á hora de crear a árbore sintáctica.

No caso da definición de clase, a regra determina unha sintaxe concreta que permite definir nome da clase, clase pai, os distintos campos e os métodos; cabe destacar que a regra que define un método concreto contén a regra do código corrente, polo se un input de código determindo é aceptado nunha acción nun escenario, tamén será aceptado nun método dunha clase polo analizador sintáctico.

No caso do código dunha acción, este está composto por regras chamadas *"line"*, é dicir, liñas de código. Cada unha destas liñas pode ser:

- Unha operación,
- Unha expresión condicional "if",
- Un bucle "for",
- Unha asignación de obxecto a un usuario,
- Unha declaración dunha variable,
- Un *echo* (output de texto).

Á súa vez, as operacións poden conter outras operacións, o que permite combinalas tal e como indica o manual (apéndice B).

5.2.2. Proceso de análise

ANTLR xera código na linguaxe que especifiquemos (que neste caso será Java) que permite facer a análise léxica e sintáctica do código, polo que aforramos ter que implementar ese compoñente pola nosa parte. O que nos ofrece ANTLR son clases que permiten enviar como argumento un InputStream co código a analizar e devolven unha árbore sintáctica.

Cando executamos ANTLR coa gramática COE, devolve os seguintes ficheiros:

- COE.tokens: Non é unha clase Java. Recopila todos os tokens definidos na gramática e asigna un identificador único a cada un deles.
- COELexer: Analizador léxico. Le o código como texto e "esnaquízao" en tokens para que o analizador sintáctico poida lelos.
- COEParser: Analizador sintáctico. Analiza o fluxo de tokens producido polo analizador léxico e organízaos nunha árbore sintáctica segundo definan as regras da linguaxe.

Unha vez estamos neste punto, o seguinte será percorrer a árbore sintáctica e executar realmente o código, é dicir, interpretar o significado das distintas regras e realizar as mudanzas correspondentes no mundo de xogo. Este paso coñécese como **análise semántica**, e ANTLR ofrece dúas formas de realizalo:

- Listener: Por defecto ANTLR xera unha clase que implementa o patrón *Listener*, que é quen de percorrer unha árbore sintáctica e producir eventos cada vez que entra ou sae dunha norma. É unha forma sinxela de realizar a análise semántica, pero non nos ofrece o control do fluxo que precisamos para unha linguaxe como a linguaxe COE, polo que non o empregaremos.
- Visitor: É posible especificar que ANTLR xere unha clase que implemente o patrón *Visitor* para percorrer a árbore. Esta é a solución que escollimos neste proxecto, e será explicado na subsección 5.2.3.

5.2.3. Patrón Visitor

Para evitar a pesadez de ter que implementar manualmente unha clase que percorra a árbore sintáctica e a interprete debidamente, ANTLR ofrécenos a posibilidade de definir unha clase que implemente o patrón Visitor a partir dunha interface que xera xunto cos analizadores léxico e sintáctico. O patrón explícase brevemente no manual de ANTLR [10]; pódese ver un diagrama na figura 5.4.

O funcionamento resumido pódese ver no diagrama da figura 5.5: ANTLR xera unha interface chamada COEVisitor, a cal debemos implementar para xerar o noso propio Visitor (CodeVisitor neste caso). A interface danos un método "visit" por cada regra que atope na gramática. Neste caso, por simplificar, teremos

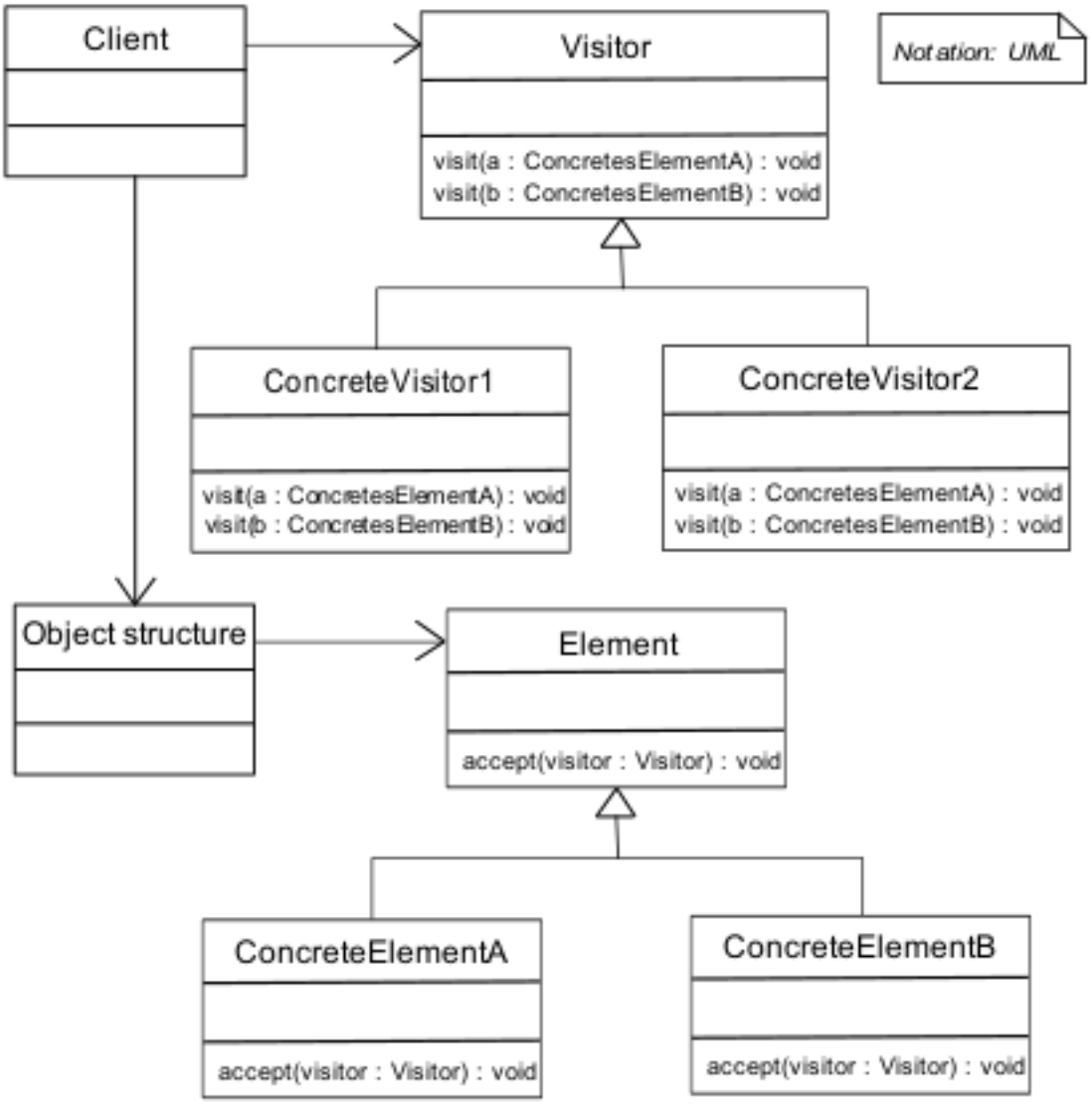

Figura 5.4: Diagrama de clases xenéricodo patrón Visitor.

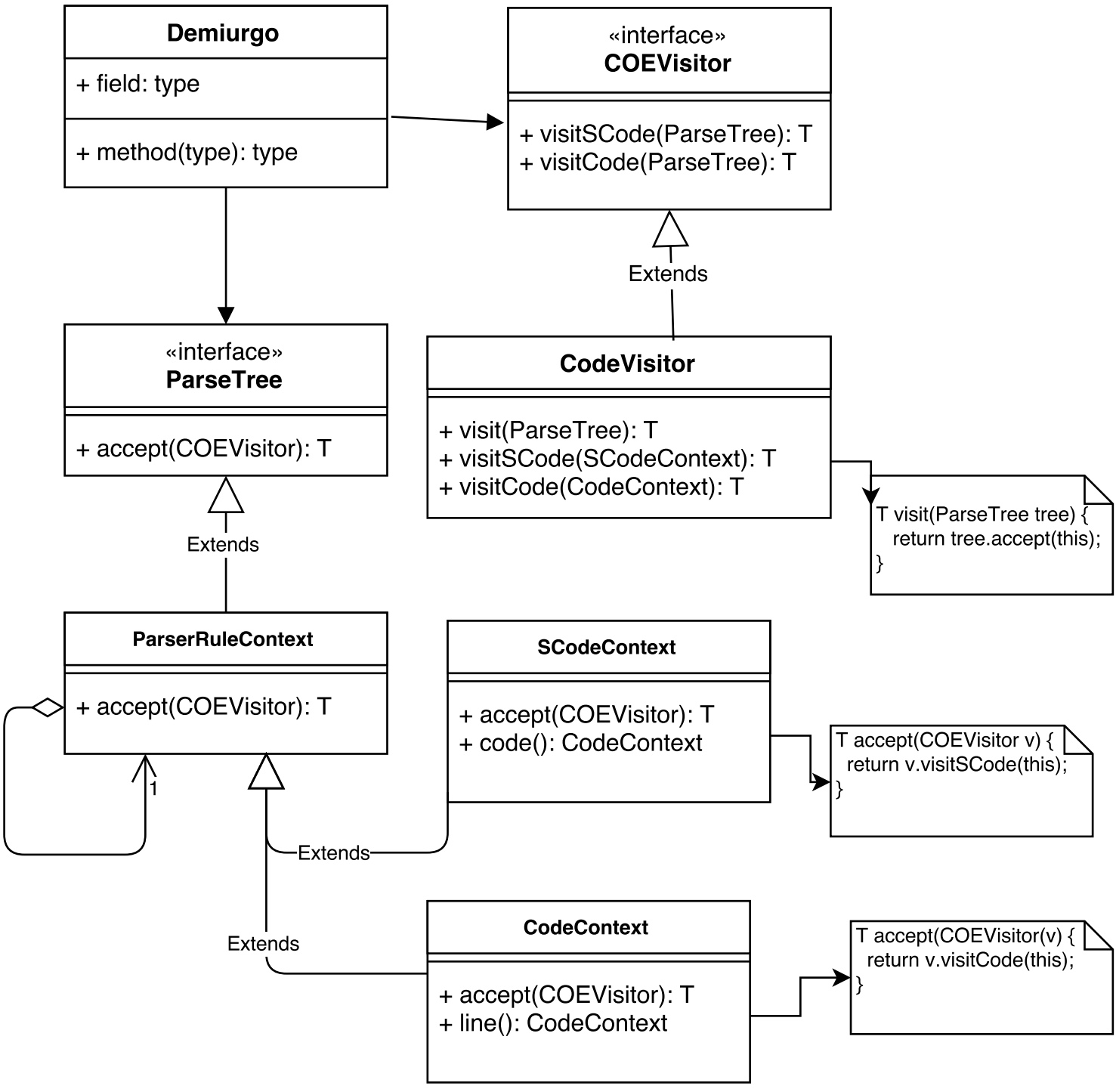

Figura 5.5: Diagrama de clases do patrón Visitor aplicado no proxecto.

en conta só dúas entradas: Code, relativo á regra chamada *code*, e SCode, relativo á segunda alternativa da regra chamada *s* (isto é, a que seguiría no caso de atopar código), tal e como o definimos na gramática:

```
s : nl? class_def nl?   #classDef
  | nl? code nl?        #sCode
  |                     #empty
  ;
```

Por tanto, a interface ofrécenos un método chamado visitSCode(), e outro chamado visitCode(); ambos reciben un ParseTree como argumento, unha clase que usa ANTLR para referirse a calquera elemento da árbore sintáctica (incluída a raíz).

Cabe destacar que na práctica non implementamos directamente a interface COEVisitor, senón que ANTLR nos ofrece a maiores unha clase abstracta chamada COEBaseVisitor da que podemos herdar que aporta métodos útiles; entre eles, o propio método visit() que no diagrama aparece definido en COEVisitor. Este tipo de detalles eliminároнse do diagrama para simplificar o proceso.

Partimos da base de que a execución do analizador léxico e sintáctico nun código determindo devolveunos unha árbore sintáctica; isto é, un obxecto que implementa a interface ParseTree e que contén fillos segundo estea definido nas regras da gramática. Na realidade este nodo é un obxecto da clase ParserRuleContext, tal e como mostra o diagrama. Entón, o proceso a seguir é chamar ao método *visit()* do novo Visitor personalizado, enviando a nosa árbore sintáctica como argumento. Este executará o método visit() por defecto, que non fai outra cousa máis que invocar o método *accept()* do nodo raíz.

No nodo raíz, que é un obxecto de tipo SCodeContext (xa que é regra inicial da gramática), execútase o método accept(). Tal e como describe o diagrama, este método invoca o método *visitSCode()* do Visitor, que estará definido por nós e por tanto fará o que nós especifiquemos. A maiores, o obxecto SCodeContext envíase como argumento e, tal e como o seu nome indica, servirá de contexto dentro do método visitSCode(): poderemos empregar os métodos que nos ofrece para obter os seus fillos. Neste caso, só dispoñemos do método *code()*, que nos devolve a súa regra filla chamada "code". Polo que nalgún momento na definición de visitSCode(), será preciso facer visit(ctx.code()) para seguir co proceso.

Por tanto, para aplicar o patrón Visitor en ANTLR, o único que precisamos é definir os distintos visitX() que precisemos a partir da interface no noso Visitor personalizado.

CodeVisitor e ClassVisitor

Debido a que hai dous contextos diferenciados na execución de código no Demiurgo (código de acción e definición de clases), implementáronse tres clases coa interface COEVisitor:

- ExecVisitor: Clase abstracta. Contén código común a ambos contextos.
- CodeVisitor: Filla de ExecVisitor. Contén código exclusivo do contexto da execución de código de acción. Almacena unha referencia ao escenario actual, e devolve unha excepción se recoñece código de definición de clases.
- ClassVisitor: Filla de ExecVisitor. Contén código específico para definir clases, e garda unha referencia á clase que está sendo definida. Se recoñece código de acción, devolve unha excepción.

Scopes e ScopeManager

Nun momento dado da execución, o *scope* ou ámbito determina os símbolos aos que ten acceso o analizador. Xunto cos Visitors implementouse un xestor de scopes (*ScopeManager*) que contén o scope que se está empregando en cada momento, así como unha pila de scopes na que se van engadindo os scopes novos. Podemos distinguir os seguintes (todos son clases que herdan da clase abstracta *Scope*):

- WorldScope: Tamén chamado Scope global. Contén unha referencia ao mundo de xogo.
- RoomScope: O scope inicial do código de accións. Contén unha referencia ao escenario actual. Cando se lle solicitan símbolos, devolve variables do escenario.
- ClassScope: O scope inicial da definición de clases. Contén unha referencia á clase en definición. Cando se lle solicitan símbolos, devolve os campos da clase.
- ObjectScope: Este scope só se emprega en solitario cando se xeran os valores por defecto dun obxecto. Aparte diso, emprégase como pai de FunctionScope ao invocar o método dun obxecto. Contén unha referencia ao obxecto en cuestión, e as referencias a símbolos devolven campos do obxecto.
- MethodDefiningScope: Scope que é utilizado ao definir os métodos dunha clase. Contén os argumentos do método (que se extraerán unha vez remate a definición para formar o método en si), e ten unha referencia a un ClassScope como pai para acceder aos campos da clase en definición.
- LoopScope: Scope empregado nun bucle ou nun condicional. Contén unha lista de variables locais que desaparecen unha vez o bucle ou condicional rematan. Tamén ten unha referencia ao scope anterior (pai).
- FunctionScope: Clase filla de LoopScope empregada para os bucles for. Contén unha lista de valores a empregar para facer o bucle, e o nome da variable que empregará para almacenalos iterativamente.

- ForScope

Xestión de erros

ANTLR por defecto ofrece unha estratexia de xestión de erros que busca evitar que a análise se deteña; é dicir, en caso de atopar un erro léxico ou sintáctico, ANTLR intenta recuperarse para proseguir coa análise.

Este sistema non nos serve para o proxecto, xa que a nosa pretensión é que o sistema se deteña e mostre un erro en canto atope un erro no código, mostrando a posición do mesmo. Debido a isto, foi necesario configurar tanto o analizador léxico como o semántico para que empreguen a estratexia BailErrorStrategy, definida por ANTLR, que detén a análise ante calquera erro.

A maiores, foi preciso buscar o modo de deter a análise semántica en canto se detecte un problema (por exemplo, unha operación ilegal). Para isto optouse por encapsular as distintas excepcións que o noso Visitor pode lanzar dentro de RuntimeExceptions, para así deter o percorrido do Visitor e poder recuperar as causas do erro.

5.3. DemiurgoWeb

Capítulo 6

Implementación e probas

Capítulo 7

Conclusións e posibles ampliacións

Conclusións e posibles ampliacións

Apéndice A

Manuais técnicos

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

Apéndice B

Manual de usuario

B.1. Introdución

A linguaxe COE (acrónimo de Código de Obxectos e Escenarios) é a linguaxe empregada nas partidas no Demiurgo, o software de xestión de partidas de rol asíncronas. O director de xogo (*DX*) pode escribir código en COE para definir clases no mundo de xogo, e para realizar accións nos escenarios nas que os distintos obxectos interactúan entre si.

A finalidade deste manual é definir esta linguaxe e ensinar a darlle uso para poder empregar todas as funcionalidades do Demiurgo sen dificultade, ofrecendo para isto definicións formais, explicacións e exemplos en cada apartado. Grazas a este sistema, o DX pode manter un mundo loxicamente coherente que lle faga de soporte á hora de levar as partidas, evitando así erros e automatizando en gran medida o apartado técnico das mesmas.

B.2. Escenarios, clases e obxectos

Un mundo de xogo de Demiurgo está composto por obxectos, cada un dos cales segue a definición dunha clase e atópase dentro dun escenario. A medida que a partida avanza, os distintos obxectos créanse, modifícanse, móvense dun escenario a outro e interactúan entre eles.

B.2.1. Exemplo dunha situación

Expresaremos unha situación normal do xogo mediante este exemplo:

A situación transcorre na taberna da capital do reino. Tres ananos están nun recuncho discutindo entre eles acaloradamente. Detrás da barra pódese ver ao taberneiro, un home corpulento que se distrae limpando un vaso. Detrás da barra ten un mosquetón agochado por se as cousas se poñen feas, e por se acaso non lle quita ollo aos ananos.

En base a esta descrición da situación, o director de xogo pode definir internamente o seguinte:

- Un escenario que se corresponde coa **taberna**. O escenario debería ter un nome recoñecible para o director de xogo, como por exemplo */reino/capital/taberna*, máis iso explicarase máis en detalle en B.2.2.

- Como mínimo **5 obxectos**:

  - Tres obxectos que representen aos **ananos**. Xa que son tres obxectos moi semellantes, o normal é que pertenzan á mesma clase; por exemplo, unha clase chamada *Anano*. Máis isto non é unha norma, dependerá de como o director de xogo teña definido o mundo.
  - Un obxecto que represente ao **taberneiro**. Pode pertencer a unha clase diferenciada da dos ananos (como *Humano*); non obstante, ben poden pertencer tanto o taberneiro como os ananos a unha mesma clase *Intelixente* ou *Criatura*, por exemplo.
  - Un obxecto que faga referencia á **barra**, xa que o **mosquetón** que contén pode resultar relevante no futuro. Por tanto, o normal sería que houbese un obxecto que representase á barra (de clase *Mobiliario* por exemplo), e no seu interior se atopase un inventario que conteña ao obxecto mosquetón (de clase *Mosquetón*). Explicarase con detalle o concepto dos inventarios en B.2.4.

Esta configuración é un simple exemplo que non ten por que cumprirse ao 100 % nunha partida real, pero da unha idea do concepto. Poñamos agora outro exemplo: na mesma situación, comezan a suceder cousas.

Parece que a discusión dos ananos chega a un punto crítico. Dous deles deciden simultaneamente que non poderán resolver as súas diferencias mediante a diplomacia, polo que deciden sacar os puños e golpearse mutuamente. O taberneiro, que non lles quitaba ollo, saca o mosquetón como resposta por se comeza unha escalada de violencia.

Isto pode suceder por dous motivos: os dous ananos son personaxes controlados por xogadores e ambos tomaron a mesma decisión (golpear ao outro) ao mesmo tempo; ou ben son personaxes non xogadores (*NPC*, *non player character*) e foi o propio DX o que tomou a decisión por eles (para que outros xogadores no escenario presenciaran isto, por exemplo).

En calquera caso, o que lle corresponde ao DX é executar o código necesario para cambiar o estado do mundo. O código a redactar debe facer tres cousas:

1. Un dos ananos debe **golpear** ao outro, cos cambios que iso implique (por exemplo, reducir o seu campo *vida*).

2. O segundo anano debe **golpear** ao primeiro.

3. O mosquetón debe deixar de estar no inventario da barra e debe **moverse** ao inventario do taberneiro.

Este sería un exemplo de como podería ser este código:

```
anano1.golpear(anano2)
anano2.golpear(anano1)
barra.contidos[0] >> taberneiro.inventario
```

Como entendemos este código? Temos unha variable anano1 que fai referencia ao obxecto *Anano* que da o primeiro golpe, unha variable anano2 que fai referencia ao segundo, unha variable *barra*, unha variable *taberneiro*, e un método *golpear()* definido para a clase *Anano*. Este código enténdese así:

1. O obxecto correspondente á variable anano1 executa o **método** *golpear()*, collendo o obxecto da variable anano2 como argumento. Este método, definido na clase *Anano*, reduce a vida do anano seleccionado como argumento. Explicarase máis en detalle isto na sección B.2.3.

2. O mesmo que no anterior, pero ao revés.

3. O primeiro obxecto (*[0]*) dos contidos da barra ("contidos" é un inventario da clase *Mobiliario*), que neste caso entendemos que é o mosquetón, **móvese** (») ao inventario do taberneiro.

B.2.2. Escenarios

Definición de escenario

O mundo de xogo está dividido en escenarios (tamén chamados *habitacións* polo seu homólogo *rooms* en inglés). Cada escenario está composto por:

- O seu nome, sobre o cal falaremos na subsección B.2.2.

- O conxunto de obxectos que se atopan "dentro" do escenario.

- As variables do escenario, as cales poden conter datos tales como números enteiros ou texto, pero tamén referencias a obxectos ou outros escenarios.

O escenario forma unha unidade lóxica fundamental no mundo de xogo, e o habitual é que as accións en cada escenario afecten só aos obxectos do propio escenario (aínda que pode haber excepcións).

Nome do escenario

O nome de escenario serve para que o DX o identifique rapidamente. Ademais de ofrecer un nome descritivo, o nome completo permite organizar os escenarios nunha estrutura xerárquica, cunha lóxica semellante á das rutas de ficheiros nos sistemas operativos.

Exemplo:

```
/continente/reinohumano/cidadecapital/armeria
/continente/reinohumano/cidadecapital/taberna
/continente/reinohumano/aldeacosteira
```

Estes tres escenarios atópanse dentro de /continente/reinohumano, é dicir, na estrutura lóxica do mundo de xogo están dentro do mesmo reino, o cal forma parte do continente. A maiores, os dous primeiros están na mesma cidade. Estes compoñentes normalmente reciben o nome de *rexións*, pero poden ter o significado semántico que o DX de xogo prefira, por exemplo, nunha partida futurista poderíamos atopar algo como:

```
/planetamarte/coloniahumana/suburbios/casa3/cocinha
/planetamarte/orbita/sateliteflotante/laboratorio
/espazoexterior/zonadeasteroides
```

Por último, o DX pode renunciar completamente a usar a xerarquía de escenarios, aínda que iso pode dificultar o nomeado de escenarios:

```
/chairas_da_capital_humana
/sala_do_gran_mago
```

É importante ter en conta que o nome dos escenarios só pode estar composto por caracteres alfanuméricos e barras baixas, e non pode conter letras que non formen parte do alfabeto inglés (tales como *ñ* ou *ç*). O nome completo sempre debe comezar por /. Por último, o Demiurgo non distingue entre minúsculas e maiúsculas.

Nota: / en si mesmo pode ser un escenario, o coñecido como *escenario raíz*, pero non é habitual o seu uso.

Contido do escenario

O escenario pode conter obxectos (e de feito é habitual que os conteña). Estes obxectos á súa vez poden dispoñer de inventarios (ver subsección B.2.4) que conteñan outros obxectos. Mediante a interface gráfica, o DX pode ver os distintos obxectos do escenario para operar con eles no código.

Variables

O DX pode definir variables no escenario a través do código. Estas variables están accesibles dende o propio escenario e poden ter todo tipo de utilidades. Por

exemplo, é habitual referenciar os obxectos do escenario en variables para poder manexalos co código máis facilidade.

Exemplo: Imaxinemos que no escenario temos un obxecto que representa a un trasno. O obxecto ten un identificador propio (*ID*) que é #1234 (ver sección B.2.4). No momento de querer actuar, se non empregamos variables precisamos referirnos a el co seu ID:

```
#1234.facerTrasnadas()
if(#1234.famento == true)
  ! "O trasno ten bastante fame"
```

En cambio, se temos unha variable cun nome descritivo (por exemplo, *trasno*) podemos codificalo así:

```
trasno.facerTrasnadas()
if(trasno.famento == true)
  ! "O trasno ten bastante fame"
```

Outro uso frecuente das variables é gardar datos. Neste exemplo tírase un dado de 20 caras e móstrase un texto ou outro segundo o resultado:

```
int resultado = d20
if(resultado > 10) {
  ! "A accion tivo exito"
}
if(resultado < 3) {
  ! "A accion foi un fracaso absoluto"
}
```

B.2.3. Clases

As clases son definicións empregadas a modo de modelo para crear obxectos. Todos os obxectos que teñan a mesma clase terán unha serie de métodos e campos comúns definidos na propia clase.

Nesta sección explicarase como funcionan as clases e todo o relacionado con elas. No capítulo B.4 explícase polo miúdo a sintaxe formal para crear clases.

Campos

Os campos son variables vinculadas aos obxectos, e a súa finalidade é almacenar datos relacionados con eles. Por exemplo, nun mundo de rol de fantasía medieval, é habitual atopar nos personaxes campos como forza, axilidade ou intelixencia. Nun mundo con máquinas de guerra, normalmente atoparemos campos como a potencia de disparo ou o blindaxe. E tamén podemos atopar campos en obxectos inanimados, como o peso, o prezo, a cor, etcéra.

Os campos defínense coa clase, e cando é creado un obxecto desa clase, automaticamente pasa a posuír todos os campos da mesma.

Exemplo:

```
elfo {
  int forza = 1
  int axilidade = 3
  int constitucion = 1
  int intelixencia = 2
  int vida
}
```

No exemplo vemos a definición dunha clase chamada *Elfo*. Como vemos, os elfos teñen os campos forza, axilidade, constitución, intelixencia e vida, os cales son números enteiros *int* e teñen asignados uns valores por defecto (excepto vida). Cando se crean obxectos da clase *Elfo* automaticamente xorden con eses campos, pero o seu valor pode ser modificado posteriormente.

Métodos

Cada clase pode ter definidos unha serie de métodos. Os métodos son fragmentos de código relacionados coa clase que poden ser invocados en calquera momento do xogo, útiles para simplificar procesos repetitivos.

Por exemplo:

```
elfo {
  bendicir(elfo outro) {
    outro.vida = outro.vida + 5
    ! "O elfo foi bendecido"
  }
}
```

No exemplo, a clase *Elfo* dispón dun método *bendicir()* que recibe como argumento un obxecto *Elfo* (outro elfo ou el mesmo!) e aumenta a súa vida en 5. Deste modo, o DX non precisa escribir ese código cada vez que queira facer que un elfo bendiga a outro, e pode invocalo do seguinte xeito:

```
elfo1.bendicir(elfo2)
```

Argumentos Vimos no exemplo do método *bendicir()* que o código fai referencia a unha variable chamada *outro*. Isto é porque o ámbito dun método está reducido ao obxecto que o invoca e aos argumentos que recibe; isto é, non pode acceder ás variables do escenario.

Cada método pode ter definidos un número indeterminado de argumentos de entrada, e un argumento de saída (ou ningún). Todos estes argumentos deben especificar o tipo, sexa un tipo primitivo como número enteiro ou texto, ou a clase

se se trata dun obxecto. Se a chamada do método se realiza con argumentos incorrectos (por exemplo, enviando un obxecto dunha clase incorrecta, ou enviando un texto cando require un número) o Demiurgo devolverá un erro.

Dixemos que no ámbito dos métodos tamén está o obxecto que o invoca. Isto faise mediante a variable *this*:

```
elfo {
  ceder_vida(elfo outro) {
    outro.vida = outro.vida + 5
    this.vida = this.vida - 5
  }
}
```

Os argumentos sempre teñen prioridade sobre os campos do obxecto en caso de que os nomes coincidan; debido a isto, o uso de *this* é necesario para referirse aos campos do obxecto cando isto suceda.

Construtor Por último, hai un método especial moi importante chamado **construtor**, que se executa automaticamente no momento de crear o obxecto. É útil por tanto para inicializar clases. O construtor recoñécese polo feito de chamarse igual ca clase. No caso de ter argumentos, é necesario especificalos na creación do obxecto:

```
elfo {
  int idade
  str nome
  elfo(int novaidade, str novonome) {
    this.idade = novaidade
    this.nome = novonome
  }
}
```

E no momento de crealo:

```
elfo legolas = new elfo(100, "Legolas o elfo")
```

Herdanza de clases

Un aspecto fundamental das clases é a herdanza das mesmas. Unha clase pode especificarse como herdeira ou filla doutra clase, e deste modo herdará todos os campos e métodos da clase pai. Esta característica, propia das linguaxes orientadas a obxectos, permite aforrar moito traballo:

```
animal {
  int altura
  str cor
```

```
  int peso
  str ruido

  facerRuido() {
  ! "este animal fai " + ruido
  }
}

can : animal {
  str tipoPelo
  str raza

  can() {
    ruido = "guau"
  }
}
```

Outra utilidade da herdanza é que podemos gardar un obxecto da clase filla nunha variable da clase pai. Por exemplo:

```
personaxe {
  int vida = 10
}

guerreiro : personaxe {
  int ataque = 5
  int vida = 20
  atacar(personaxe outro) {
    outro.vida = outro.vida - ataque
  }
}
```

Neste exemplo, o método *atacar()* pode recibir un personaxe como argumento, ou calquera obxecto dunha clase filla (como outro guerreiro).

Clase Object Realmente todas as clases herdan dalgunha outra clase. Se a clase pai non é definida, esta será automaticamente a clase *Object*, unha clase auxiliar que non conta con campos. É posible crear obxectos da clase Object directamente, pero realmente non aportan demasiado ao xogo xa que non conteñen valor semántico ningún.

B.2.4. Obxectos

Os obxectos son o elemento central do xogo. Cada obxecto representa a unha entidade no xogo, que pode ser física (como un cervo ou unha mesa) ou un

concepto (como un estado alterado ou unha idea). Todos os obxectos teñen as seguintes características:

- Identifícanse individualmente por un número enteiro coñecido como ID.
- Correspóndense cunha clase determinada.
- Atópanse nunha localización determinada, que pode ser un escenario, ou o inventario de outro obxecto.

Os obxectos dispoñen dos campos definidos pola súa clase, e poden chamar métodos da mesma. Estes aspectos explicáronse na sección "Clases" na sección B.2.3.

Inventarios

Os obxectos poden dispoñer duns campos especiais que non existen no caso das variables de escenario: os inventarios. Un inventario é unha localización vinculada a un determinado obxecto. Os inventarios, como o resto de campos, defínense na definición da clase, pero a diferencia do resto de campos non poden ser reasignados: o obxecto nace cos seus inventarios e só desaparecerán cando desapareza o obxecto.

Exemplo:

```
heroe {
  % inventario
  % estados

  personaxe() {
    new espada() >> inventario
  }
}
```

No exemplo podemos ver como a clase *Heroe* conta con dous inventarios: un deles chámase "inventario" directamente, e o outro "estados". No construtor da clase (ver B.2.3) especifícase que no momento de crearse o obxecto, crearase tamén un obxecto de clase *Espada* e gardarase no inventario.

Os inventarios, a pesar de ter este nome, poden ter o valor semántico que desexe o DX: poden conter os tesouros que atopara o personaxe, conter unha lista de estados alterados, conter as ideas dun NPC, etcétera.

B.3. Accións e usuarios

A acción do xogo desenvólvese mediante accións, que é o que comunica aos usuarios co mundo. Os usuarios escriben o que queren facer cos seus personaxes,

e é labor do DX executar o código pertinente e mostrarlles unha narración por escrito que describa o sucedido.

B.3.1. Decisións

Un xogador nun momento dado pode escribir o que desexa que o seu personaxe faga. Isto no Demiurgo coñécese como *decisión*. Este texto non ten ningún tipo de efecto no sistema, simplemente é mostrado ao DX cando abre o escenario no que se atopa o obxecto do xogador.

B.3.2. Accións

Por acción enténdese todo o sucedido entre dúas narracións, é dicir, todo o código executado e as decisións correspondentes que o provocaron. O DX pode executar todo o código que precise, e facer varias execucións diferenciadas, antes de engadir unha narración e concluír por tanto a acción.

Testemuñas

As testemuñas son os xogadores que se atopan no escenario no momento de rematar a acción. Estas testemuñas recóllense xusto antes de executar o último código (o código que iniciou o proceso de narración), polo que se incluirán entre elas os personaxes que cambiasen de escenario no último momento.

B.3.3. Narracións

A narración é o resultado textual da acción. O DX redáctao baseándose no estado do escenario e nos *echos* (ver sección B.4.2) producidos. Os xogadores que sexan testemuñas poderán ver este resultado textual.

Fragmentos ocultos

O DX poderá marcar fragmentos da narración como ocultos empregando as etiquetas [o][/o]. Por exemplo:

```
Este texto e visible para todos.
[o=gamer]Este texto so e visible para o xogador 'gamer'.[/o]
[o=gamer user]
Este texto e visible para 'gamer' e 'user'.
[/o]
```

B.4. Definición da linguaxe COE

No capítulo anterior abordouse o funcionamento dos escenarios, as clases e os obxectos. Neste capítulo mostrarase unha definición formal da linguaxe, tanto na definición de clases como no código das accións.

Nas descricións sintácticas é preciso ter en conta as seguintes marcas:

- **<elemento >**: Os elementos marcados deste xeito son elementos obrigatorios, pero non deben ser entendidos como código literal, xa que o contido é unha descrición do elemento. Exemplo: <argumento >.
- [**elemento**]: Os elementos marcados deste xeito son opcionais.
- **elemento ...** : Cando aparecen os puntos suspensivos, o último elemento antes deles pódese repetir varias veces. Exemplo: <arg>[<arg>] ... .

B.4.1. Tipos de datos

No Demiurgo e na linguaxe COE podemos atopar os seguintes tipos de datos:

- **Número enteiro:** Representa un número sen decimais. Útil para facer cálculos, e para realizar tiradas de dados. Exemplo: *27*. Tamén representa valores lóxicos, sendo 0 o equivalente a dicir "falso" e calquera outro valor (normalmente 1) o equivalente a dicir "verdadeiro". O seu tipo denótase coa palabra reservada *int*.
- **Número con decimais:** Tamén coñecido como número con punto flotante ou simplemente flotante, representa un número non enteiro. Pode empregarse cando se empreguen campos que precisen decimais. Exemplo: *3.14*. Debe terse en conta que ao converter números con decimais a números enteiros, perderanse os decimais (non se redondea). O seu tipo denótase coa palabra reservada *float*.
- **Cadea de texto:** Calquera fragmento de texto. Útil para mostrar mensaxes aos xogadores, ou para gardar anotacións do DX. Exemplo: *"este é un texto de exemplo"*. O seu tipo denótase coa palabra reservada *str*.
- **Escenario:** As variables e campos de escenario non conteñen o escenario como tal, senón unha referencia ao mesmo. Pode ser útil almacenar referencias a escenarios concretos para automatizar o movemento de obxectos entre eles. Aparte diso, non serven para facer operacións. Para obter unha referencia a un escenario concreto, emprégase o carácter @. Exemplo: *@/rexion1/cidade1/taberna*. No caso de non comezar por /, enténdese que é un nome parcial; por exemplo se o código se executa en /rexion1/cidade1/taberna, e se chama a @casa3, o resultado é o mesmo que

chamando a @/rexion1/cidade1/casa3. Tamén é posible chamar ao propio escenario co punto (@.), a un nivel superior cos dous puntos (@..) ou calquera combinación. O seu tipo denótase co carácter *@*.

- **Inventario:** Exclusivo dos campos de obxecto, fan referencia ao inventario pertinente. Funcionan de maneira moi similar aos escenarios, salvo polo feito de que non se poden enviar como argumento nin sobreescribir o seu valor con outra referencia. O seu tipo denótase co carácter %.

- **Obxecto:** Referencia a un obxecto concreto. Permite interactuar co obxecto dende outro escenario, ou gardalo nunha variable cun nome descritivo. É posible obter unha referencia a un obxecto coñecendo a súa ID. Exemplo: *#1024*. O seu tipo denótase co nome da clase correspondente.

- **Lista:** Lista de datos de calquera dos outros tipos. Exemplo: {*1, 2, 3, 4*}. Tamén se poden crear listas baleiras: {}. O seu tipo denótase co tipo de elementos que contén e uns corchetes, por exemplo int[]. Tamén se poden especificar listas de listas: int[][].

B.4.2. Echo

Echo é o termo que recibe o feito de emitir un resultado textual. No Demiurgo, pódese facer echo de calquera valor que poida ser lido como unha cadea de texto (tamén números enteiros e flotantes, e listas). Facer echo no código ten unha utilidade: cando o DX estea redactando a narración da acción, poderá ver todos os echos, polo que é útil para ver información antes de facer a narración completa.

O echo faise co carácter reservado **!**.

```
! <valor_a_mostrar>
```

Exemplo:

```
! "este e un texto de proba"
```

B.4.3. Operacións

No Demiurgo atopamos as seguintes operacións:

- **+ Sumar:** Suma dous valores. Se os dous elementos son enteiros, o resultado é un enteiro; se un é flotante e o outro enteiro ou flotante, o resultado é un flotante. Se un dos elementos é unha cadea de texto, a operación no canto de ser unha suma de números convírtese nunha concatenación de texto: "proba de " + "texto" devolvería "proba de texto", e "valor de " + 4 devolvería "valor de 4". En calquera outro caso, emite un erro.

- \- **Restar:** Resta dous valores. Semellante a suma, salvo polo feito de que non serve para concatenar texto. Pódese usar cun só operando para obter o seu negativo.
- ! **Operador lóxico "non":** Operador unario, é dicir, que acepta un operando. Devolve 1 se o dato é falso, ou 0 se o dato é verdadeiro. Un número enteiro ou flotante é verdadeiro se é distinto de 0. Unha lista devolverá unha lista de 1s 0s; o resto de tipos devolverá sempre un valor falso.
- == **Igual:** Compara dous datos. Se os datos son loxicamente iguais, devolve un 1. En caso contrario, devolve un 0. Dúas referencias son iguais se referencian ao mesmo obxecto ou escenario.
- != **Non igual:** Devolve 0 se os datos son loxicamente iguais, 1 en caso contrario.
- >**Maior que:** Devolve 1 se o primeiro valor é maior co segundo, 0 en caso contrario. Funciona con enteiros e flotantes.
- <**Menor que:** Devolve 1 se o primeiro valor é menor co segundo, 0 en caso contrario.
- >= **Maior ou igual que:** Devolve 1 se o primeiro valor é maior ou igual co segundo, 0 en caso contrario.
- <= **Menor ou igual que:** Devolve 1 se o primeiro valor é menor ou igual co segundo, 0 en caso contrario.
- & **Operador lóxico "e":** Compara dous elementos lóxicos. Se os dous son verdadeiros, devolve 1, en caso contrario, devolve 0.
- |**Operador lóxico "ou":** Compara dous elementos lóxicos. Se polo menos un dos dous é verdadeiro, devolve 1, en caso contrario, devolve 0.
- = **Asignar:** Asigna o valor da dereita á variable ou campo da esquerda. Ademais, se se usa en conxunto con outras operacións, devolve o valor asignado. En caso de non ser compatibles o valor e a variable, emite un erro.
- >>**Mover:** Move o obxecto da esquerda á localización da dereita. Tamén se poden mover listas de obxectos: en tal caso, móvense todos os obxectos da lista.
- ++ **Concatenar:** Concatena dúas listas. Deben ser do mesmo tipo. Se un dos dous operandos non é unha lista pero ten o mesmo tipo que o tipo interno da lista, engádese ao comezo ou ao final (dependendo da orde dos operandos).

- D **Tirada de dados:** Pódese usar como operador unario ou binario. Se se usa cun só operando, fai unha tirada dun dado onde o número de caras é o número especificado polo operando (que debe ser un enteiro) e devolve o resultado. Se se usa con dous operandos, fai X tiradas de Y caras, onde X é o valor do primeiro operando e Y o do segundo, e devolve unha lista de resultados.

A maiores, é posible recoller operacións entre parénteses para darlles prioridade.

A prioridade de operadores é a seguinte:

Tiradas de dados >parénteses >multiplicar ou dividir >sumar ou restar >comparacións >operacións lóxicas >asignacións >movementos

Débese ter en conta que as asignacións sempre collerán unha variable do lado esquerdo, polo que teñen a máxima prioridade por este lado.

Exemplos de operacións:

```
(1+2*(3-4))                 // Devolve -1
3d20 > 10                   // Devolve unha lista de 3 valores 0 ou 1
#42 >> @/zona1/estancia3    // Move o obxecto de ID 42 ao escenario
```

Operacións con listas

As listas permiten realizar practicamente todas as operacións dos elementos que conteñen, pero teñen unha característica: o resultado cambia segundo a operación sexa con outra lista ou cun elemento dun tipo distinto.

Operacións entre elementos e listas Se se realiza unha operación entre un dato que non sexa lista e unha lista, o Demiurgo o que devolverá será unha lista de resultados entre o dato e cada elemento da lista.

Exemplo:

```
3*{1,2,3}
```

Este exemplo devolvería a lista {3,6,9}.

Isto funciona tamén con listas de listas:

```
5+{{5,5},{10,10}}
```

Este exemplo devolve a lista {{10,10},{15,15}}.

Operacións entre listas As operacións entre listas simplemente devolven unha lista cos resultados elemento a elemento. Se o tamaño das listas non coincide, devolverá un erro.

Exemplo:

```
{2,3,6}+{1,2,5}
```

Este exemplo devolve a lista {3,5,11}.

Tamén valen combinacións entre os dous tipos de operación:

```
{2,4}*{{10,10},{100,100}}
```

Devolvería a lista {{20,20},{400,400}}.

B.4.4. Variables

As variables de escenario créanse especificando o seu tipo e o nome da variable. No momento de definir unha variable nova, é posible asignarlle un valor na mesma liña:

```
<tipo> <nome_variable> [ =<valor> ]
```

Exemplo de variables:

```
int a                // define o enteiro 'a'
str b = "proba"      // define a cadea de texto 'b' co valor "proba"
elfo c = new elfo()  // define a referencia 'c' e garda un novo Elfo
```

B.4.5. Condicionais

É posible marcar fragmentos do código para que sexan executados ou non en base a unha condición. Se a condición é verdadeira, o código execútase; en caso contrario, non se executa nada. Tamén é posible especificar un código alternativo para este último caso.

Definición formal:

```
if (<condicion>) {
  <codigo_condicion_verdadeira>
}
[else {
  <codigo_condicion_falsa>
}]
```

Se o código só é unha liña, pódese prescindir das chaves.

Exemplo de condicional:

```
if(d6 > 3) {  // tira un dado de 6 e o compara co enteiro 3
  ! "exito"
}
else {
  "fracaso"
}
```

B.4.6. Bucles

É posible especificar bucles no código, é dicir, fragmentos de código que se executarán varias veces. O Demiurgo conta co bucle **for**, que permite executar un fragmento de código unha vez por cada elemento dentro dunha lista.

Definición formal:

```
for (<variable> : <lista>) {
  <codigo_for>
}
```

Por cada elemento da lista, farase unha execución do código asignando tal elemento á variable especificada. Se no canto dunha lista se lle pasa unha cadea de texto, contará como unha lista de caracteres.

Se o código só é unha liña, pódese prescindir das chaves.

Exemplo de bucle:

```
for(i : 5d10) {
  ! "resultado: " + i
}
```

B.4.7. Funcións integradas

É posible empregar funcións predefinidas no propio Demiurgo para realizar determinadas operacións complexas. Estas funcións chámanse do seguinte xeito:

```
[<saida>] = <nome_funcion> ( <arg> [, <arg> ] ... )
```

As funcións dispoñibles son as seguintes:

- count(*lista*): Recibe unha lista e devolve o número de elementos. Se o argumento é unha cadea de texto, devolve o número de caracteres.
- seq(*inicio, fin*): Recibe dous números enteiros e devolve unha lista formada pola secuencia de enteiros entre ambos números, cun paso de 1. O número de inicio debe ser menor ou igual co de fin.
- reverse(*lista*): Recibe unha lista e devolve a lista do revés, é dicir, co primeiro elemento en última posición e así sucesivamente. Se o argumento é unha lista de listas, as listas do interior non son modificadas, só a súa orde.
- sub(*lista, inicio, fin*): Recibe unha lista e dous números enteiros e devolve unha sublista composta polos elementos entre a posición de inicio (inclusive) e a posición de fin (exclusive).
- sum(*lista*): Recibe unha lista de enteiros e devolve a suma dos seus elementos.

- zeros(*total*): Recibe un número enteiro e devolve unha lista composta por tantos ceros como especifique o argumento.

- destroy(*obxecto*): Recibe unha referencia de obxecto, e destrúe ese obxecto. Desaparecerá do mundo de xogo, e con el os seus inventarios e os obxectos que estes conteñan. Esta función non devolve ningún valor.

Exemplo:

```
sum( seq(1,10) )            // suma os enteiros do 1 ao 10
```

B.4.8. Obxectos e métodos

Creación de obxectos

Os obxectos créanse coa palabra reservada "new" o nome da clase, e se é preciso os argumentos do método construtor. A definición é esta:

```
new <nome_clase> ( [ <arg> [, <arg> ] ... ] )
```

Ademais de crear o obxecto, este método devolve unha referencia ao mesmo, que pode ser gardada nunha variable mediante unha asignación.

Os obxectos no momento de ser creados aparecen no escenario no que se estea escribindo o código.

Exemplo de creación de obxecto:

```
elfo legolas = new elfo(100, "Legolas o elfo")
```

Invocación de métodos

Para invocar un método cun obxecto é preciso ter a referencia dese obxecto, extraer o seu método co punto (**.**) e engadir os argumentos en caso de ser necesarios. Se o método ten argumento de saída, pódese gardar o seu valor nunha variable, ou empregarse directamente para facer operacións.

```
<referencia_obxecto>.<nome_metodo> ( [ <arg> [, <arg> ] ... ])
```

Exemplo:

```
legolas.bendicir(outroelfo)
```

B.4.9. Definición de clases

No momento de crear a clase, especifícanse os atributos e métodos que terán os obxectos creados a partir dela. A definición dunha clase é a seguinte:

```
<nome_clase> {
  <atributos>
  <metodos>
}
```

Os atributos defínense como as variables de escenario. Os métodos defínense da seguinte forma:

```
[<tipo> <arg_saida> =]
  <nome_metodo>([ <tipo> <arg>[, <tipo> <arg>]...])
{
  <codigo_metodo>
}
```

Se no código do método se asigna un valor ao argumento de saída, este devolverase cando o método sexa chamado.

Exemplo de clase:

```
anano : criatura {
  int forza = 3
  int axilidade = 1
  int intelixencia = 1
  int vida = 30
  str nome

  anano(str nome) {
    this.nome = nome
  }

  int dano = golpear(criatura outro) {
    dano = sum((forza)d6)
    criatura.vida = criatura.vida - dano
  }
}
```

B.4.10. Contidos das localizacións

Pódese obter unha lista dos obxectos que se atopan nun escenario ou nun inventario empregando o carácter reservado % do seguinte xeito:

```
<localizacion>.%
```

Por exemplo:

```
personaxe.inventario.% >> @.
```

Este exemplo colle unha lista de obxectos no inventario do personaxe e móveos todos ao escenario actual.

B.4.11. Usuarios

Os usuarios non poden ser manexados como valores por ningún tipo de dato; pero hai unha operación especial (**->**) que permite asignarlle un obxecto a un usuario coñecendo o seu nome.

$<nome_usuario> -> <referencia_obxecto>

Exemplo:

$gamer -> #30

Apéndice C

Licenza

Se se quere pór unha licenza (GNU GPL, Creative Commons, etc), o texto da licenza vai aquí.

Bibliografía

[1] Páxina web do MUD. *http://www.british-legends.com/CMS/*. Consultado o 13 de xaneiro do 2017.

[2] Role-playing game. Artigo da wikipedia (*http://en.wikipedia.org*). Consultado o 24 de outubro do 2016.

[3] Roll20, xogo de rol. Páxina web: *https://roll20.net*

[4] Páxina web de Roleplay Workshop. *http://www.roleplay-workshop.com/*. Consultado o 13 de xaneiro do 2017.

[5] LambdaMOO Programmer's Manual. Versión 1.8.0p6, 1997. Dispoñible en *http://www.hayseed.net/MOO/manuals/ProgrammersManual.html*.

[6] J. Patrick Williams; Sean Q. Hendricks; W. Keith Winkler, *Gaming as Culture: Essays on Reality, Identity and Experience in Fantasy Games*, 2006.

[7] *The Scrum Guide*. Xullo 2006. Dispoñible en *http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf*.

[8] *Pro Git*, 2ª edición. 2014. Dispoñible en *https://git-scm.com/book/en/v2*.

[9] A.V. Aho, R. Sethi, J.D. Ullman. *Compiladores. Principios, tecnicas y herramientas*. 1a edición. Addison Wesley, 1990.

[10] Terence Parr, *The Definitive ANTLR 4 Reference*, The Pragmatic Bookshelf, 2013.

[11] Getting Started Guides de Spring Framework. Dispoñible en *https://spring.io/guides*. Consultado o 24 de outubro do 2016.