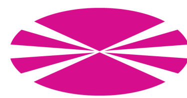

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Aplicación web altamente escalable para un xogo de dedución social

Estudante: Diego Valcárcel Barbeito
Dirección: Laura Milagros Castro Souto

A Coruña, xuño de 2021.

To the future

Agradecementos

Grazas a todas aquelas persoas que influencian a miña vida con esperanza e ánimos, por permitirme apreciar o día a día. Grazas ós meus amigos, parella, familia e a miña titora por todo.

Resumo

A interacción social a través da tecnoloxía está vivindo un grande aumento en usuarios activos, tanto por motivos laborais como de ocio. Porén, entretementos clásicos e "analóxicos" como os xogos de mesa poden atopar o seu lugar neste contexto e acomodar as novas persoas usuarias de xeito máis accesible. Neste proxecto impleméntase unha versión do xogo de dedución social Mafia nunha aplicación web.

Mafia é un xogo de dedución social onde os participantes están divididos en dous grupos: o campesiñado e a mafia. A mafia é un subconxunto reducido con coñecemento e habilidades especiais, que tenta eliminar ao campesiñado, desinformado malia constituír a maioría. O xogo transcorre por quendas consecutivas e repetitivas (día e noite), durante as que se realizan un certo número de accións (por exemplo, xulgar a alguén sospeitoso polo día ou eliminar inocentes pola noite).

O produto final deste traballo é unha aplicación desenvolvida en Elixir, que implementa a lóxica do xogo, e unha aplicación web feita usando o marco de desenvolvemento web da linguaxe, Phoenix. O sistema resultante adapta as regras de Mafia e facilita a creación de partidas para xogar cun grupo de coñecidos, de forma sinxela e sen obstáculos. O seu deseño está orientado a conseguir unha alta escalabilidade e tolerancia a fallos. Ademais, faise un uso intensivo de probas baseadas en propiedades para conseguir unha validación da lóxica máis exhaustiva da que se podería obter cos métodos de probas máis tradicionais.

Abstract

Technology-enabled social interaction is growing in active users, for reasons both professional and personal. We argue that classic, "analog" entertainment elements such as board games can find their space in this new context and welcome new users in a more accessible way. In this project, we implement a web version of the social deduction game Mafia.

Mafia is a social deduction game where players are divided into two groups: the town and the mafia. The mafia is a small subset with wide knowledge and special abilities, trying to wipe out the uninformed, larger crowd of the town. The game is a sequence of repeating play phases (day and night), during which and a set of different actions can be carried out (such as, putting a player on trial during the day, or removing them during the night).

The end product of this project is an Elixir application that implements the game logic and a web application developed using the Phoenix framework. The resulting product adapts the rules of Mafia and enables a group of players to easily create and run games. System design is focused on high availability and fault tolerance. In addition, we leverage the power of

property-based testing to prove the correctness of the game logic to an extent not achievable by more traditional testing methods.

Palabras chave:

- Elixir
- Phoenix Framework
- Aplicación web
- Probas baseadas en propiedades
- Mafia

Keywords:

- Elixir
- Phoenix Framework
- Web application
- Property-Based Testing
- Mafia

Índice Xeral

1 Introdución **1**

1.1 Motivación . . . 1

1.2 Obxectivos . . . 2

1.3 Metodoloxía e planificación . . . 3

2 Contexto **5**

2.1 Mafia . . . 5

2.1.1 Xogos de de deducción social . . . 5

2.1.2 Regras . . . 6

2.1.3 Regras das quendas . . . 6

2.1.4 Variantes . . . 7

2.2 Elixir . . . 8

2.2.1 Erlang . . . 9

2.2.2 Escalabilidade . . . 9

2.2.3 *Behaviours* (modelos de comportamento) . . . 9

2.2.4 Tolerancia a fallos . . . 10

2.2.5 Relevancia . . . 10

2.3 Framework Phoenix . . . 11

2.3.1 Channels . . . 11

2.3.2 LiveView . . . 11

2.4 Probas baseadas en propiedades . . . 11

3 Deseño **13**

3.1 Requisitos . . . 13

3.1.1 A nosa variante de Mafia . . . 13

3.1.2 Requisitos funcionais . . . 14

3.1.3 Requisitos non funcionais . . . 16

3.2 Arquitectura: modelo C4 . . . 17

3.2.1 Diagrama de Contexto do sistema 17
3.2.2 Diagrama de Contedor 17
3.2.3 Diagramas de Compoñente 17
3.2.4 Código 18
3.3 Deseño da Interface 21

4 Implementación **25**
4.1 Aplicacións en Elixir 25
4.2 MafiaEngine 26
4.3 MafiaInterface 33

5 Probas **37**
5.1 Probas de unidade 37
5.2 Probas de integración 38
5.3 Probas baseadas en propiedades 40
5.3.1 Definindo propiedades con máquinas de estados 41
5.3.2 Exemplos de comandos 43
5.3.3 Resultados 46

6 Conclusións **47**
6.1 Obxectivos 47
6.2 Leccións aprendidas 48
6.3 Traballo Futuro 49

Bibliografía **51**

Índice de Figuras

3.1 Fases da nosa versión de mafia 14
3.2 Diagrama de contexto do sistema 17
3.3 Diagrama de contedor do Xogo de Mafia 19
3.4 Diagrama de compoñente do Mafia Engine 19
3.5 Diagrama de compoñente do Mafia Interface 20
3.6 Diagrama de estado do xogo 22
3.7 Deseño da interface 23
3.8 Deseño de axustes dentro do lobby 24

Índice de Táboas

1.1 Tempo e custo total do proxecto e dividido por tarefa 4

2.1 Distribución de roles en base ao total destes nunha partida 6

Capítulo 1

Introdución

O primeiro capítulo da memoria tratará as motivacións polas cales se fixo este proxecto e os obxectivos a cumprir neste, ademais da metodoloxía que se empregou para a realización do traballo.

1.1 Motivación

As tecnoloxías de videochamadas están a acoller un aumento moi grande nos usuarios destas. A mensaxería directa deixou de lado no seu día ás chamadas telefónicas como nova forma de comunicarse a distancia, e agora un proceso semellante está a ocorrer coas videochamadas. Aínda que o uso destas non chegaba ata a maioría de usuarios, coa chegada do distanciamento social isto cambiou, movendo a unha gran cantidade de usuarios a empezar a usar con regularidade as videochamadas para unha mellor comunicación. Desta forma as interaccións sociais a distancia ocupan un espazo máis grande hoxe en día.

A vía habitual do entretemento en liña son os videoxogos, pero estes teñen como xogador obxectivo a unha persoa habituada ós controis e linguaxe destes (o movemento do personaxe, o movemento de cámara, a sincronización de accións, etc). Para acomodar as novas persoas usuarias das máis recentes tecnoloxías con entretemento máis accesible, voltamos ós xogos de mesa. Estes son unha fonte de diversión e de fomento da sociabilidade entre os seus xogadores, historicamente xogados en reunións presenciais, mais non ten porque ser así hoxe en día. Aínda que constan de regras e materiais que son necesarios para poder ser xogados, co crecente uso da tecnoloxía en tódolos aspectos das nosas vidas, non é de estrañar que os xogos de mesa poidan ser desfrutados tamén en novos medios, máis tecnolóxicos.

Hoxe en día as aplicacións web están en auxe como unha forma sinxela de entregar un servizo ou produto, xa que ofrecen moitos beneficios frente ás aplicacións de escritorio. En primeiro lugar, abstraen os sistemas software que se desenvolven dos detalles concretos dos posibles clientes, grazas a depender só do navegador web. Deste xeito, elimínase a respon-

sabilidade do mantemento e o despregamento de actualizacións neses clientes. Ademais, ao trasladar a computación maioritariamente ao servidor, permite a redución dos requisitos necesarios por parte do cliente. Por último mais non menos importante, permite enfocar o desenvolvemento nun único proxecto no canto de dous (considerando que un cliente de escritorio ou cliente *pesado* pode asimilarse a un proxecto en si mesmo).

Neste proxecto implemetarase unha versión do xogo Mafia nunha aplicación web. En Mafia os participantes están divididos en dous grupos: campesiñado e mafia. O xogo consiste de dúas quendas consecutivas repetitivas (día e noite) e certo número de accións (por exemplo, votar a un xogador polo día). A mafia consta de habilidades adicionais (coñecerse entre si e matar pola noite) mentres que o campesiñado son unha maioría sen capacidades especiais. Cremos que unha aplicación web melloraría a experiencia de xogo permitindo ademais o xogo a distancia. Normalmente Mafia xógase cun xogador que "conduce" a partida, os cambios de fase e as regras das accións; pasar esta responsabilidade ó código elimina a necesidade de que unha persoa non participe e axiliza os cambios entre fases.

Este concepto, o de adaptar xogos de mesa a novos medios máis tecnolóxicos, non é algo novo posto que na actualidade existen xa unha ampla multitude de aplicacións que achegan esta posibilidade. Porén, estas versións de Mafia comparten un mesmo problema: a necesidade de rexistrarse para xogar, asumindo que o usuario necesita da propia aplicación para encontrar xente coa que xogar. A alternativa a este problema que se propón neste proxecto é facer unha aplicación web *plug and play*, na que os usuarios xa teñen un grupo co que xogar e non precisan de rexistrarse para desfrutar do servizo. A aplicación web sería a ferramenta coa cal xogar con máis facilidade: o que tradicionalmente sería a caixa do xogo de mesa coas instrucións e o taboleiro.

Para a realización deste proxecto, decidíuse empregar tecnoloxías nas que xa se tiña coñecemento previo: Elixir como linguaxe de programación, xa que facilita a creación de aplicacións concorrentes, escalables e tolerantes a erros, e Phoenix como marco de desenvolvemento para facer a aplicación web, posto que destaca pola súa simplicidade para construír aplicacións web dinámicas.

1.2 Obxectivos

O principal obxectivo que se ten neste proxecto é o de desenvolver unha aplicación web que adapte o popular xogo Mafia, cun enfoque na facilidade de uso. Para facer isto, empregaranse tódalas capacidades que achega o marco de desenvolvemento escollido, Elixir/Phoenix, cunha especial énfase en LiveView, un compoñente que permite *renderizar* HTML no servidor en base ó estado da conexión.

A maiores, esta aplicación cumprirá certos requisitos non funcionais para asegurar uns mínimos de calidade:

1. Ser escalable para permitir manexar múltiples partidas e varios centos de xogadores simultaneamente.
2. Contar con tolerancia a fallos para ser robusta ante fallos inesperados mantendo ó estado da partida.

Para cumprir estes dous obxectivos, utilizarase a librería estándar da linguaxe de programación que se usa no proxecto, Elixir, xa que as propias capacidades fundamentais desta tecnoloxía que escollemos como base nos axudarán a conseguilos.

1.3 Metodoloxía e planificación

O desenvolvemento deste proxecto fíxose seguindo unha metodoloxía áxil baseada na metodoloxía Scrum, aínda que con modificacións para adaptala ao feito de que o proxecto foi realizado por unha persoa e non por un equipo, que é no que se orienta esta metodoloxía. Como é usual nesta metodoloxía, o traballo realizouse en pequenos *sprints* (iteracións), nos que ían xurdindo ideas que se refinaban para converter en obxectivos ben definidos a cumprir, ademais de constar dunha duración de tempo fixa (2 semanas) e de sempre ter a finalidade de, ao rematar cada unha das iteracións, ter unha versión da aplicación que fose funcional. A maiores realizáronse reunións semanais coa directora do proxecto para discutir o traballo realizado, o traballo por realizar, e ideas ou cuestións sobre o proxecto.

O proxecto pódese dividir nunha serie de etapas, ou tarefas, que engloban cada unha parte do total de obxectivos que se propuxeron durante os *sprints*. Estas son:

1. Deseño da lóxica básica da aplicación.
2. Implementación da lóxica e das probas.
3. Deseño da interface da aplicación web.
4. Implementación da interface e das probas.

A maiores, unha vez levadas a cabo todas estas etapas, realizouse unha última fase iterativa na que se era necesario se deseñaba unha ampliación da aplicación, logo se implementaba xunto ás probas necesarias, cambios na interface, e probas na interface, e se volvía comprobar se era necesario ampliar máis.

Para calcular o custo total do proxecto tomouse como referencia o salario medio das persoas que exercen a Enxeñaría do Software en España [1], cifrado en 27€/hora para valorar

economicamente o esforzo do estudante. A maiores, é importante destacar que como a totalidade do proxecto se fixo con ferramentas de código aberto con licenzas libres (GitHub [2] como repositorio, Gigalixir [3] para despregar a aplicación), non hai custos adicionais no proxecto. Na táboa 1.1 móstrase o total de horas que levou finalizar cada unha das tarefas, xunto co custo destas e do total do proxecto. O proxecto tamén ten licenza libre, nomeadamente MIT e pode atoparse en `https://github.com/menxs/mafia_engine` e `https://github.com/menxs/mafia_interface`. Por outra parte o despregue en Gigalixir pode atoparse en `https://mafia-game.gigalixirapp.com/`.

Tarefa	**Esforzo (en horas)**	**Custo (€)**
Deseño da lóxica básica da aplicación	40	1080
Implementación da lóxica e das probas	85	2295
Deseño da interface da aplicación web	35	945
Implementación da interface e das probas	100	2700
Implementación de amplicación reiterativa	120	3240
Documentación da memoria	80	2160
Total	460	12420

Táboa 1.1: Tempo e custo total do proxecto e dividido por tarefa

Capítulo 2

Contexto

NESTE capítulo falaremos dos conceptos nos que se basea este proxecto necesarios para entender o resto da memoria, como poden ser as características diferenciadoras das tecnoloxías co cal este se levou a cabo ou a noción de que son os xogos de dedución social.

2.1 Mafia

Mafia[4], creado por Dimitry Davidoff en 1986, é un xogo no cal cada un dos xogadores asume un rol en segredo, xeralmente clasificado como mafia ou campesiñado. As persoas pertencentes á mafia son a minoría dos xogadores e coñecen a identidade dos seus aliados, mentres que o campesiñado é a maioría pero non coñece o verdadeiro rol dos demais xogadores. O xogo divídese en quendas de día e noite con diferentes accións posibles nas diferentes quendas. A condición de vitoria para cada bando é a eliminación do bando contrario, o que se pode conseguir a través de discusións abertas e votacións para eliminar a xogadores, ou no caso específico da mafia, "matando" membros do campesiñado durante as quendas de noite. O xogo resúmese en que unha maioría desinformada xoga en contra dunha minoría informada, e esta é a base coa que a mafia xera discusións, mentiras, sospeitas, alianzas e conversas entre o grupo de xogadores. O xogo é unha boa forma de socializar e coñecer a outras persoas.

Ademais, desde a súa premisa de ocultar o rol e obxectivo dos xogadores, *Mafia* pertence ós primeiros xogos de dedución social inventados.

2.1.1 Xogos de de deducción social

Os xogos de dedución social son aqueles onde os xogadores intentan destapar o verdadeiro rol, obxectivo ou intencións do resto dos xogadores. Un xogo con estas características crea dinámicas adicionais ás regras do xogo baseadas nas interaccións persoais dos xogadores. Tódalas accións de cada persoa do grupo pasan o escrutinio deste en busca de razóns pola cal

confiar ou desconfiar dela. A maiores, algúns xogadores poden optar por actuar coma un rol que non lles pertence para obter beneficios ao longo prazo.

2.1.2 Regras

Para xogar a Mafia é necesario un grupo de polo menos 7 persoas, onde unha delas terá a tarefa de moderar a partida. En base ao número total de xogadores decídese o número de persoas que haberá na mafia nesa partida, o cal usualmente é información pública (ver táboa 2.1).

Xogadores	Mafiosos
6-7	2
8-10	3
11-13	4
14-16	5

Táboa 2.1: Distribución de roles en base ao total destes nunha partida

Unha vez o moderador reparte os roles ós xogadores en segredo, a partida pode comezar. O xogo consta de dúas fases que se repiten en bucle ata que *ou ben* a mafia elimine a todo o campesiñado, *ou* este sexa capaz de desfacerse da mafia a través das votacións que se fan ao longo da partida.

2.1.3 Regras das quendas

As regras do xogo varían lixeiramente durante as fases que se suceden en bucle.

Día

Polo día tódolos xogadores poden comunicarse libremente, e a maiores calquera pode acusar publicamente a outro xogador. No caso de haber acusacións, o moderador anuncia o xuízo á persoa que máis acusacións obtivera, cedéndolle a palabra para que argumente a súa defensa. A continuación, os demais xogadores terán que votar a favor, en contra ou absterse, para eliminar a persoa acusada. Cunha maioría a favor a persoa queda eliminada da partida, e se non, a persoa acusada segue dentro da partida. Os xuízos polo día usualmente teñen limitacións, permitindo soamente unha eliminación ou certo número de xuízos “errados” (isto é, sen veredicto de eliminación). Se non hai acusacións ou non pode haber máis xuízos o moderador avanza o xogo á seguinte fase.

Noite

Durante a noite os xogadores "durmen", pechando os ollos e permanecendo en silencio. O moderador narra como a mafia esperta e, as persoas pertencentes a esta, abren os ollos, recoñécense entre si, e sinalando póñense de acordo para eliminar a un membro do campesiñado. A narración continúa describindo como a mafia volve durmir e amence na vila, todos os xogadores espertan, e o moderador termina a narración co anuncio da eliminación da persoa obxectivo da mafia, a cal deixa a partida. Se a partida non acaba coa eliminación (isto é, se aínda quedan campesiños), vólvese á fase do día.

2.1.4 Variantes

Debido a que o xogo se estendeu de persoa en persoa e a que non conta cunha "versión oficial", existen múltiples variantes nas regras dependendo do grupo no que se xogue. Exemplos de variacións poden ser como solucionar o caso do empate nos xuízos, como elixir entre dúas persoas a eliminar nun xuízo, ou como obrigar á eliminación dun xogador polo día. A extensión máis importante e popular é a de engadir roles adicionais con regras moi específicas para estes. Exemplos de posibles roles adicionais habituais inclúen:

- **O detective**: forma parte do campesiñado e pola noite esperta para investigar a un xogador e coñecer o seu rol.
- **O médico**: forma parte do campesiñado e pola noite esperta para curar a un xogador; no caso de que este sexa atacado pola mafia esa noite, impide a súa eliminación.
- **O padriño**: forma parte da mafia, mais no caso de ser investigado polo detective parecerá ser do campesiñado.
- **O bufón**: forma parte do campesiñado, mais ten unha condición de vitoria diferente á do resto de xogadores, xa que gaña en caso de ser eliminado por votación polo día, perdendo no caso contrario.

A sólida base de *Mafia* dunha minoría informada en contra dunha maioría desinformada utilízase en moitos xogos de mesa populares, aínda que con premisas e regras diferentes:

- ***Los hombres lobo de Castronegro*** **[5]** cambia a mafia por un grupo de homes lobo que atacan pola noite, agregando varios roles adicionais con múltiples habilidades e cartas para representar os roles.
- ***Bang!*** **[6]** xogo centrado no salvaxe oeste, no que un grupo de pistoleiros con identidades segredas e obxectivos ocultos se enfrontan nun tiroteo mentres que o *sheriff* intenta poñer orde.

- ***Secret Hitler*** [7], onde os liberais tentan impedir que os fascistas aproben políticas fascistas no goberno e que Hitler chegue á presidencia.

Existen tamén versións para xogar na rede, variantes deseñadas para xogar nun foro ou cun acercamento máis parecido a un videoxogo. Estes sitios contan con complexidade e configuración extensiva cun grande elenco de roles e habilidades.

- ***Mafia scum*** [8], a páxina web máis grande dedicado o xogo de Mafia no cal os usuarios se organizan a través do seu foro para xuntar un moderador con xogadores e xogar a través dun fío no foro.

- ***Mafia.gg*** [9], unha versión baseada en texto, cun chat de texto entre os xogadores na partida e regras automatizadas.

- ***Town of Salem*** [10], centrado nos xuízos de bruxas de Salem cunha interface gráfica similar á dun videoxogo, permite partidas de 7 a 15 xogadores.

Tamén cabe destacar **Among us** [11], un xogo de dedución social onde os xogadores son a tripulación dunha nave e deben terminar as tarefas desta antes de que os impostores maten á tripulación. Este xogo tivo unha explosión en popularidade en 2020 demostrando que existe interese por xogos cunha barreira de entrada sinxela que fomentan a sociabilidade entre os xogadores.

2.2 Elixir

Elixir [12] é unha linguaxe de programación dinámica e funcional, o que significa que non comproba o tipado das variables ata o tempo de execución por ser dinámico e que ten ás funcións como cidadás de primeira clase (ou sexa, que poden empregarse como variables, argumentos a funcións ou poden devolverse como resultado doutras funcións).

Elixir está baseada na BEAM, a máquina virtual de Erlang [13], a cal achega un modelo de concorrencia sinxelo e potente implementando o modelo actor, que na linguaxe ocorre como procesos especiais, illados entre si, que comparten información a través de mensaxes; ademais, grazas a que estes procesos son estremadamente lixeiros e á filosofía "*let it crash*", que aboga pola ausencia da programación defensiva e a súa combinación coa supervisión e reinicio de tarefas, aporta á linguaxe unha grande robustez e tolerancia a erros, ademáis dunha alta escalabilidade. É por isto que Elixir facilita a creación de grandes aplicacións escalables, mantibles e cunha alta dispoñibilidade [14].

2.2.1 Erlang

Erlang foi creado en Ericsson, deseñado para mellorar o desenvolvemento das aplicacións de telecomunicacións. Neste entorno onde miles de conexións tiñan que ser atendidas de forma rápida nacen as características de Erlang e a súa máquina virtual.

2.2.2 Escalabilidade

O código na BEAM (Erlang, Elixir) execútase en fíos lixeiros illados entre si e intercomunicados por paso de mensaxes. Estes fíos denomínanse procesos.

Coa habilidade de correr centos de miles de procesos na mesma máquina, aproveitando os recursos de forma eficiente e achegando un gran nivel de escalado vertical, estes procesos contan tamén coa capacidade de comunicarse con procesos noutras máquinas virtuais na mesma rede de xeito transparente a quen programa, permitindo coordinar traballo entre diferentes nodos, obtendo escalado horizontal de forma sinxela.

2.2.3 *Behaviours* (modelos de comportamento)

No deseño e implementación de sistemas distribuídos altamente escalables e tolerantes a fallos, hai patróns ou modelos de comportamento que se repiten con frecuencia (supervisión de procesos, procesos xestores de certa información, procesos que ofrecen servizos, etc.) Elixir permite implementar este tipo de funcionalidades con moi pouco esforzo, grazas aos `behaviours`, unha abstracción que funciona de forma semellante ao patrón GoF Template [15], frecuentemente implementada mediante herdanza de clases en programación orientada a obxectos.

O exemplo mais sinxelo de `behaviour` é o `GenServer`, que permite crear procesos de larga duración capaces de atender peticións e ter estado propio. Se queremos implementar un módulo con este `behaviour` debemos simplemente implementar as funcións obligatorias e engadir funcionalidade ás opcionais. Como exemplo, na documentación de Elixir crean un *stack* 2.1 cun `GenServer` en apenas 30 liñas:

```
1 defmodule Stack do
2   use GenServer
3
4   # Client
5   def start_link(default) when is_list(default) do
6     GenServer.start_link(__MODULE__, default)
7   end
8
9   def push(pid, element) do
10    GenServer.cast(pid, {:push, element})
```

```
  end

  def pop(pid) do
    GenServer.call(pid, :pop)
  end

  # Server (callbacks)
  @impl true
  def init(stack) do
    {:ok, stack}
  end

  @impl true
  def handle_call(:pop, _from, [head | tail]) do
    {:reply, head, tail}
  end

  @impl true
  def handle_cast({:push, element}, state) do
    {:noreply, [element | state]}
  end
end
```

Listing 2.1: Exemplo dun *stack* mediante uso dun `GenServer`

2.2.4 Tolerancia a fallos

Elixir segue o espírito de Erlang ("*let it crash*"): como a execución de código adoita dividirse en procesos, se topamos cun fallo inesperado nalgún deles ese proceso "rompe", estendendo o erro ó proceso ou procesos conectados a el, nos cales podemos detectar a situación e reiniciar o proceso nun bo estado e continuar coa execución, ou manexar o erro si se prefire. Esta filosofía simplifica o código do proceso orixinal, liberándoo da lóxica de xestión de erros e favorecendo a separación de responsabilidades e a reutilización.

2.2.5 Relevancia

Grazas ao seu enfoque á escalabilidade sinxela xunto cun fantástico *framework* web, Elixir é a base perfecta para o desenvolvemento de aplicacións web escalables e mantibles cun mínimo esforzo. Exemplos de produtos creados usando esta linguaxe de programación son, entre outros: Discord [16], unha aplicación de voz por IP (VoIP) capaz de manexar 140 millóns de usuarios activos mensualmente; Heroku [17], unha plataforma coma servizo; Pinterest[18], unha rede social centrada en imaxes; ou PepsiCo [19].

2.3 Framework Phoenix

Phoenix [20] é o marco de desenvolvemento de aplicacións web principal de Elixir, deseñado para facilitar a creación de aplicacións web interactivas e modernas de forma sinxela e sen necesidade de moitas liñas de código ou diferentes sistemas. Phoenix usa o Modelo-Vista-Controlador como arquitectura das aplicacións, unha integración sinxela con bases de datos usando o paquete de Elixir *Ecto* [21], e moitas funcionalidades comúns para falicitar un desenvolvemento rápido. Entre estas funcionalidades destacan para o noso proxecto os *Channels* e *LiveView*.

2.3.1 Channels

As canles de Phoenix son unha das principais ferramentas para engadir funcionalidade en tempo real as súas aplicacións: permiten unha comunicación "case en tempo real" con e entre millóns de clientes conectados. Os clientes, unha vez establecida a conexión co servidor, poden subscribirse a diferentes temas facendo coñecer ó servidor o seu interese nas mensaxes asociadas a estes temas.

2.3.2 LiveView

LiveView [22] proporciona HTML *renderizado* no servidor para obter unha vista dinámica e reactiva sen necesidade de encher o *front-end* con JavaScript. A conexión do cliente e o servidor pasa a ter asignada un estado a partir do cal se constrúe a páxina para mostrarlle o cliente; no caso de que algunha mensaxe cambie o estado, os cambios que deben efectuarse na páxina mándanselle ó cliente para actualizar a vista.

2.4 Probas baseadas en propiedades

As probas baseadas en propiedades son unha técnica de *testing* na que se emprega unha ferramenta capaz de xerar valores de entrada aleatorias para as probas de forma automática. Esta técnica orixinouse como alternativa ás probas de unidade, nas que para comprobar o correcto comportamento do software é necesario escribir moitos casos de proba, cada un coa súa propia entrada e saída escollida a man.

Pola contra, no mundo das probas baseadas en propiedades, no canto de casos de proba concretos defínense propiedades lóxicas que o software debe de cumprir para un rango de entradas, e a ferramenta de elección xerará entradas aleatorias para comprobar se as propiedades se manteñen ou non.

En 2.2 móstrase unha comparación de como se definirían probas en ambas técnicas en relación á funcionalidade de sumar dous números enteiros.

```
# unit testing
x = 1
y = 2
assert(x + y, 3)

# property-based testing
forall {x, y} <- {integer(), integer()} do
    z = x + y
    assert(is_integer(z) and (z > x) and (z > y))
end
```

Listing 2.2: Exemplo mínimo comparativo entre probas de unidade e probas baseadas en propiedades

As probas baseadas en propiedades poden dividirse en dous tipos: por unha banda están as probas que tratan con propiedades puras (aquelas que proban código sen efectos secundarios ou estado, como o exemplo anterior), tamén denominadas propiedades *stateless*, e por outro lado están aquelas que pola contra si teñen estado ou "efectos secundarios" e o seu código é impuro (isto é, non ten transparencia referencial), que se coñecen como propiedades *stateful*.

Este último tipo de probas resultaron ser unha peza chave á hora de comprobar o correcto comportamento do código do proxecto, xa que para a lóxica do xogo se escribiu unha propiedade *stateful* cunha máquina de estados para modelar o comportamento do xogo e verificar os resultados esperados en cada fase con cada acción nun gran conxunto aleatorio de probas.

Aínda que hai multitude de ferramentas para realizar este tipo de probas, as máis relevantes no ecosistema de Elixir e Erlang son `PropCheck` e `PropEr`, que teñen multitude de funcionalidades e están extensamente documentadas e explicadas, como no libro *Property-Based Testing with PropEr, Erlang, and Elixir* [23], que se usou como referencia e banco de información no proxecto.

Capítulo 3
Deseño

No terceiro capítulo falaremos dos requistos do proxecto, o deseño das aplicacións a través do modelo c4 e o deseño da interface.

3.1 Requisitos

A continuación listaranse tanto os requisitos funcionais como os non funcionais que ten que cumprir a aplicación, despois de explicar os cambios de deseño que se aplicaron na aplicación, xa que esta implementa unha versión modificada do xogo Mafia.

3.1.1 A nosa variante de Mafia

Debido a que existen distintas variacións sobre o xogo necesitamos concretar as regras da nosa versión específica. Esta basease mais na versión de *Town of Salem* a cal engade fases adicionais dentro do día para moderar o discurso.

Na nosa versión o día divídese en 5 posibles fases a maiores nas cales diferentes accións son posibles (ver figura 3.1) comenzando a partida na fase de tarde para non permitir xuizos no primeiro día da partida. As principais fases son as da mañá, acusacións e tarde. A mañá e a tarde son tempos de discusión entre os xogadores, despois e antes da noite, onde a única acción posible é a comunicación entre os xogadores. No medio do día execútase a fase de acusacións onde os xogadores poden acusar a outra persoa para ir a xuízo; se ninguén acumula acusacións suficientes pásase á tarde, pero no caso contrario a persoa é enviada a xuízo e comeza a fase de defensa. Na fase de defensa o xogador acusado intenta convencer ós demais da súa inocencia. Acto seguido na fase do xuízo os xogadores deciden se o acusado morre ou vive, e transiciónase á tarde. Por defecto, as fases contan cun tempo predeterminado de duración e avanzan automaticamente ó final deste. En canto morre unha persoa na noite ou no xuízo avalíanse as condicións de vitoria para saber se o xogo acabou. No noso caso o campesiñado vence se elimina a todos os xogadores da mafia. Pola contra, a mafia gaña se

o campesiñado deixa de ser a maioría entre os vivos, xa que neste caso o campesiñado non pode conseguir unha maioría nas acusacións e non dispón de ferramentas para eliminar aos mafiosos.

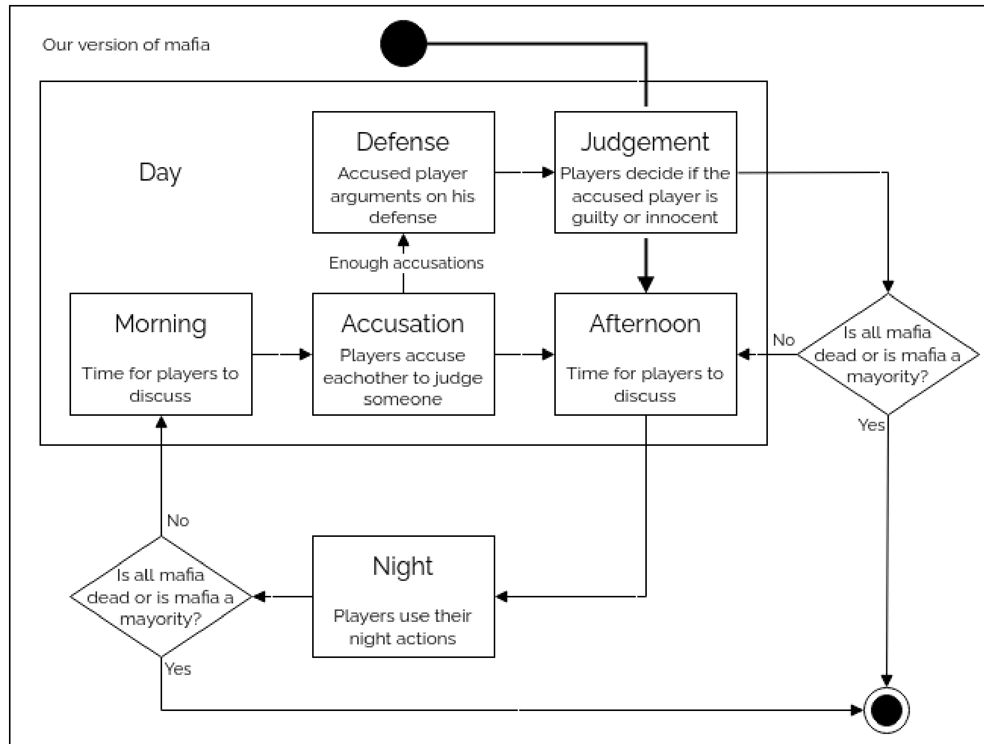

Figura 3.1: Fases da nosa versión de mafia

3.1.2 Requisitos funcionais

Requisitos do motor de xogo

- **Xogador e lista de xogadores**
  Modelar a estrutura que representa a un xogador e a lista destes. Un xogador necesita conter a información do seu nome, o seu rol e se está vivo ou non.

- **Acusacións**
  Permitir acusacións entre xogadores, cada xogador só pode acusar a unha persoa simultaneamente. No caso de que as acusacións superen a maioría dos xogadores, devolver a persoa acusada.

- **Xuízo dun xogador**
  Manexar votos dos xogadores a favor ou en contra dunha persoa en xuízo. A persoa en xuízo non pode votar. No caso dun empate nos votos o resultado é inocente.

- **Accions Nocturnas**
  Procesar a selección de obxectivos do doutor, o sheriff e a mafia. Non se pode seleccionar a xogadores mortos. Os mafiosos non poden seleccionar a outro mafioso. Ademais débense procesar as interaccións entre accións como no caso no que o doutor cure á vítima da mafia.

- **Maquina de Estados**
  Modelar a máquina de estados que implementa a fases do xogo e restrinxe as accións do xogo a fase correspondente.

- **Axustes da partida**
  Estrutura de axustes que establece a lista de roles a usar e a duración das fases. A partida usará esta estrutura para configurarse.

- **PubSub de comunicación cos xogadores**
  Un rexistro de comunicación cos xogadores onde a partida publicará as actualizacións públicas da partida ós xogadores desa partida e actualizacións privadas a xogadores individualmente.

- **Supervisor de partidas**
  Supervisor para crear instancias de partidas, as cales corren simultaneamente.

Requisitos da interface

- **Creación de partidas**
  Permitir crear unha nova partida para a persoa usuaria.

- **Invitar a unha partida**
  Crear un enlace identificativo da partida creada para invitar a máis xogadores.

- **Unirse a unha partida**
  Entrar nunha partida creada a partir do enlace de invitación.

- **Establecer o nome do xogador**
  Asociar un nome introducido pola persoa usuaria para ser usado como nome de xogador na partida.

- **Cambios de Axustes**
  Cambiar os axustes predeterminados de número de roles específicos e duración das fases da partida.

- **Comezar a partida**
  Unha vez todos os xogadores estean listos debe comezar a partida.

- **Mostrar a información da partida**
  Antes de comezar a partida deberase mostrar a lista de xogadores e ó comezar a partida mostrar ademais os roles coñecidos dos xogadores, a fase, o tempo restante e se os xogadores están vivos ou mortos.

- **Chat de texto**
  Un chat para permitir a comunicación entre xogadores dentro da páxina, pola noite só permitirá a comunicación entre os mafiosos.

- **Acusar**
  Acusar a calquera dos xogadores vivos durante a fase de acusación.

- **Seleccionar obxectivo da acción nocturna**
  Seleccionar o xogador o cal investigar, curar ou votar para matar pola noite dependendo do rol.

- **Votar**
  Na fase de xuízo permitir ós xogadores que non son o acusado votar a favor, en contra ou absterse.

- **Crear unha nova partida ó finalizar unha**
  Ó finalizar a partida para facilitar xogar outra partida entre os mesmos xogadores deberase crear unha nova partida e incorporar a todos os xogadores.

- **Manter a vista actualizada co estado da partida**
  A vista deberá estar actualizada para mostrar o estado actual da partida e os cambios desta.

3.1.3 Requisitos non funcionais

- Escalabilidade
  A aplicación ten que ser sinxela de escalar para acomodar a mais usuarios e partidas simultáneas.

- Tolerancia a fallos
  No caso de ocorrer algún fallo inesperado o sistema debe ser capaz de recuperarse e manter o estado das partidas.

- Responsive mobile-first design
  A páxina debe mostrase de forma adecuada en todo o rango de dispositivos, dende un móvil a un ordenador de mesa.

- Sinxeleza e accesibilidade
  O uso da aplicación debe ser intuitivo e simple para facilitar a súa usabilidade e lexibilidade.

3.2 Arquitectura: modelo C4

O modelo C4 é un acercamento que prioriza a abstracción para modelar a arquitectura do noso sistema, con diferentes vistas en cada escala. Comezamos cunha vista do contexto do sistema e diminuimos o nivel de abstracción paso a paso para engadir información mais detallada.

3.2.1 Diagrama de Contexto do sistema

O diagrama de contexto mostra como o noso sistema interactúa co seu exterior, a grandes trazos. Mostra con que usuarios ou outros sistemas se comunica o noso sistema.

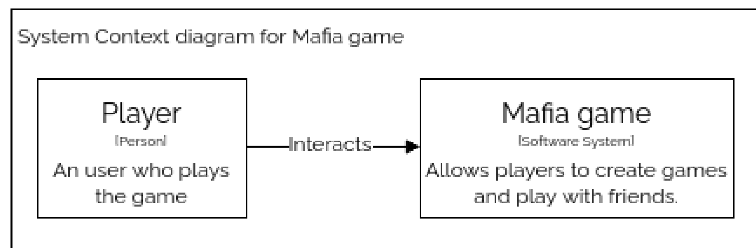

Figura 3.2: Diagrama de contexto do sistema

Como se observa no noso diagrama de contexto 3.2, o noso sistema non require de outros sistemas para funcionar e só interactúa co usuario final.

3.2.2 Diagrama de Contedor

No seguinte paso "desmóntase" o sistema e sepárase nos distintos contenedores que o forman.

No noso diagrama de contedor 3.3 pódese ver como a interface está separada do motor de xogo e comunícanse entre elas por paso de mensaxes para procesar as accións do xogador. Non contamos cunha base de datos, xa que o motor de xogo encargarase de gardar o estado das partidas actuais en memoria.

3.2.3 Diagramas de Compoñente

Estes diagramas mostran unha versión máis completa de cada contedor do sistema cos seus compoñentes máis importantes e as súas responsabilidades. Como xa vimos no nivel de

contedor, contamos con dous compoñentes principais: Mafia Engine e Mafia Interface.

Mafia Engine

No diagrama do motor de xogo 3.4, móstrase as diferentes partes deste. O supervisor da aplicación inicializa a aplicación e supervisa ó rexistro de partidas (`Game Registry`), ó supervisor das partidas (`Game Supervisor`) e ó sistema de publicacións das partidas (`Game PubSub`): no caso de que algún destes elementos termine de forma inesperada, o supervisor da aplicación reinicia unha nova instancia. No rexistro das partidas gárdase a relación dos identificadores das partidas co seu proceso onde está a executarse cada unha. O supervisor de partidas encárgase de comezar as partidas e reinicialas en caso dun erro inesperado, unha relación de líder-traballadores. As partidas pola súa parte reciben as mensaxes da interface, executando as regras do xogo e manexando o estado da súa partida: en cada cambio de estado, o proceso garda este estado na ETS (*Erlang Term Storage* [24]) para no caso de reiniciarse poder retomar a partida no último estado coñecido. Por último, cada partida publica actualizacións á interface a través do servidor de publicacións e subscricións do motor de xogo ao cal a interface subscríbese ó entrar na partida.

Mafia Interface

Na segunda parte do sistema, a interface 3.5 está construída no *framework* de Phoenix usando a *renderizacion* no servidor de LiveView. A estrutura de Phoenix é dun Model-View-Controler pero o noso modelo reside fóra de Phoenix coma unha dependencia ó proxecto, polo que a interface consiste, de forma simplificada, na vista e no controlador. O xogador conéctase a páxina web por medio do `Endpoint` e establece unha conexión ó servidor persistente establecendo un *websocket* e un estado asignado a este. O controlador crea o HTML usando o módulo da vista en base ó estado e envíallo ó xogador: cando este efectúa unha acción no navegador, mándase o evento ó controlador, o cal realiza a acción e se o estado cambia, reavalíanse as partes relevantes dos cambios e envíanse os cambios do HTML necesarios ó xogador. A vista axúdase dos patróns (`Templates`) para definir o HTML dos tres casos: a páxina principal, a sala de espera da partida e xogar a partida. A maiores o controlador fai uso da tecnoloxía dos canles de phoenix 2.3.1 para manexar un chat de texto para que os xogadores se comuniquen.

3.2.4 Código

Os diagramas de código son o cuarto nivel do modelo C4 e correspóndense tradicionalmente con diagramas de clases. Ao ser Elixir unha tecnoloxía funcional estes diagramas de código non resultan tan informativos, polo que os consideramos opcionais. Alternativamente,

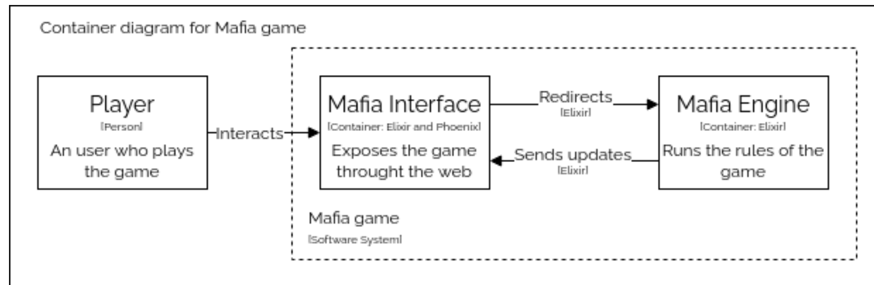

Figura 3.3: Diagrama de contedor do Xogo de Mafia

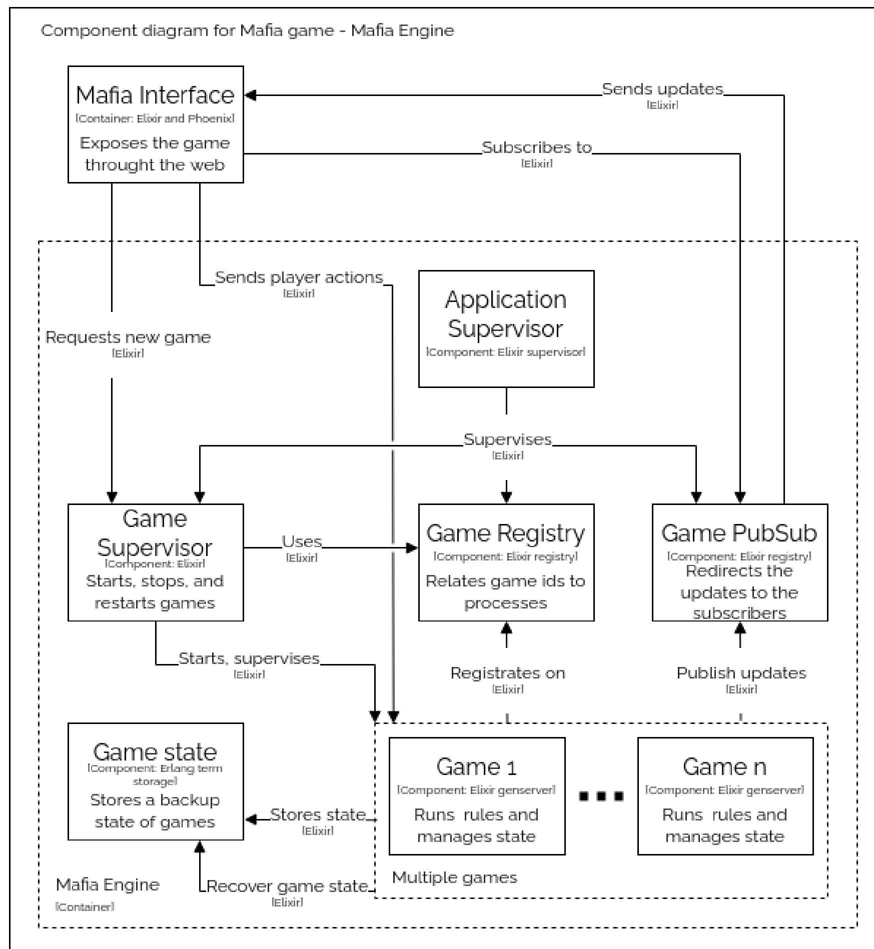

Figura 3.4: Diagrama de compoñente do Mafia Engine

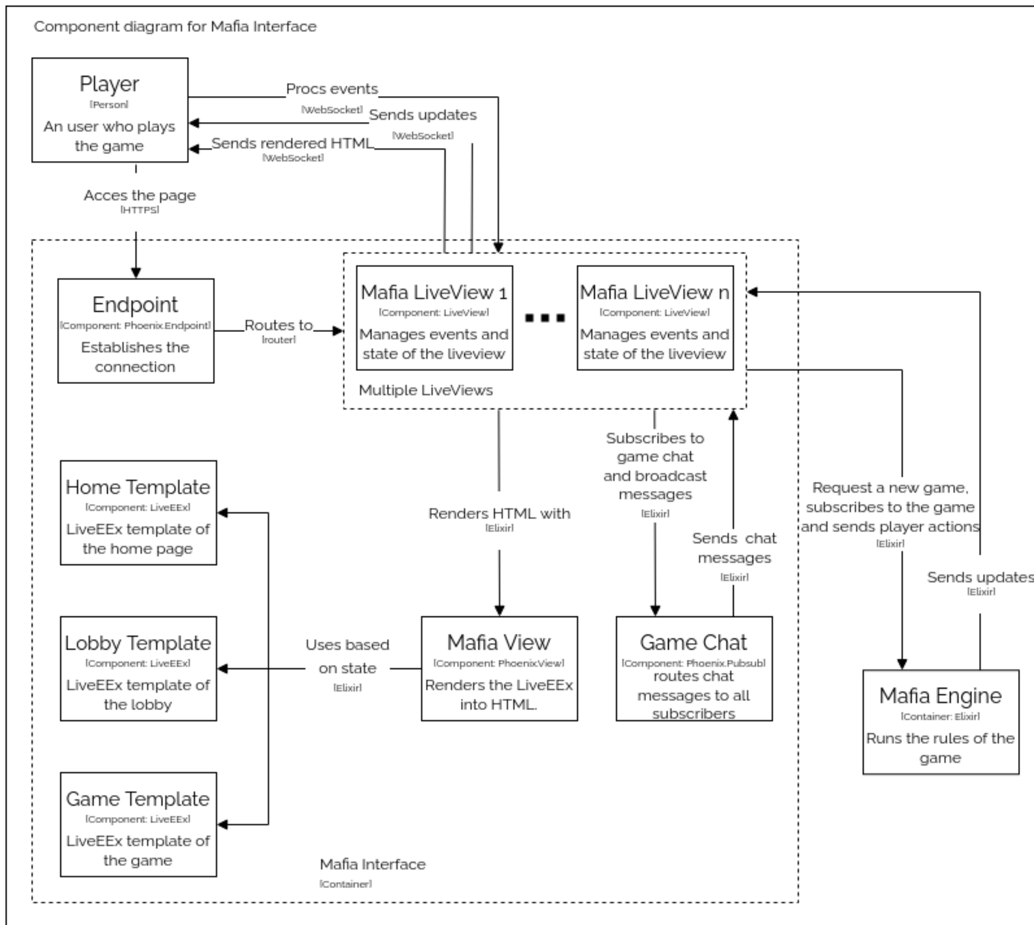

Figura 3.5: Diagrama de compoñente do Mafia Interface

e dado que no noso caso o compoñente máis complexo é a partida dentro do motor de xogo, para especificar o seu comportamento usamos un diagrama de estados que representa os seus estados e as súas transicións (figura 3.6).

O compoñente da partida está deseñado en base a nosa versión de Mafia 3.1 pero con varias diferencias. A partida ó inicializarse comproba se existe un estado de *backup* (o cal ocorrería no caso de que se estivera a xogar a partida e xurdira un erro que levara ao seu peche inesperado) para recuperar ese estado no caso de que exista. Se non, a partida pasa o estado inicializada onde os xogadores poden unirse a esta e cambiar os axustes da partida (xa que se tratará inequivocamente dunha partida nova). Ó comezar a xogar, a partida pasa ó bucle de estados principal de día e noite como explicamos na nosa versión. Por último, este conta tamén cun estado de partida finalizada antes de acabar, onde se revelarán os roles de tódolos xogadores e mostrar quen gañou a partida.

3.3 Deseño da Interface

Para o deseño da interface creouse un deseño pensado para dispositivos móbiles 3.7, o cal é facilmente extensible para dispositivos mais grandes de forma sinxela, algo que non poderiamos garantir se orientasemos o deseño ó escritorio. A navegación entre as distintas vistas fíxose da maneira mais sinxela e directa para que o uso da aplicación sexa simple.

Cando estamos a xogar unha partida requirimos mostrar diferentes cualidades dos xogadores cun espazo moi limitado, polo que decidimos aforrar espazo de diversas formas. Primeiro, como as accións están limitadas dependendo da fase, estas ocupan todas o mesmo lugar e aparecen de forma dinámica. A maiores, introdúcese o uso do estilo nas entradas dos xogadores para mostrar o seu estado: se están en cursiva e letra gris o xogador está morto; se está en negrita e a entrada conta cun fondo escuro, o xogador está seleccionado como obxectivo da acción actual (por exemplo, como acusado na quenda de acusacións).

Para permitir a configuración da partida se se desexa, o espazo do chat é intercambiable pola vista de configuración 3.8 xa que non é unha característica esencial para o xogo e así no ocupa un espazo dedicado soamente a isto.

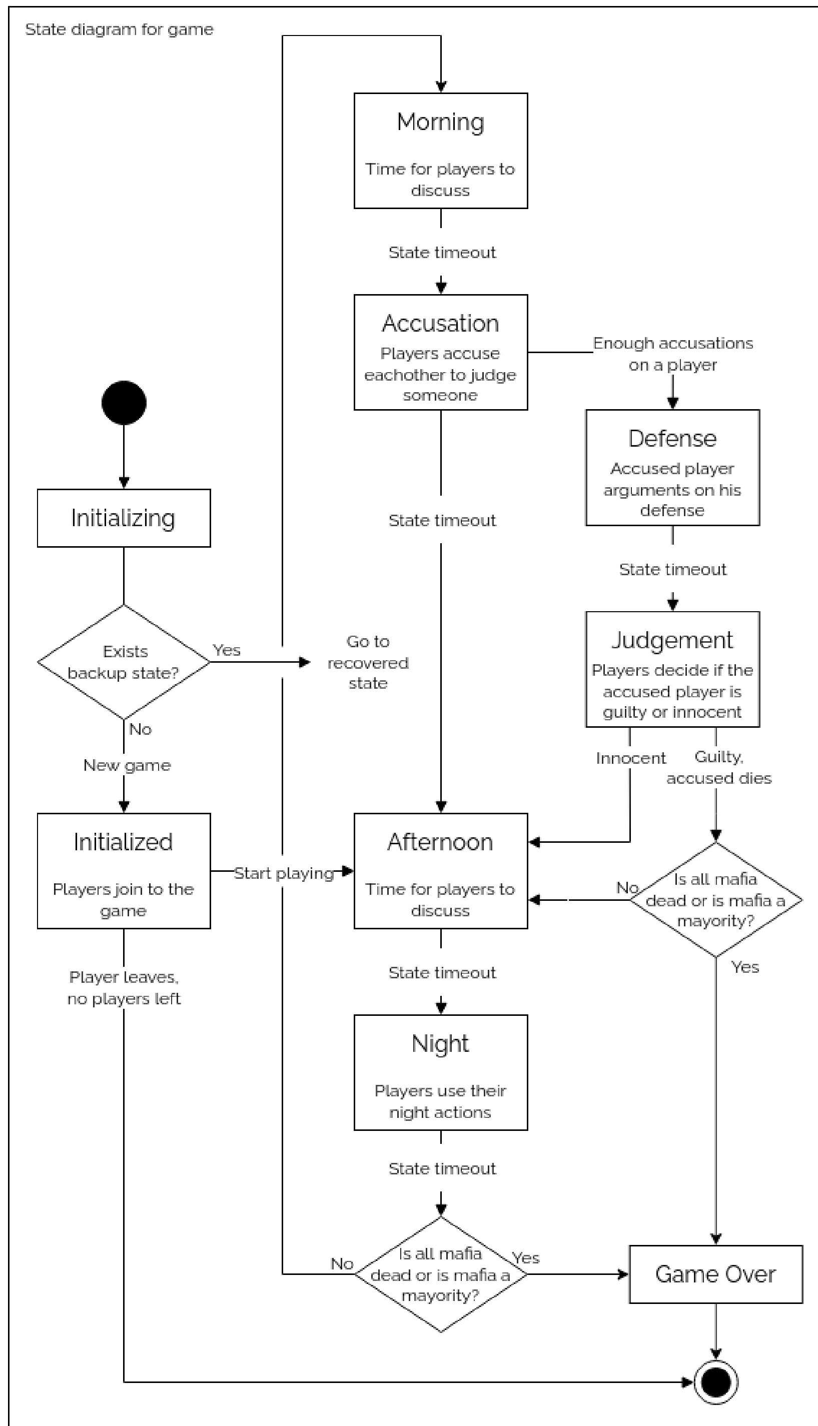

Figura 3.6: Diagrama de estado do xogo

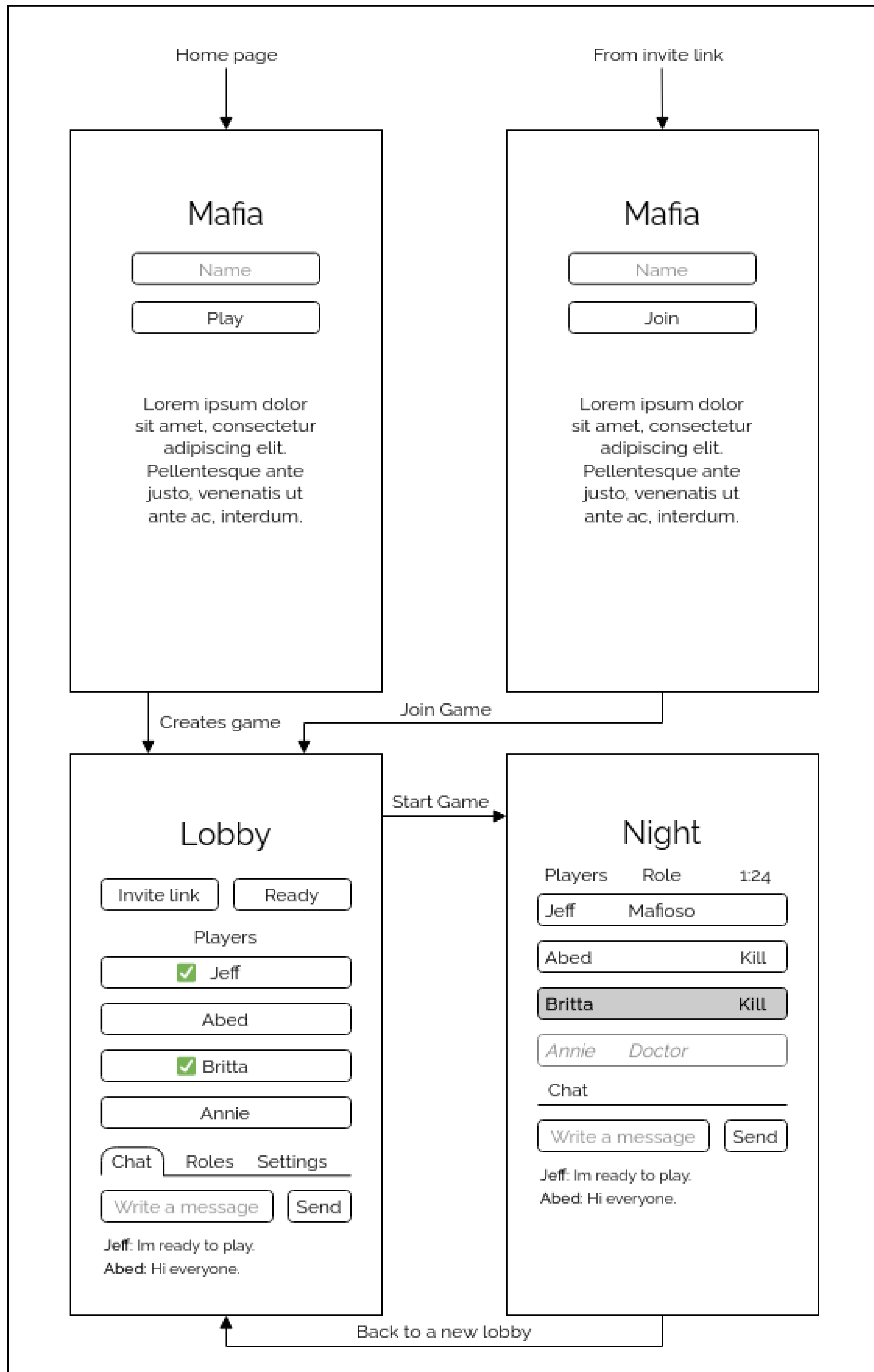

Figura 3.7: Deseño da interface

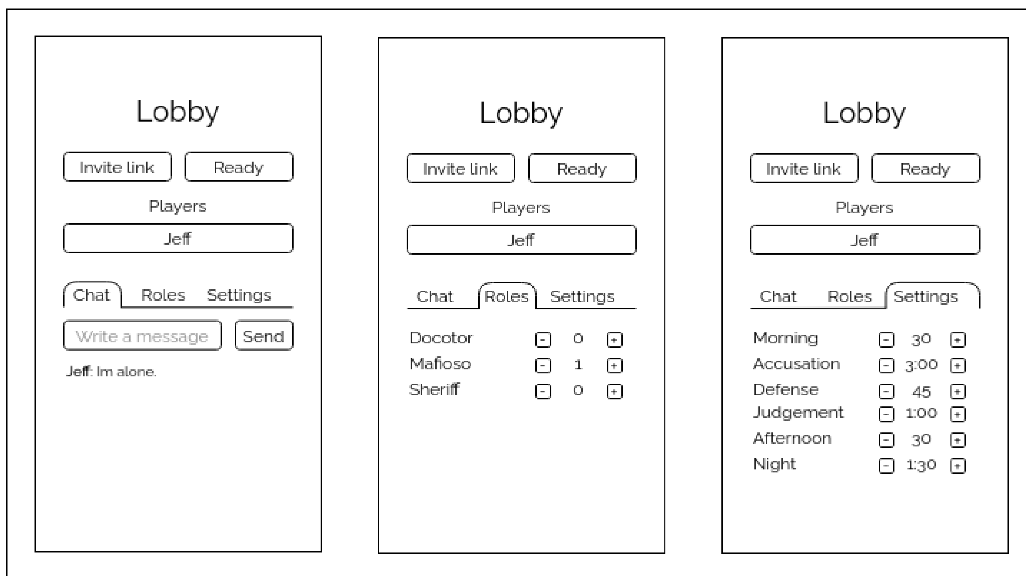

Figura 3.8: Deseño de axustes dentro do lobby

Capítulo 4

Implementación

NESTE capítulo falarase sobre como se poden facer aplicacións en Elixir e que tipo de aplicación se escolleu para este proxecto, seguido da introdución á implementación da aplicación, dividida nos seus compoñentes máis esenciais.

4.1 Aplicacións en Elixir

Para a implementación deste proxecto decídese illar a lóxica do xogo da interface deste, facendo posible, no caso de que queiramos implementar outro tipo de interface para o xogo, que a súa lóxica sexa completamente reutilizable. Así, `MafiaEngine` é unha aplicación de Elixir.

O ecosistema de Elixir proporciona ferramentas para o desenvolvemento de aplicacións de forma que estas se poidan compartimentar de forma sinxela. Isto quere dicir que é moi fácil preparar unha aplicación para o seu uso noutros proxectos. Á hora de xuntar múltiples aplicacións nunha, pódese facer uso das *aplicacións umbrella*, concepto propio da linguaxe que permite agrupar varias aplicacións de Elixir nun mesmo proxecto, englobadas baixo unha aplicación xeral.

Outra posibilidade que existe á hora de traballar con esta linguaxe é o manexo de dependencias nun proxecto, xa que o xestor de paquetes de Elixir fai moi fácil tanto publicar aplicacións no repositorio oficial, como incluír estas noutra aplicación ou proxecto.

Para a realización deste traballo, decidiuse deseñar a estrutura do proxecto empregando a segunda opción, ou sexa, co manexo de dependencias. As principais razóns polas que consideramos esta opción mellor que o uso de *aplicacións umbrella* son:

1. Poder publicar a lóxica do xogo (`MafiaEngine`) como un paquete en si mesmo, por separado da interface, o cal permite a súa reusabilidade como dependencia en diferentes interfaces ou proxectos da comunidade. Isto non é posible no caso de traballar con

aplicacións umbrella, xa que aínda que achega unha separación das aplicacións nun proxecto, non facilita o seu uso por separado.

2. A maiores, unha das principais desvantaxes que teñen as *aplicacións umbrella* é que non permiten empregar tódalas capacidades do xestor de paquetes da linguaxe, como pode ser o caso do control de versións das dependencias. Isto pode resultar moi útil á hora de desenvolver aplicacións, xa que raramente un proxecto é en todo momento compatible coa última versión das súas dependencias.

Tendo isto en conta, o traballo está divido en dúas aplicacións: `MafiaEngine`, que implementa a lóxica do xogo e está publicada como paquete, e `MafiaInterface`, unha aplicación Phoenix que aporta a aplicación web.

4.2 MafiaEngine

A aplicación de `MafiaEngine` divídese en múltiples módulos con funcionalidade diferenciada responsables de distintos aspectos do sistema.

Estructuras de datos

Para implementar a lóxica do xogo necesitamos saber como organizamos e manexamos os datos que representan o estado deste. Normalmente empezamos a engadir e modelar as estruturas en base o uso dunha base de datos pero neste caso escolleremos outro camiño. Usaremos as ferramentas que achega Elixir cons estruturas e módulos escribindo dunha maneira natural para a linguaxe e sen necesidade de ter en conta a base de datos de por medio. Desta maneira gañamos en claridade e simplicidade mantendo a opción de engadir unha base de datos no futuro.

Os módulos que representan as estruturas de datos na nosa aplicación son os seguintes:

`MafiaEngine.Player`
Define a estrutura dun xogador na partida e funcións para manexalo. O xogador consta do seu nome, rol e se esta vivo ou non.

`MafiaEngine.Players`
Define a lista de xogadores e funcións para manexala.

`MafiaEngine.Accusations`
Define a estrutura das acusacións dunha fase de acusacións: créase a partir do número necesario para enviar unha persoa a xuízo e garda un mapa das acusacións.
Conta con funcións para engadir ou eliminar acusacións de xogadores, e, no caso de

que ó engadir unha acusación esta sexa a derradeira acusación necesaria para enviar a unha persoa a xuízo, devolve a persoa acusada polo campesiñado.

`MafiaEngine.Votes`
Define a estrutura dunha votación do xuízo dun xogador, sabendo o xogador que está acusado a este non se lle permite a participación no seu propio xuízo. Proporciona as funcións para votar a favor ou en contra ou absterse e obter o resultado.

`MafiaEngine.NightActions`
Define o mapa das accións nocturnas coa funcionalidade da súa selección e deselección. A maiores a función `execute/1` serve para efectuar as accións dos roles pola noite convertendo a intención de cada rol nunha lista de eventos.

Co transcurso da partida, os datos cambian e é por isto que estes módulos a maiores conteñen a funcionalidade para transformar as estruturas e efectuar lóxica dependendo de cada un.

Como exemplo a estrutura de `MafiaEngine.NightActions` é un simple mapa e ten as funcións para seleccionar e deseleccionar obxectivos (ver 4.1).

```
1  defmodule MafiaEngine.NightActions do
2
3    def new(), do: %{}
4
5    def unselect(night_actions, actor) do
6      Map.delete(night_actions, actor)
7    end
8
9    def select(night_actions, actor, target) do
10     # Comproba que a acción e válida
11     with :ok <- check_action(actor, target) do
12       {:ok, Map.put(night_actions, actor.name, {actor.role,
   target.name})}
13     else
14       {:error, reason} -> {:error, reason}
15     end
16   end
17
18   # ...
19 end
```

Listing 4.1: Estructura das accións nocturnas

A maiores tamén conta coa función `execute/1`, xa que as accións dos xogadores non se transmiten directamente en eventos, senón que as accións pode que interactúen entre si. A

vítima da mafia é escollida entre as seleccións da mafia para morrer pero se esta é curada polo doutor, evita a súa morte. Para isto a función transforma as accións na lista de eventos que ocorren na noite como se pode ver nas liñas 4 a 8 do *snippet* 4.2, onde a mafia ataca, o doutor tenta curar e o sheriff investiga.

```
# Código simplificado

def execute(night_actions) do
  night_actions
  |> actions_to_list()
  |> mafia_attacks()
  |> doctor_heals()
  |> sheriff_investigates()
end

# ...

defp mafia_attacks(events) do
  {mafioso_events, events} =
    Enum.split_with(events, fn {_, role, _} -> role == :mafioso end)

  mafia_target =
    mafioso_events
    |> Enum.map(fn {_, _, target} -> target end)
    |> Enum.frequencies()
    |> Enum.max_by(fn {_, freq} -> freq end)
    |> elem(0)

  mafia_actor =
    mafioso_events
    |> Enum.map(fn {actor, _, _} -> actor end)
    |> Enum.take_random(1)
    |> List.first()

  [{mafia_actor, :kill, mafia_target} | events]
end

defp doctor_heals(events) do
  {doctor_events, events} =
    Enum.split_with(events, fn {_, role, _} -> role == :doctor end)

  Enum.map(events, &process_doctor_action(&1, doctor_events))
end


```

```
41 defp process_doctor_action({_actor, :kill, target} = event,
     doctor_events) do
42   healed =
43     Enum.find(doctor_events, :none,
44       fn {_actor, _role, heal_target} -> target == heal_target end)
45
46   case healed do
47     :none -> event
48     {actor, :doctor, target} -> {actor, :heal, target}
49   end
50 end
51 defp process_doctor_action(event, _doctor_events), do: event
52
53 defp sheriff_investigates(events) do
54   Enum.map(events, &process_sheriff_action/1)
55 end
56
57 defp process_sheriff_action({actor, :sheriff, target}), do:
58   {actor, :investigate, target}
59 defp process_sheriff_action(event), do: event
```

Listing 4.2: Función `execute/1`

Engadindo estado con `GenStateMachine`

Para agrupar as estruturas e terminar de implementar a lóxica necesitamos estado global e interactuable polos xogadores. A través dun `GenServer` podemos gardar o estado da partida, e a funcionalidade que depende do estado. Como modelamos a partida como unha máquina de estados podemos usar `GenStateMachine`, unha dependencia que implementa un `behaviour` que representa unha variación do `GenServer` para máquinas de estados.

O estado interno dunha partida será representado internamente como se mostra en 4.3.

```
1  {
2    :initialized,                     # Estado da maquina de estados
3    %{
4      :game_id => game_id,            # Identificador da partida
5      :settings => Settings.new(),    # Axustes da partida
6      :players => Players.new(),      # Xogadores da partida
7      :accusations => nil,            # Acusacións
8      :votes => nil,                  # Votos
9      :night_actions => nil           # Accións nocturnas
10   }
11 }
```

Listing 4.3: Representación interna das partidas

`GenStateMachine` manexa por nós o estado da máquina e achega utilidades para o código. Normalmente para implementar os *callbacks* dun `GenServer` usamos *pattern-matching* na cabeceira para implementar as diversas chamadas. Como nós queremos darlle diferente funcionalidade dependendo de en que estado se encontre a partida, podemos usar *pattern-matching* no estado. Para que isto sexa mais lexible e sinxelo podemos agrupar as chamadas usando unha opción de `GenStateMachine`. En lugar de usar `handle_event/4` podemos implementar funcións co nome correspondente ó estado por exemplo a cabeceira de función para votar na fase de xuízo sería `judgement(:cast, {:vote, vote, voter_name}, data)`.

Para transicionar de estado na máquina de estados usamos a resposta dunha función, cando esta sexa do tipo `{:next_state, new_state, updated_data}` a máquina transiciona ao novo estado dado por `new_state` e cos novos datos da partida.

A partida debe de contar con transicións automáticas por tempo debido á natureza do xogo: por exemplo, cando o tempo de discusión da mañá remata, a partida transiciona á fase de acusacións. Para isto podemos facer uso de *timeouts* no estado, o cal provoca unha autochamada ó xogo coa mensaxe especificada ó pasar o tempo do *timeout*.

Para establecer o *timeout* ó entrar nun estado e inicializar datos deste o inicio do estado `GenStateMachine` achega a funcionalidade de *enter calls*. Cada vez que se transicione de estado este recibe unha chamada co argumento `:enter` na cal podemos executar o código necesario, como preparar o número de acusacións requeridas ou establecer o *timeout* do estado (mostrado en 4.4).

```
1  #Código para manexar o estado de acusacións simplificado
2
3  #No caso de chgar as acusacións dende :initializing non cambiamos
       os datos da partida
4  def accusation(:enter, :initializing, data), do:
5    success(:keep_state_and_data, :accusation, data,
6            [{:state_timeout, data.settings.timer.accusation,
       :go_next}])
7
8  # Entramos no estado
9  def accusation(:enter, _old_state, data) do
10   required = half_players(data.players) + 1
11   # inicializamos a estructura das acusación
12   updated_data = update_accusations(data, Accusations.new(required))
13   success(:keep_state, :accusation, updated_data,
14                   # Establecemos o timeout do estado
15           [{:state_timeout, data.settings.timer.accusation,
       :go_next}])
16 end
17
```

```
18 def accusation(:cast, {:accuse, accuser_name, accused_name}, data)
     do
19     accuser = Players.get(data.players, accuser_name)
20   accused = Players.get(data.players, accused_name)
21
22   with {:ok, accusations} <- Accusations.accuse(data.accusations,
     accuser, accused) do
23     updated_data = update_accusations(data, accusations)
24     # Se non se chegan os votos requeridos non se cambia de estado
25     success(:keep_state, :accusation, updated_data)
26   else
27     {:accused, accused, accusations} ->
28       updated_data =
29         data
30         |> update_accusations(accusations)
31         |> update_votes(Votes.new(accused))
32       PubSub.pub(data.game_id, {:game_update, :accused, accused})
33       # Se  se chegan os votos requeridos cambiase a defensa
34       success(:next_state, :defense, updated_data)
35 end
36
37 def accusation(:cast, {:withdraw, accuser}, data) do
38   accusations = Accusations.withdraw(data.accusations, accuser)
39   updated_data = update_accusations(data, accusations)
40   success(:keep_state, :accusation, updated_data)
41 end
42
43 # Cando o timeout se acaba pasamos a tarde
44 def accusation(:state_timeout, :go_next, data), do:
45   success(:next_state, :afternoon, data)
46 # Ignoramos outro tipo de accións
47 def accusation(_, _, _), do: :keep_state_and_data
```

Listing 4.4: Funcións no estado de acusación

Na aplicación executanse múltiples instancias do xogo ó mesmo tempo polo que necesitamos unha maneira de saber que proceso é o que esta asignado á partida particular coa que queremos interactuar. Para isto nomeamos aos procesos a través de `Registry.Game`. Este é un rexistro único o cal garda a información de que proceso executa a partida asociada a un `game_id`.

Comunicación cos xogadores

Os clientes poden comunicarse coa partida sabendo o identificador desta, pero aínda necesitamos o camiño de volta: actualizar ós xogadores en grupo e individualmente ante eventos

relevantes no transcurso da partida. Para iso temos un servizo de publicación-subscripción (en diante, “PubSub”) no cal os clientes subscríbense co seu nome de xogador á partida que están a xogar. O PubSub garda a dirección dos clientes e permite ó proceso da partida enviar mensaxes a xogadores individuais ou a toda a partida.

Supervisión e tolerancia a fallos

No mundo real ás veces as cousas fallan e tamén temos que estar preparados para estes casos. Xa que a execución do noso código está compartimentada nos procesos podemos supervisar estes e reaccionar no caso de que algo vaia mal. En primeiro lugar, contamos co supervisor da aplicación, o cal coñece os seus fillos e, no caso de que algún falle, reiníciase. Como xa comentamos, este supervisor supervisa o rexistro de nomes dos xogos, o servizo PubSub que usan os clientes, e un supervisor adicional que supervisa ás partidas. O supervisor das partidas non ten unha lista de fillos estática, xa que estas créanse e rematan de forma dinámica. Para implementar este caso úsase un supervisor dinámico, e coas chamadas de `start_children` e `stop_children` podemos crear e eliminar fillos deste. Xa que as partidas acaban pola súa conta, se a mensaxe de terminar é normal, o supervisor dinámico non inicia de novo a partida 4.5. Así se unha partida ten un fallo inesperado esta é automaticamente iniciada de novo, pero se non facemos nada para impedilo, perderemos os datos da partida, xa que o novo proceso comeza de novo no estado inicial.

Para preservar o estado necesitamos gardalo anteriormente para poder recuperalo no reinicio do xogo. Unha Erlang Term Storage (ETS) servirá como almacén chave-valor para o estado das partidas en curso. Ó comezo da partida o estado desta gárdase na ETS e despois de cada chamada con éxito actualízase. No caso de que a partida falle e teña que reiniciarse, esta comproba na ETS se existe o estado dunha partida co seu mesmo identificador e se é o caso recupérase. Ao finalizar a partida de forma normal esta elimina o estado da ETS e o identificador da partida queda libre.

```
1  defmodule MafiaEngine.GameSupervisor do
2    use DynamicSupervisor
3
4    alias MafiaEngine.Game
5
6    def start_link(_options), do:
7      DynamicSupervisor.start_link(__MODULE__, :ok, name: __MODULE__)
8
9    def start_game() do
10     game_id = gen_game_id()
11     {:ok, _} = DynamicSupervisor.start_child(__MODULE__, {Game,
       game_id})
```

```
    game_id
  end

  def stop_game(game_id) do
    :ets.delete(:game_state, game_id)
    DynamicSupervisor.terminate_child(__MODULE__,
   pid_from_game_id(game_id))
  end

  def exists_id?(game_id) do
    not(Registry.lookup(Registry.Game, game_id) == [])
  end

  @impl true
  def init(:ok), do:
    DynamicSupervisor.init(strategy: :one_for_one)

    # ...

end
```

Listing 4.5: Supervisor das partidas

4.3 MafiaInterface

No *router* da nosa aplicacion de Phoenix establecemos as rutas da nosa páxina web 4.6: no noso caso ou ben accedemos á páxina raíz ou contamos cun `game_id` na URL. Nos dous casos a conexión é manexada polo módulo `MafiaInterfaceWeb.Mafia` pasándolle como parámetro o identificador da partida no caso de que exista:

```
# mafia_interface/lib/mafia_interface_web/router.ex:17
scope "/", MafiaInterfaceWeb do
  pipe_through :browser

  live "/", MafiaInterfaceWeb.Mafia
  live "/:game_id", MafiaInterfaceWeb.Mafia

end
```

Listing 4.6: Rutas soportadas no *router* da nosa interface

O módulo `MafiaInterfaceWeb.Mafia` é o noso controlador da vista e polo tanto ten que implementar `mount/3` e `render/1`. A primeira para que o cliente se conecte, e a segunda

para devolver o contido a visualizar. `render/1` colle como parámetro o estado asinado ó *socket* para devolver unha vista acorde ó estado.

Dado que a nosa implementación da páxina ten varios patróns dependendo de se estamos xogando, na sala de espera, ou entrando a xogar pero só un controlador, este *renderiza* diferentes patróns en base ó estado do *socket*:

```
# mafia_interface/lib/mafia_interface_web/live/mafia.ex
# Render in function of state assign

@impl true
def render(%{:state => :create} = assigns), do:
  MafiaView.render("home.html", assigns)

@impl true
def render(%{:state => :join} = assigns), do:
  MafiaView.render("home.html", assigns)

@impl true
def render(%{:state => :lobby} = assigns), do:
  MafiaView.render("lobby.html", assigns)

@impl true
def render(%{:state => :playing} = assigns), do:
  MafiaView.render("playing.html", assigns)
```

Listing 4.7: Función `render/1` do controlador

Adicionalmente, cando o cliente se conecta, `handle_params/3` xestiona os posibles parámetros da URL: no noso caso, se a URL non especifica `game_id` móstrase a páxina para crear unha sala. No caso que teñamos un `game_id`, se a persoa usuaria xa ten un nome asociado pasa á sala de espera, se non pregúntaselle primeiro por un nome.

As interaccións na vista que xeran eventos para o controlador xestiónanse no *callback* `handle_event/3` os cales redireccionan as chamadas á partida, ou executan accións no controlador.

En canto o xogador se une á partida o controlador suscríbese ó PubSub do motor do xogo e ó PubSub de Phoenix co tema do identificador da partida. Desta forma polo primeiro poderá recibir mensaxes da partida e polo segundo intercambiar mensaxes cos demais xogadores. Requirimos dunha canle separada para as actualizacións do xogo xa que o motor é o que sabe cando ten que enviar as mensaxes pero non ten coñecemento da tecnoloxía das canles de Phoenix xa que dependen da interface. Do mesmo xeito, as mensaxes do chat non involucran ó motor de xogo, e usar o seu PubSub sería engadirlle funcionalidade non desexada a este. As mensaxes recibidas por estas canles non son xestionadas como eventos se non como mensaxes

pola función `handle_info/2`.

Os patróns de LiveView usan a extensión .leex (Live EEx), e funcionan como os patróns de Elixir embebido .eex, coa mellora de que minimizan os datos enviados pola conexión. Co uso de Elixir embebido podemos escribir segmentos de código da linguaxe no HTML, os cales serán avaliados para ser enviados ó cliente. En 4.8 podemos ver como, se o valor de `@players` ou `@ready` cambia, LiveView executará de novo o segmento e enviará o novo contido.

```
<div id="players">
  <%= for p <- @players do%>
    <div id="player">
        <%= if @ready[p.name] do "Ready " end %>
        <%= p.name %>
    </div>
  <% end %>
</div>
```

Listing 4.8: LiveEEx para mostrar ós xogadores

Esta forma de actualizar o HTML funciona moi ben se as accións requiren modificacións no servidor, pero no caso de barras de navegación engaden o percorrido de ida e volta ata o servidor cando non é necesario e pode solucionarse dende o cliente. Para engadir funcionalidade de JavaScript en LiveView hai que definir un *hook* no cliente que define as funcións a executar coas diferentes accións que pode sufrir un elemento do HTML, como por exemplo cando o elemento se engade á vista coa acción `mounted`. Así podemos engadir eventos de JavaScript e manexar a navegación entre axustes de tempo, roles e o chat na sala de espera.

Capítulo 5
Probas

NESTE capítulo falarase das diferentes técnicas de *testing* que se aplicaron no proxecto (probas de unidade, probas de integración, probas baseadas en propiedades, etc), para que partes ou compoñentes do código se empregaron e con que obxectivos.

5.1 Probas de unidade

As probas de unidade (*unit testing*) son unha forma de probar o código sinxela e directa, na cal se divide o código nos compoñentes básicos que o estruturan para facer probas sobre estes. Nestas probas, o programador ten que escoller unha serie de entradas a man e comprobar que a saída é a esperada.

Un exemplo dunha proba de unidade 5.1 sería na que se crea un xogador, chamado Abed, se lle asigna o rol de campesiño e se mata; o resultado esperado é que no campo que garda se o xogador está vivo indique que non o está.

```
test "killing a player sets their alive status to false" do
    abed =
        MafiaEngine.Player.new("Abed")
        |> MafiaEngine.Player.set_role(abed, :townie)
        |> MafiaEngine.Player.kill(abed)
    assert abed.alive == false
end
```

Listing 5.1: Exemplo dunha proba de unidade do proxecto

Os compoñentes do motor de xogo contan con dous tipos de probas de unidade: por un lado teñen un módulo de probas onde se comproba a funcionalidade de cada compoñente, e por outro lado tamén contan cunha serie de casos de proba dentro da documentación de cada módulo, as cales serven unha dobre función: ao engadir na documentación dos compoñentes

exemplos de uso das súas funcións, Elixir é capaz de empregar eses exemplos á hora de lanzar as probas da aplicación.

Un exemplo dunha proba que é parte da documentación dun módulo móstrase en 5.2. Nesta proba créanse dous xogadores que teñen que votar si unha terceira é ou non culpable; logo compróbase que só o xogador esperado está na lista dos que votaron culpable e o mesmo coa listaxe dos que votaron inocente. No caso inicial, como se dá un empate, o veredicto da votación é que a xogadora é inocente, mais se quitamos o voto do xogador que dicía que era inocente, o novo resultado é que a xogadora é culpable.

```
defmodule MafiaEngine.Votes do
  @moduledoc """
  This module defines the type for votes and functions to handle
    them.
  ## Examples
    iex> v = MafiaEngine.Votes.new("Annie")
    ...> jeff = MafiaEngine.Player.new("Jeff")
    ...> abed = MafiaEngine.Player.new("Abed")
    ...> {:ok, v} = MafiaEngine.Votes.vote(v, :guilty, jeff)
    ...> {:ok, v} = MafiaEngine.Votes.vote(v, :innocent, abed)
    ...> v.guilty
    #MapSet<["Jeff"]>
    iex> v.innocent
    #MapSet<["Abed"]>
    iex> MafiaEngine.Votes.result(v)
    :innocent
    iex> v = MafiaEngine.Votes.remove_vote(v, "Abed")
    ...> v.innocent
    #MapSet<[]>
    iex> MafiaEngine.Votes.result(v)
    :guilty
  """
  ...
```

Listing 5.2: Exemplo dunha proba na documentación do proxecto

Porén, como este tipo de probas non son suficientes para ofrecer unha alta confianza en que a aplicación ten o comportamento correcto, tamén se fixeron probas de integración para constatar a cohesión entre os compoñentes.

5.2 Probas de integración

As probas de integración (*integration testing*) pola súa parte involucran a varios dos compoñentes individuais do sistema ó mesmo tempo e comproban a súa funcionalidade de forma

conxunta. No noso caso, verificouse a integración da lóxica do xogo, xunto co supervisor de partidas.

En 5.3 móstrase un exemplo dunha partida entre tres xogadores os cales entran á partida e comezan a xogar. Cando chega a fase de acusacións estes acusan a un dos xogadores para na fase de xuízo votar culpable e eliminalo. Como son tres xogadores e un deles é o único mafioso, a partida acaba tanto se se elimina ó mafioso (por acabar con toda a mafia) ou a un campesiño (quedan vivos un campesiño e un mafioso e o campesiñado perde a maioría). Compróbase que a partida acaba e elimínase co supervisor.

```
test "voting test" do
    game_id = MafiaEngine.GameSupervisor.start_game()
    MafiaEngine.Game.add_player(game_id, "Troy")
    MafiaEngine.Game.add_player(game_id, "Britta")
    MafiaEngine.Game.add_player(game_id, "Shirley")

    expected = ["Shirley", "Britta", "Troy"]
    players  =
        game_pid(game_id)
        |> :sys.get_state()
        |> elem(1)
        |> Map.get(:players)
        |> MafiaEngine.Players.names()
    assert players == expected

    MafiaEngine.Game.next_phase(game_id)

    assert MafiaEngine.Game.started?(game_id)

    # Afternoon -> Night
    MafiaEngine.Game.next_phase(game_id)
    # Night -> Morning
    MafiaEngine.Game.next_phase(game_id)
    # Morning -> Accusations
    MafiaEngine.Game.next_phase(game_id)

    expected = :accusation
    phase = game_phase(game_id)
    assert phase == expected

    MafiaEngine.Game.accuse(game_id, "Troy", "Shirley")
    MafiaEngine.Game.accuse(game_id, "Britta", "Shirley")

    expected = :defense
    phase = game_phase(game_id)
```

```
36      assert phase == expected
37
38      # Defense -> Judgement
39      MafiaEngine.Game.next_phase(game_id)
40
41      MafiaEngine.Game.vote_guilty(game_id, "Troy")
42
43      MafiaEngine.Game.next_phase(game_id)
44
45      expected = :game_over
46      phase = game_phase(game_id)
47      assert phase == expected
48
49      MafiaEngine.GameSupervisor.stop_game(game_id)
50
51      assert not MafiaEngine.GameSupervisor.exists_id?(game_id)
52  end
53  defp game_phase(game_id), do:
54      game_pid(game_id) |> :sys.get_state() |> elem(0)
55  defp game_pid(game_id), do:
56      Registry.lookup(Registry.Game, game_id) |> List.first() |>
        elem(0)
```

Listing 5.3: Exemplo dun caso de integración do proxecto

Mesmo con estas probas, a lóxica do xogo non está verificada cunha confianza tan alta como se desexaría, polo que para mellorar este aspecto decidiuse facer probas baseadas en propiedades con estado.

5.3 Probas baseadas en propiedades

Como xa se comentou cando se contexualizaron as probas baseadas en propiedades 2.4, existen dous tipos de propiedades: as puras ou *stateless*, que proban código sen estado e libre de efectos secundarios, e as probas impuras ou *stateful*, que si lidian con aquel código que ten estado ou efectos secundarios.

Debido a que a lóxica do xogo non só ten un estado interno para xestionar os xogadores que hai na partida, en que fase está a partida, que accións se poden realizar nese moment, etc., senón que as accións que se poden realizar (por exemplo, asasinar pola noite se es da mafia) teñen efectos secundarios (seguindo o exemplo, executar esa acción implica que un xogador morre), as propiedades que se poidan escribir serán impuras ou *stateful*.

Para facer probas baseadas en propiedades en Elixir existen tres alternativas, cada unha coas súas vantaxes e desvantaxes:

- **PropEr** [25] é unha das ferramentas existentes para facer probas baseadas en propiedades máis poderosas. Está escrita en Erlang, é a opción de facto nesa linguaxe, e conta con soporte para probas puras e impuras, entre outras cousas. O problema que ten é manexar as súas macros, implementadas en Erlang, e que malia poderen ser empregadas dende Elixir, fan que o código quede menos lexible e mantible do desexado.

- **PropCheck** [26] é unha aplicación *wrapper*[1] escrita en Elixir arredor de PropEr. Conta coas mesmas características e coa vantaxe de permitir escribir as probas en Elixir con facilidade e maior lexibilidade.

- **StreamData** [27] é unha aplicación escrita na súa totalidade en Elixir para realizar este tipo de probas. Porén, en comparación coas outras dúas aplicacións, ten o problema de que lle faltan moitas das funcionalidades que as outras teñen (en concreto a capacidade de executar propiedades impuras ou *stateful*) e a día de hoxe ten un maior enfoque en ser usada para a xeración de datos que este tipo de ferramentas achegan que para realizar probas.

Tendo en conta isto, decidiuse empregar `PropCheck` para a realización das probas baseadas en propiedades neste proxecto, xa que a diferencia entre esta e `PropEr` é únicamente a lexibilidade das probas e `StreamData` non é quen de executar as propiedades que escribiremos.

O caso de uso das probas baseadas en propiedades neste proxecto foi o de comprobar o correcto funcionamento da lóxica do xogo. Para facer isto, empregouse a funcionalidade para definir máquinas de estados que `PropCheck` achega, a cal se explicará con *snippets* do proxecto.

5.3.1 Definindo propiedades con máquinas de estados

En 5.4 móstrase o inicio da definición do módulo de probas que contén a máquina de estados. Nas liñas 2-4 impórtanse as librerías necesarias para poder realizar estas probas: primeiro *ExUnit.Case* para poder definir casos de proba, seguido de *PropCheck* para traer tódala funcionalidade xeral da ferramenta e por último *PropCheck.StateM* para poder empregar as funcións específicas da máquina de estados de `PropCheck`.

```
1 defmodule PropStateTest do
2   use ExUnit.Case
3   use PropCheck
4   use PropCheck.StateM
5
```

[1] Unha aplicación *wrapper* é aquela que aporta unha API para manexar a API da aplicación que "envolve".

```
6  @opts [:verbose, {:constraint_tries, 500}, {:numtests, 1000},
     {:max_size, 80}]
7  property "stateful property", @opts do
8    forall cmds <- commands(__MODULE__, initial_state(12)) do
9
10     game_setup(initial_state(12))
11     {history, state, result} = run_commands(__MODULE__, cmds)
12     game_cleanup()
13
14     # lóxica para xestionar o resultado
15     ...
16   end
17 end
```

Listing 5.4: Definición da propiedade impura

Na liña 6 defínense as opcións que se queren dar á ferramenta, xa que os valores por defecto non valían para o noso caso: con *:verbose* facemos que mostre por pantalla máis información da usual durante as execucións; *:constraint_tries* é para aumentar o número de intentos que a ferramenta ten para xerar comandos que cumpran as condicións que especifiquemos; *:numtests* cambia o número de execucións a realizar; e por último *:max_size* cambia o tamaño máximo da cadea de comandos, xa que nos interesa que sexa tan exhaustiva como sexa posible.

Finalmente, na liña 7 se lle pasan estas opcións á ferramenta e na liña 8-12 defínese a propiedade; esta propiedade poder lerse como que, para tódolos comandos que existen no módulo, primeiro hai que iniciar o xogo, despois executar os comandos e por último facer unha limpeza do estado do xogo, todo iso sen que se produza ningún erro.

Á hora de definir a nosa propiedade, atopamos certas dificultades que ten `PropCheck` (e por ende, `PropEr`): tal e como acabamos de ver, as partidas do xogo empezan nas probas cun estado inicial aleatorio, debido que a asignación de roles é aleatoria, o que complica a validación dos diferentes escenarios ao non poder contar con esa información a priori. A ferramenta fai dúas execucións cando proba propiedades *stateful*: primeiro fai unha execución para crear a cadea de comandos a executar, e logo fai unha segunda execución utilizando eses comandos sobre o código:

1. Na primeira execución, crea un estado inicial e escolle un comando aleatorio da lista de comandos; se a postcondición (as condicións que se teñen que cumprir despois de executar un comando) de executar ese comando sobre o modelo da propiedade é correcta, a ferramenta escolle outro comando aleatoriamente e o executa, xerando así unha secuencia de comandos a utilizar.

2. Na segunda execución, `PropCheck` inicializa de novo o estado inicial de forma aleatoria

e executa a cadea de comandos que xerou sobre o código real e no modelo da propiedade á vez para comprobar as postcondicións de cada comando. Como os estados iniciais entre ambas execucións non teñen porque coincidir, xa que se crean con datos aleatorios, poden darse situación nas que a cadea de comandos que se xerou a partir do modelo non teña sentido no estado da execución real (por exemplo, na primeira execución o xogador "Abed" é un mafioso e un dos comandos que se xera nesta é que Abed selecciona, para matar, a outro xogador "Jeff"; agora ben, na segunda execución "Abed" non é un mafioso, polo que este comando fallará).

Para solucionar este problema, créase o estado inicial aleatorio antes da execución da proba e se inicializa tanto a propiedade, para que xere os comandos, como a partida co mesmo estado, de forma que tanto o modelo como a partida poidan aceptar a mesma cadea de comandos.

Outro problema co que nos atopamos na xeración da cadea de comandos é que a aleatoriedade desta dificulta que certas accións que poden ocorrer no xogo ocorran tamén nas probas (por exemplo, é moi improbable que saia un xogador acusado pola maioría). Para tratar con isto, `PropCheck` ofrece a posibilidade de modificar as frecuencias coas que os comandos son executados, permitindo facer máis frecuentes ou menos usuais cada un dos comandos. Porén, aínda que cambiemos, por exemplo, a frecuencia do comando que simula as acusacións dos xogadores a outro valor para que ocorra moito máis a miúdo, segue habendo o problema de que, nos comandos xerados, non se ten porque alcanzar a maioría de acusacións necesarias para que a acción continúe cun xuízo. Isto solucionouse engadindo un comando que xunte as acusacións de tódolos xogadores dirixidos á mesma persoa, dependendo desta maneira só de que se escolla o comando en non das acusacións individuais.

5.3.2 Exemplos de comandos

Ata agora explicouse a definición da propiedade coa que se proba a lóxica do xogo e os problemas que houbo que solucionar para poder facer unha máquina de estados con `PropCheck` do noso xogo. Porén, aínda non se explicou nin como funciona a máquina de estados, nin se mostraron exemplos dos comandos que esta utiliza.

Como xa se dixo anteriormente, `PropCheck` fai dúas execucións: a primeira sobre o modelo para xerar a cadea de comandos e a segunda sobre a aplicación coa cadea xerada. Na segunda execución, que é a importante para verificar o código, cando a ferramenta ten que executar un comando fai o seguinte:

1. Primeiro, comproba se a precondición do comando é certa, en caso de non selo a proba xa fallou.

2. En caso de que a postcondición sexa certa, executa o comando. Se por isto ocurre calquera tipo de erro, a proba falla.

3. Tras executar o comando, e no caso de non haber erros, comproba se a postcondición do comando é certa. Igual que coa precondición, se non é o caso a proba falla.

4. De non ser así, a execución continúa; a ferramenta garda o resultado de executar o comando no seu estado e pasa ao seguinte comando da cadea, co cal repite todo este proceso ata non haber máis comandos (ou fallar).

Os comandos que se executan pódense tamén dividir en función do estado do modelo. No noso caso, empregouse a etapa da partida (día, acusación, noite, etc) para decidir que comandos son válidos. En 5.5 móstrase como exemplo os comandos válidos para a fase de acusacións, que teñen diferentes frecuencias para favorecer as acusacións individuais (co comando *accuse/2*) sobre as acusacións grupais (co comando *all_accuse/2*).

```
1 def command(%{:phase => :accusation} = s) do
2     frequency([
3         {20, {:call, MafiaEngine.Game, :accuse, ["Testing",
    name(s), name(s)]}},
4         {5, {:call, __MODULE__, :all_accuse, [alive_players(s),
    name(s)]}},
5         {1, {:call, MafiaEngine.Game, :next_phase, ["Testing"]}}
6     ])
7 end
```

Listing 5.5: Definición dos comandos que se poden executar na fase de acusacións

A maiores, para cada comando hai que definir as precondicións e postcondicións. As precondicións úsanse para comprobar que o comando se pode executar co estado actual, como comprobar que o xogador que queremos acusar estea vivo. As postcondicións comproban que o resultado da execución do comando na partida produce os resultados esperados 5.6.

```
1 # precondicións para o comando acusar en grupo:
2 # 1. a fase ten que ser a de acusacións
3 # 2. o xogador a ser acusado ha de estar vivo
4 def precondition(s, {:call, _mod, :all_accuse, [_, accused]}) do
5     s.phase == :accusation
6     && s.players[accused].alive
7 end
8
9 # precondicións para votar inocente
10 # 1. a fase ten que ser a do xuízo
```

```
11 # 2. o que vota non pode ser o acusado
12 # 3. o que vota non votou xa
13 def precondition(s, {:call, _mod, :vote_innocent, [_, voter]}), do:
14     s.phase == :judgement
15     && s.players[voter].alive
16     && voter != s.accused
17     && not has_voted?(s, voter)
18 end
19
20 # postcondicións para acusar en grupo:
21 # 1. como a maioría acousou a unha persoa, a seguinte fase ten que
       ser a da defensa
22 # 2. o xogador acusado na partida é o esperado
23 def postcondition(_s, {:call, _mod, :all_accuse, [_, accused]},
       _res) do
24     game_phase() == :defense
25     && Map.get(game_votes(), :accused) == accused
26 end
27
28 # postcondición para votar inocente:
29 # 1. o xogador que votou está na lista de xogadores que votaron
       inocente
30 def postcondition(_s, {:call, _mod, :vote_innocent, [_, voter]},
       _res) do
31     voter in Map.get(game_votes(), :innocent)
32 end
33
34 # exemplo de claúsula para calquera outra postcondición
35 def postcondition(_s, {:call, _mod, _fun, _args}, _res), do: true
```

Listing 5.6: Exemplos de precondicións e postcondicións

Por último, os cambios que provocan os comandos no estado do modelo defínense como se mostra en 5.7. Desta forma a máquina pode manter un modelo do estado sen necesidade da execución e así poder usalo nas precondicións e postcondicións.

```
1 # o acabar a tarde, o seguinte estado é a noite
2 # a maiores hai que borrar as posibles seleccións que se gardaron
      no modelo
3 def next_state(%{:phase => :afternoon} = s, _res, {:call, _mod,
      :next_phase, _args}) do
4     set_phase(s, :night) |> clear_selections()
5 end
6
7 # o acabar a noite, o seguinte estado é a mañá
```

```
8  # tamén hai que comprobar se houbo asesinatos pola mafia e se
       acabou a partida por iso
9  def next_state(%{:phase => :night} = s, _res, {:call, _mod,
       :next_phase, _args}) do
10     mafia_kill(s) |> set_phase(:morning) |> check_end()
11 end
```

Listing 5.7: Exemplos de transicións entre os estados da máquina de estados

5.3.3 Resultados

Aínda que as probas baseadas en propiedades teñen unha complexidade maior á de outros métodos de facer probas máis tradicionais, como as probas de unidade ou as de integración, a cambio son moito máis robustas e teñen o beneficio engadido de que escribir propiedades obriga a entender a lóxica que queremos especificar (e, por tanto, implementar) cun maior nivel de coñecemento.

Grazas á súa incorporación neste proxecto, púidose verificar o comportamento correcto da lóxica do xogo cun nivel de confianza que non sería posible de alcanzar coas diferentes técnicas de *testing* que hai no proxecto, e sen necesidade de ter que escribir extensos módulos de probas para tentar garantir a ausencia de erros. Xa que as probas baseadas en propiedades xeran entradas aleatorias e de forma automática, executar estas probas comproba dunha forma moito máis exhaustiva o correcto funcionamento da aplicación.

A maiores, aínda que non se chegou a facer durante a realización deste traballo, este método de facer probas podería ser usado para facer probas de carga: definindo unha propiedade que xere o número de procesos a usar, e empregar esa cantidade para crear procesos que lancen a propiedade que xa se ensinou, a que proba en si a lóxica; desta forma, o resultado final sería que habería unha serie de números aleatorios de partidas executándose paralelamente, facendo o que sería unha proba de carga.

Capítulo 6

Conclusións

No derradeiro capítulo da memoria, falaremos sobre os obxectivos acadados, as leccións aprendidas durante o transcurso do traballo e o traballo futuro que queda por facer na aplicación.

6.1 Obxectivos

Para recapitular, os obxectivos que se propuxeron para este traballo eran:

- Primeiro, o desenvolvemento dunha aplicación web que adaptase unha versión modificada do xogo de dedución social Mafia.
- Segundo, garantir a escalabilidade da aplicación para acomodar a unha grande demanda de partidas de forma sinxela.
- Terceiro, e por último, asegurar que a aplicación, alén de estar intensivamente validada na súa lóxica, é tolerante a fallos e robusta, sendo capaz de retomar partidas sen perder o seu progreso.

O resultado final do traballo é unha aplicación desenvolvida en Elixir, que implementa a lóxica do xogo, e unha aplicación web feita usando co marco de desenvolvemento da linguaxe, Phoenix. A aplicación resultante adapta as regras de Mafia e facilita a creación de partidas para xogar cun grupo de coñecidos, de forma sinxela e sen obstáculos. A cal leva despregada dende abril de 2020 en Gigalixir e pode atoparse en

`https://mafia-game.gigalixirapp.com`

Debido ás tecnoloxías que se usaron neste traballo, a aplicación é capaz de escalar con facilidade para ofrecer o seu servizo a unha gran cantidade de usuarios sen esforzo [28] e *out of the box*, coa posibilidade de ampliarse requirindo cambios mínimos.

Por último, grazas as ferramentas que achega Elixir na súa librería estándar, crear procesos que supervisan o estado da aplicación e garanten o retorno da súa dispoñibilidade en caso de

erros acadouse con facilidade. Grazas ó cada día máis amplo ecosistema que ten a linguaxe, púidose realizar a validación dunha lóxica non trivial aproveitando as ferramentas existentes especializadas en métodos de probas avanzados, e acadouse aproveitando a capacidade que en particular achegan as probas baseadas en propiedades de asegurar unha alta confianza no comportamento correcto do código, xa que permiten unha verificación do código mais exhaustiva que os métodos de *testing* máis tradicionais.

6.2 Leccións aprendidas

A realización deste traballo permitiume aplicar e afianzar certas leccións e coñecementos aprendidos nas materias do grao. Destacan, entre outras, o prototipado e creación da interface da aplicación web (cfg. 3.3) que aprendín en *Interfaces Persoa Máquina*, ou os conceptos de *Arquitectura do Software* que guiaron o deseño e implementación da arquitectura da aplicación (cfg. 3.2). As bases aprendidas na materia de *Concorrencia e Paralelismo* foron tamén de grande utilidade, pola natureza da linguaxe coa que se traballou e o tipo de aplicación que se fixo, e ter afondado e aprendido métodos máis potentes para facer probas en *Validación e Verificación do Software* axudaron a facer probas baseadas en propiedades (cfg. 5.3) no proxecto.

A maiores, realizar este traballo axudoume a afianzar varias competencias do título, das que cabe destacar as seguintes:

- A25 — Capacidade para desenvolver, manter e avaliar servizos e sistemas software que satisfagan todos os requisitos do usuario e se comporten de forma fiable e eficiente, sexan accesibles de desenvolver e manter, e cumpran normas de calidade, aplicando as teorías, principios, métodos e prácticas da enxeñaría do software.

- A28 — Capacidade de identificar e analizar problemas, e deseñar, desenvolver, implementar, verificar e documentar solucións software sobre a base dun coñecemento adecuado das teorías, modelos e técnicas actuais.

- A57 — Capacidade de concibir sistemas, aplicacións e servizos baseados en tecnoloxías de rede en que se inclúan internet, web, comercio electrónico, multimedia, servizos interactivos e computación móbil.

Por último, realizar un proxecto que estende os límites da miña mención permitiume afrontar os problemas dende un punto de vista diferente e aumentar a miña perspectiva no mundo da enxeñaría informática.

6.3 Traballo Futuro

Como posible traballo futuro a realizar na aplicación poderíamos destacar:

- A realización de probas de carga, a través das probas baseadas en propiedades como se falou anteriormente no capítulo de probas (cfg. 5.3.3).
- A adición de novos roles ao xogo, para conseguir máis diversidade nas partidas tendo en conta as diferentes versións comentadas no capítulo de contexto (cfg. 2.1.4).
- Localizar o xogo, para que soporte máis linguas, empregando *gettext*, xa que ten soporte nativo en Phoenix [29].
- A adición dunha guía de xogo na páxina principal.

Bibliografía

[1] Salario medio dos Enxeñeiros do Software en España (2021). [En liña]. Dispoñible en: https://www.glassdoor.es/Sueldos/ingeniero-de-software-sueldo-SRCH_KO0,21.htm

[2] GitHub. [En liña]. Dispoñible en: https://github.com/

[3] Gigalixir. [En liña]. Dispoñible en: https://gigalixir.com/

[4] Mafia en BoardGameGeek. [En liña]. Dispoñible en: https://boardgamegeek.com/boardgame/925/werewolf

[5] Hombres Lobo de Castronegro. [En liña]. Dispoñible en: https://www.zygomatic-games.com/es/juegos/los-hombres-lobo-de-castronegro/

[6] Bang! [En liña]. Dispoñible en: https://bang.dvgiochi.com/prod.php?id=1

[7] Secret Hitler. [En liña]. Dispoñible en: https://www.secrethitler.com/

[8] Mafia Scum. [En liña]. Dispoñible en: https://www.mafiascum.net/

[9] Mafia.gg. [En liña]. Dispoñible en: https://mafia.gg/

[10] Town of Salem. [En liña]. Dispoñible en: https://www.blankmediagames.com/

[11] Among Us. [En liña]. Dispoñible en: https://www.innersloth.com/gameAmongUs.php

[12] Elixir. [En liña]. Dispoñible en: https://elixir-lang.org/

[13] Erlang. [En liña]. Dispoñible en: https://www.erlang.org/

[14] Casos de uso de Elixir. [En liña]. Dispoñible en: https://elixir-lang.org/cases.html

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Pearson Education, 1994.

[16] Real time communication at scale with Elixir at Discord. [En liña]. Dispoñible en: https://elixir-lang.org/blog/2020/10/08/real-time-communication-at-scale-with-elixir-at-discord/

[17] PaaS with Elixir at Heroku. [En liña]. Dispoñible en: https://elixir-lang.org/blog/2020/09/24/paas-with-elixir-at-Heroku/

[18] Why Pinterest just open-sourced new tools for the Elixir programming language. [En liña]. Dispoñible en: https://venturebeat.com/2015/12/18/pinterest-elixir/

[19] Marketing and sales intelligence with Elixir at PepsiCo. [En liña]. Dispoñible en: https://elixir-lang.org/blog/2021/04/02/marketing-and-sales-intelligence-with-elixir-at-pepsico/

[20] Phoenix Framework. [En liña]. Dispoñible en: https://www.phoenixframework.org/

[21] Ecto (repositorio). [En liña]. Dispoñible en: https://github.com/elixir-ecto/ecto

[22] Liveview (repositorio). [En liña]. Dispoñible en: https://github.com/phoenixframework/phoenix_live_view

[23] F. Hebert, *Property-Based Testing with PropEr, Erlang, and Elixir.* Pragmatic Bookshelf, 2019. [En liña]. Dispoñible en: https://propertesting.com/

[24] Erlang term storage (documentación). [En liña]. Dispoñible en: https://github.com/phoenixframework/phoenix_live_view

[25] E. A. Manolis Papadakis and K. Sagonas. PropEr (repositorio). [En liña]. Dispoñible en: https://github.com/proper-testing/proper

[26] K. Alfert. PropCheck (repositorio). [En liña]. Dispoñible en: https://github.com/alfert/propcheck

[27] A. Leopardi. Streamdata (repositorio). [En liña]. Dispoñible en: https://github.com/whatyouhide/stream_data

[28] The Road to 2 Million Websocket Connections in Phoenix. [En liña]. Dispoñible en: https://www.phoenixframework.org/blog/the-road-to-2-million-websocket-connections

[29] Internacionalización con Phoenix LiveView . [En liña]. Dispoñible en: https://www.paulfioravanti.com/blog/internationalisation-phoenix-liveview/