The Mage Compendium

MAS - Design and Analysis of Information Systems

Daniil Brusnikau – s24109

Contents

User Requirements

The presented system, the “Mage Compendium”, or simply “The Compendium” is a world-spanning Magic management system for a fictional reality. It tackles various aspects related to Magic, such as:

- Mages, or people capable of using the Magic around them to cast Spells.
- Schools of Magic, where Mages go to improve their abilities and obtain knowledge.
- Arcane Domains – the various categories of Magic, that define Spell functions and their karmic energy.
- Spells – the application of Magic in the form of a castable incantation, harmful and helpful both.
- Benefactors – entities that sign Partnerships with Mages for achieving various goals, from research work to warfare.

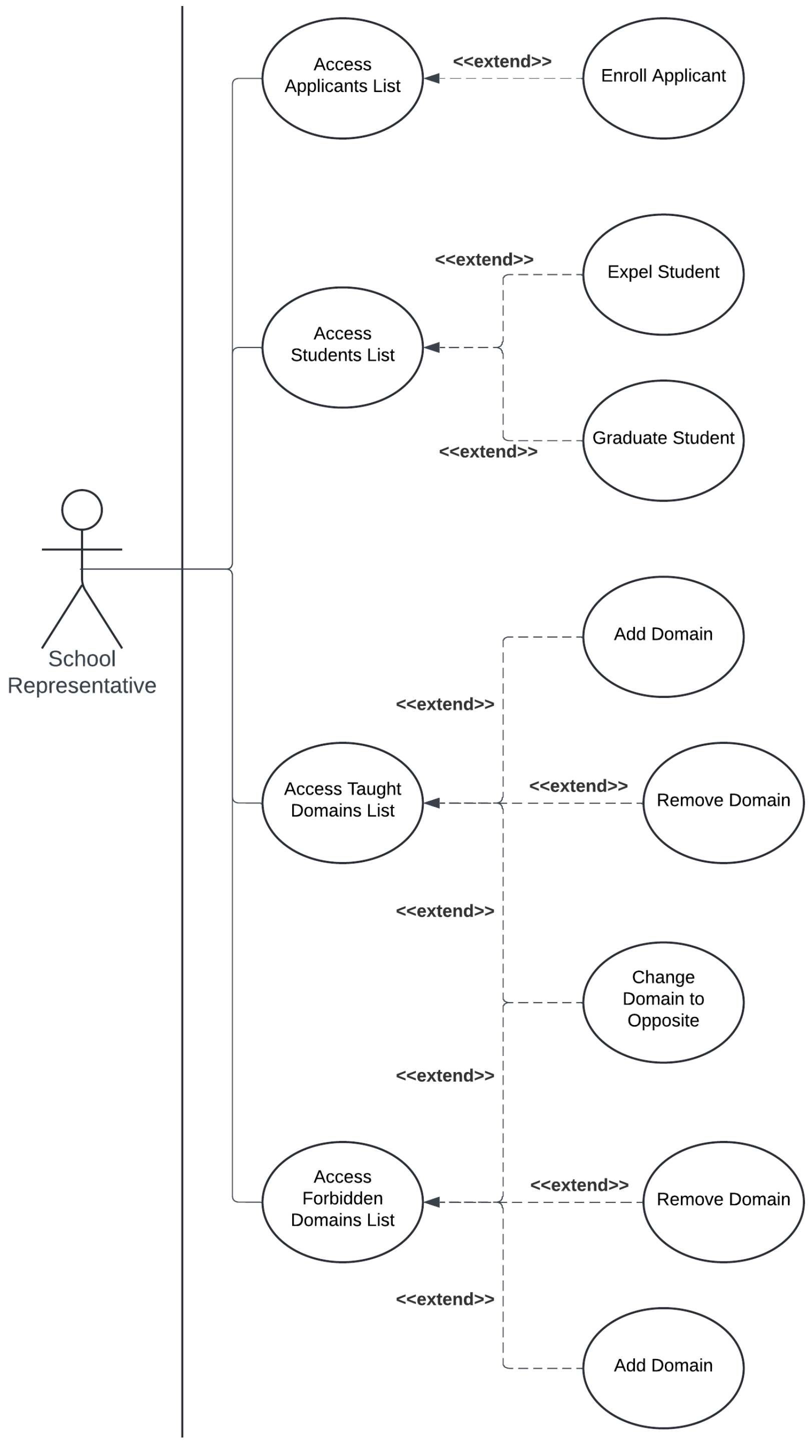
The system is divided into various modules encapsulating different functionality. A module is chosen by the user depending on what role they want to take on, and their options include:

- School Representative, which is a module dedicated to managing the systems of an Arcane School, like enrolling or expelling students, and managing the taught and forbidden domains within them.
- Student, a small module for managing a Mage’s student status within a particular school.
- Benefactor, a module for negotiating and managing Partnerships with Mages.
- Mage, the central module dedicated to connecting the person to the other modules, like applying for Partnerships, learning Spells and improving their understanding of known ones, and applying to Schools.
- Lord of Magic, which is a special module reserved for the all-powerful entity in charge of the Compendium, that lets them create and destroy Spells and Arcane Domains.

The application has a GUI implementation allowing for easy access to the functionality, particularly to the Benefactor module, and the related aspects of the Mage module.

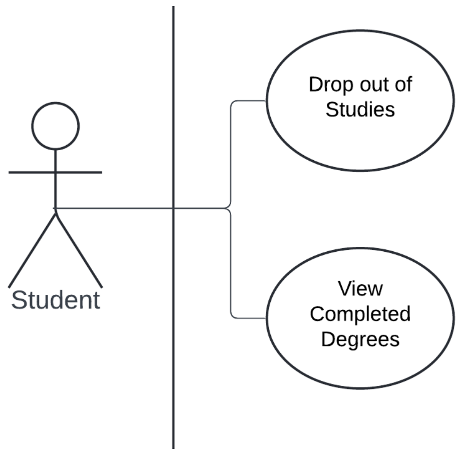
Use Case Diagrams

Use Case Diagram – School Representative



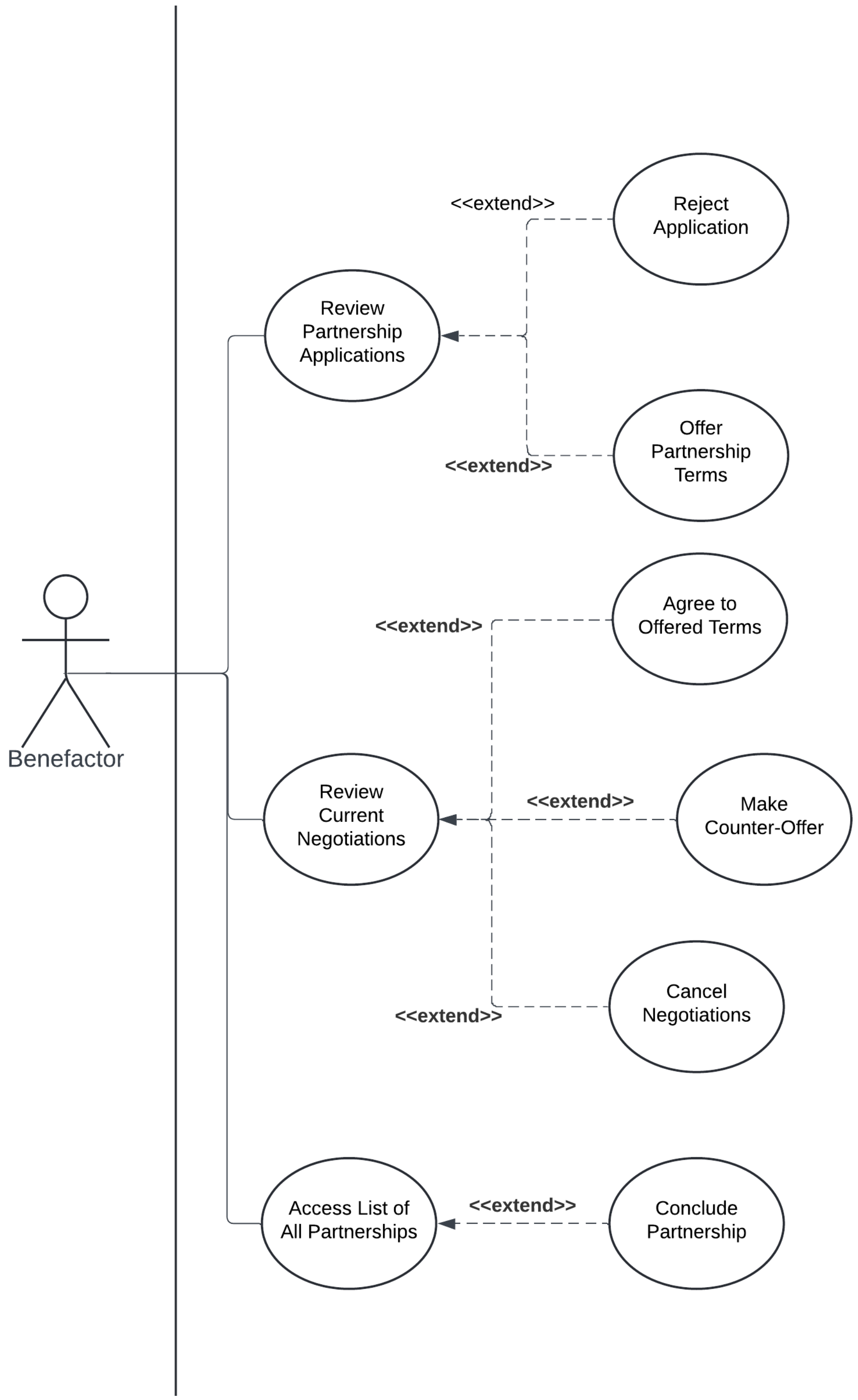
The School Representative diagram displays the various functionalities relating to managing Arcane Schools, like managing applicants, students, or the domains that are taught or forbidden at the establishment.

Use Case Diagram – Student



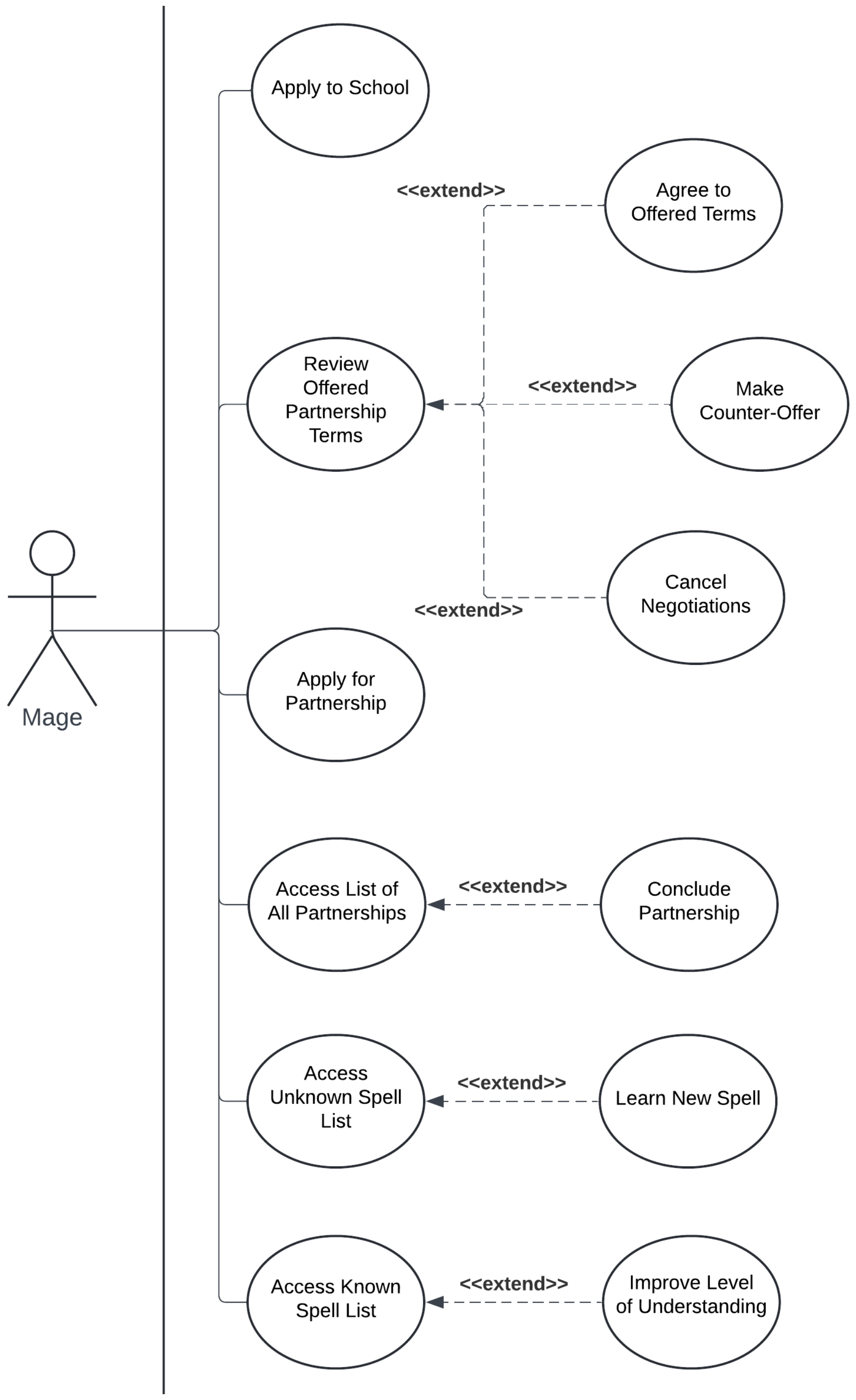
The small module covers basic student functionality, like viewing their past degrees at the Arcane School, as well as dropping out the current degree they are taking part in.

Use Case Diagram – Benefactor



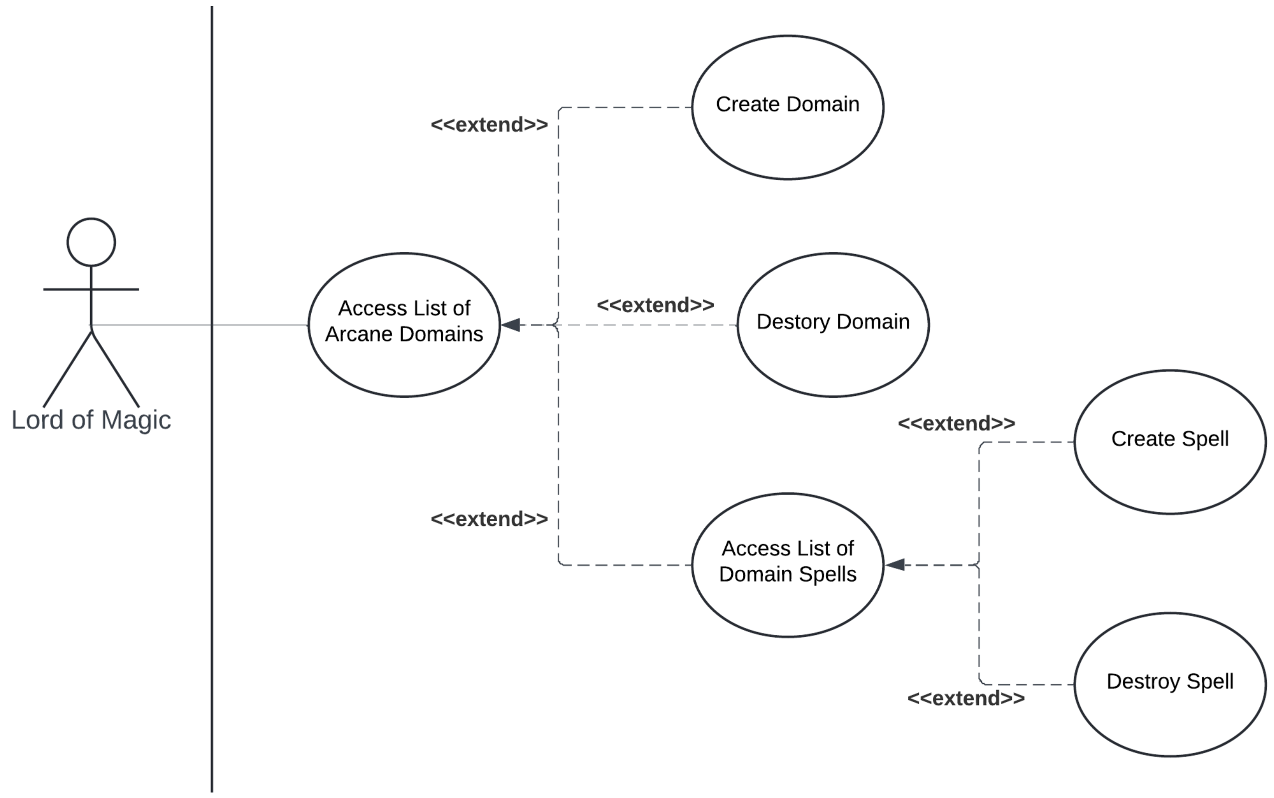
The Benefactor Use Case diagram outlines the functionalities of the module, like managing applications for partnerships, including refusal, counter-offering and possible eventual agreement, as well as the conclusion of ongoing partnerships.

Use Case Diagram – Mage



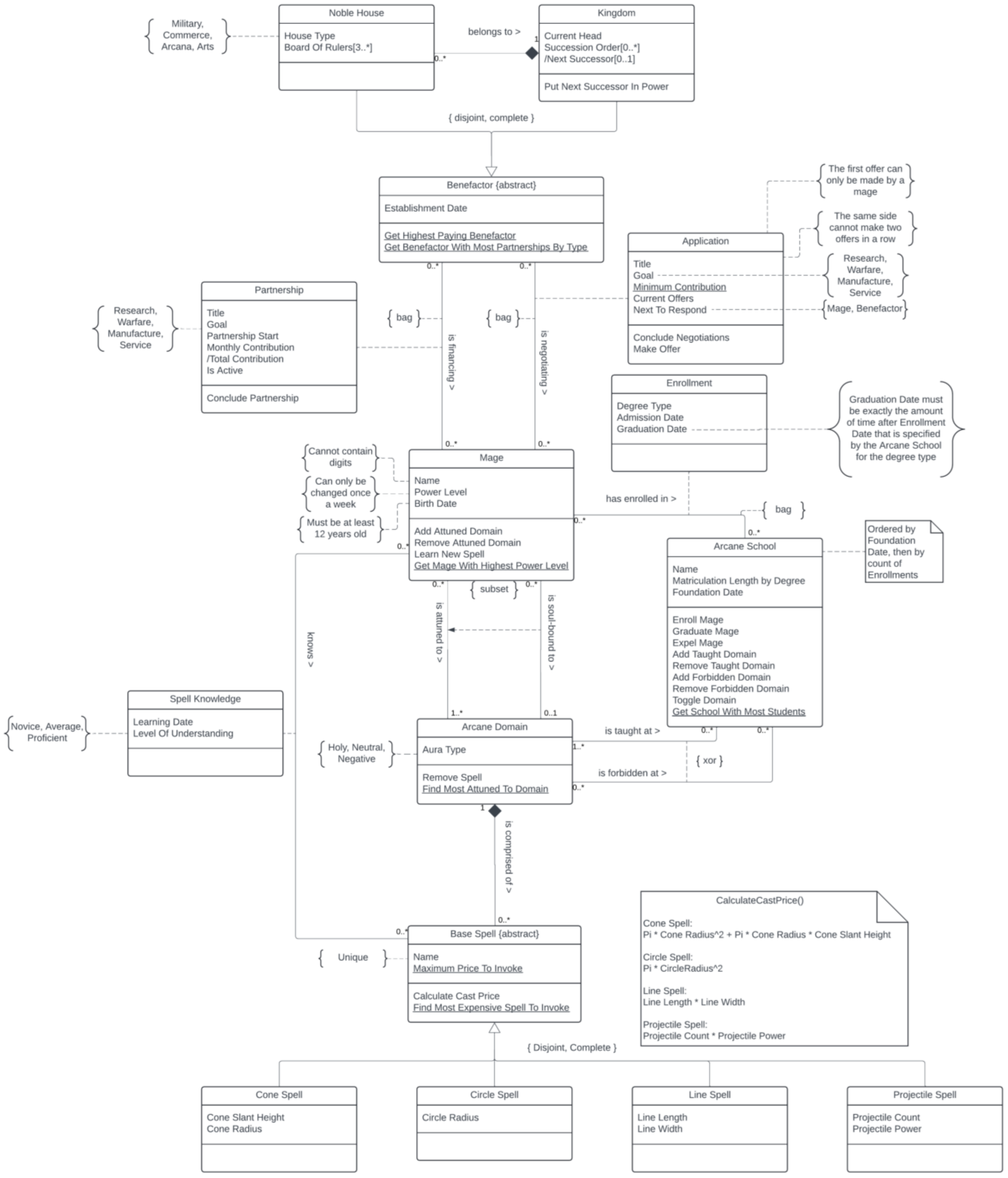
The Mage module handles various communication with the other modules, like learning and improving the knowledge of various Spells, handling Application negotiations and Partnerships with Benefactors, and applying to Arcane Schools.

Use Case Diagram – Lord of Magic



The last module handles the special functionality offered to the entity managing the Magic of the world, known as the Lord of Magic. They can create and destroy Arcane Domains, as well as create and destroy Spells within them.

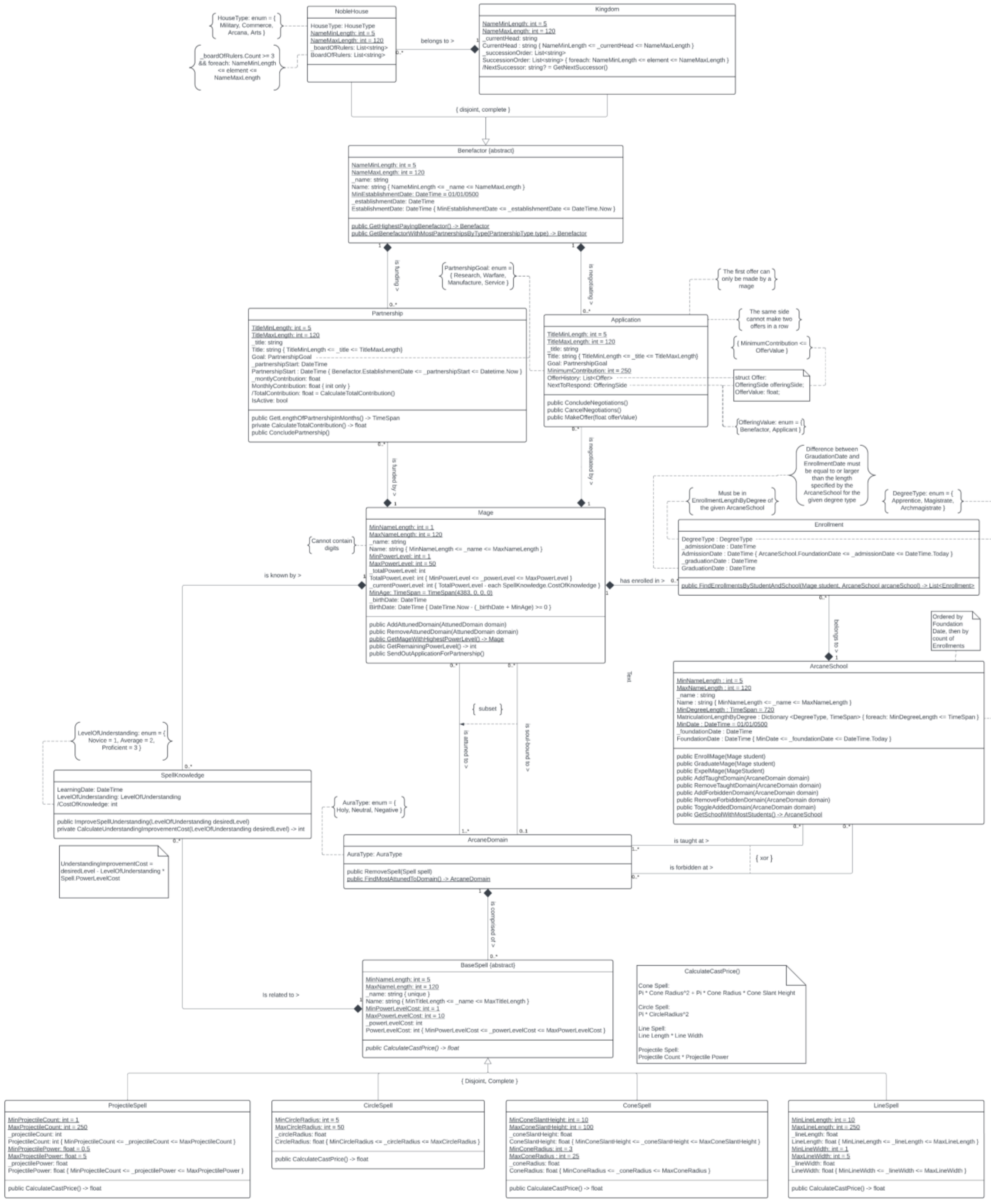
Class Diagram – Analytical



The analytical class diagram displays the general way the classes involved should function. The module split can also be noticed here based on the region of the diagram that certain classes are present in:

- Base Spell, as well as the four inheritors, and Arcane Domain handle Magic and the functionalities of the Lord of Magic. Spells are also related to Mages, who can learn them. Arcane Domains are either taught or forbidden at Arcane Schools.
- Arcane School and Enrollment handles the academic aspect of the system, storing the data on degrees taken by Mages, as well as their current state, and the Domains and their state at given Schools.
- Benefactor, Noble House, Kingdom, Partnership and Application all handle the financial part of the system, with Mages being capable of entering Partnerships with Benefactors after undergoing negotiations stored within the Application class.
- Finally, the Mage class itself is at the heart of the system and is heavily interconnected with the other classes.

Class Diagram – Design


HouseType: enum = { Military, Commerce, Arcana, Arts }
_boardOfRulers.Count >= 3 && foreach: NameMinLength <= element <= NameMaxLength
NobleHouse
HouseType: HouseType
NameMinLength: int = 5
NameMaxLength: int = 120
_boardOfRulers: List<string>
BoardOfRulers: List<string>
belongs to >
0..*
1
Kingdom
NameMinLength: int = 5
NameMaxLength: int = 120
_currentHead: string
CurrentHead : string { NameMinLength <= _currentHead <= NameMaxLength }
_successionOrder: List<string>
SuccessionOrder: List<string> { foreach: NameMinLength <= element <= NameMaxLength }
/NextSuccessor: string? = GetNextSuccessor()
{ disjoint, complete }
Benefactor {abstract}
NameMinLength: int = 5
NameMaxLength: int = 120
_name: string
Name: string { NameMinLength <= _name <= NameMaxLength }
MinEstablishmentDate: DateTime = 01/01/0500
_establishmentDate: DateTime
EstablishmentDate: DateTime { MinEstablishmentDate <= _establishmentDate <= DateTime.Now }
public GetHighestPayingBenefactor() -> Benefactor
public GetBenefactorWithMostPartnershipsByType(PartnershipType type) -> Benefactor
1
is funding >
0..*
PartnershipGoal: enum = { Research, Warfare, Manufacture, Service }
1
is negotiating >
0..*
The first offer can only be made by a mage
The same side cannot make two offers in a row
{ MinimumContribution <= OfferValue }
Partnership
TitleMinLength: int = 5
TitleMaxLength: int = 120
_title: string
Title: string { TitleMinLength <= _title <= TitleMaxLength}
Goal: PartnershipGoal
_partnershipStart: DateTime
PartnershipStart : DateTime { Benefactor.EstablishmentDate <= _partnershipStart <= Datetime.Now }
_montlyContribution: float
MonthlyContribution: float { init only }
/TotalContribution: float = CalculateTotalContribution()
IsActive: bool
public GetLengthOfPartnershipInMonths() -> TimeSpan
private CalculateTotalContribution() -> float
public ConcludePartnership()
Application
TitleMinLength: int = 5
TitleMaxLength: int = 120
_title: string
Title: string { TitleMinLength <= _title <= TitleMaxLength}
Goal: PartnershipGoal
MinimumContribution: int = 250
OfferHistory: List<Offer>
NextToRespond: OfferingSide
public ConcludeNegotiations()
public CancelNegotiations()
public MakeOffer(float offerValue)
struct Offer:
OfferingSide offeringSide;
OfferValue: float;
OfferingValue: enum = { Benefactor, Applicant }
0..*
is funded by >
1
0..*
is negotiated by >
1
Difference between GraudationDate and EnrollmentDate must be equal to or larger than the length specified by the ArcaneSchool for the given degree type
Must be in EnrollmentLengthByDegree of the given ArcaneSchool
DegreeType: enum = { Apprentice, Magistrate, Archmagistrate }
Cannot contain digits
Mage
MinNameLength: int = 1
MaxNameLength: int = 120
_name: string
Name: string { MinNameLength <= _name <= MaxNameLength }
MinPowerLevel: int = 1
MaxPowerLevel: int = 50
_totalPowerLevel: int
TotalPowerLevel: int { MinPowerLevel <= _powerLevel <= MaxPowerLevel }
_currentPowerLevel: int { TotalPowerLevel - each SpellKnowledge.CostOfKnowledge }
MinAge: TimeSpan = TimeSpan(4383, 0, 0, 0)
_birthDate: DateTime
BirthDate: DateTime { DateTime.Now - (_birthDate + MinAge) >= 0 }
public AddAttunedDomain(AttunedDomain domain)
public RemoveAttunedDomain(AttunedDomain domain)
public GetMageWithHighestPowerLevel() -> Mage
public GetRemainingPowerLevel() -> int
public SendOutApplicationForPartnership()
is known by >
1
has enrolled in >
1
0..*
Enrollment
DegreeType : DegreeType
_admissionDate : DateTime
AdmissionDate : DateTime { ArcaneSchool.FoundationDate <= _admissionDate <= DateTime.Today }
_graduationDate : DateTime
GraduationDate : DateTime
public FindEnrollmentsByStudentAndSchool(Mage student, ArcaneSchool arcaneSchool) -> List<Enrollment>
0..*
belongs to >
1
Ordered by Foundation Date, then by count of Enrollments
Text
ArcaneSchool
MinNameLength : int = 5
MaxNameLength : int = 120
_name : string
Name : string { MinNameLength <= _name <= MaxNameLength }
MinDegreeLength : TimeSpan = 720
MatriculationLengthByDegree : Dictionary <DegreeType, TimeSpan> { foreach: MinDegreeLength <= TimeSpan }
MinDate : DateTime = 01/01/0500
_foundationDate : DateTime
FoundationDate : DateTime { MinDate <= _foundationDate <= DateTime.Today }
public EnrollMage(Mage student)
public GraduateMage(Mage student)
public ExpelMage(MageStudent)
public AddTaughtDomain(ArcaneDomain domain)
public RemoveTaughtDomain(ArcaneDomain domain)
public AddForbiddenDomain(ArcaneDomain domain)
public RemoveForbiddenDomain(ArcaneDomain domain)
public ToggleAddedDomain(ArcaneDomain domain)
public GetSchoolWithMostStudents() -> ArcaneSchool
0..*
0..*
{ subset }
is attuned to >
is soul-bound to >
LevelOfUnderstanding: enum = { Novice = 1, Average = 2, Proficient = 3 }
0..*
SpellKnowledge
LearningDate: DateTime
LevelOfUnderstanding: LevelOfUnderstanding
/CostOfKnowledge: int
public ImproveSpellUnderstanding(LevelOfUnderstanding desiredLevel)
private CalculateUnderstandingImprovementCost(LevelOfUnderstanding desiredLevel) -> int
0..*
UnderstandingImprovementCost = desiredLevel - LevelOfUnderstanding * Spell.PowerLevelCost
AuraType: enum = { Holy, Neutral, Negative }
1..*
0..1
ArcaneDomain
AuraType: AuraType
public RemoveSpell(Spell spell)
public FindMostAttunedToDomain() -> ArcaneDomain
is taught at >
1..*
0..*
{ xor }
is forbidden at >
0..*
0..*
1
is comprised of >
0..*
Is related to >
1
BaseSpell {abstract}
MinNameLength: int = 5
MaxNameLength: int = 120
_name: string { unique }
Name: string { MinTitleLength <= _name <= MaxTitleLength }
MinPowerLevelCost: int = 1
MaxPowerLevelCost: int = 10
_powerLevelCost: int
PowerLevelCost: int { MinPowerLevelCost <= _powerLevelCost <= MaxPowerLevelCost }
public CalculateCastPrice() -> float
CalculateCastPrice()
Cone Spell:
Pi * Cone Radius^2 + Pi * Cone Radius * Cone Slant Height
Circle Spell:
Pi * CircleRadius^2
Line Spell:
Line Length * Line Width
Projectile Spell:
Projectile Count * Projectile Power
{ Disjoint, Complete }
ProjectileSpell
MinProjectileCount: int = 1
MaxProjectileCount: int = 250
_projectileCount: int
ProjectileCount: int { MinProjectileCount <= _projectileCount <= MaxProjectileCount }
MinProjectilePower: float = 0.5
MaxProjectilePower: float = 5
_projectilePower: float
ProjectilePower: float { MinProjectileCount <= _projectilePower <= MaxProjectilePower }
public CalculateCastPrice() -> float
CircleSpell
MinCircleRadius: int = 5
MaxCircleRadius: int = 50
_circleRadius: float
CircleRadius: float { MinCircleRadius <= _circleRadius <= MaxCircleRadius }
public CalculateCastPrice() -> float
ConeSpell
MinConeSlantHeight: int = 10
MaxConeSlantHeight: int = 100
_coneSlantHeight: float
ConeSlantHeight: float { MinConeSlantHeight <= _coneSlantHeight <= MaxConeSlantHeight }
MinConeRadius: int = 3
MaxConeRadius : int = 25
_coneRadius: float
ConeRadius: float { MinConeRadius <= _coneRadius <= MaxConeRadius }
public CalculateCastPrice() -> float
LineSpell
MinLineLength: int = 10
MaxLineLength: int = 250
_lineLength: float
LineLength: float { MinLineLength <= _lineLength <= MaxLineLength }
MinLineWidth: int = 1
MaxLineWidth: int = 5
_lineWidth: float
LineWidth: float { MinLineWidth <= _lineWidth <= MaxLineWidth }
public CalculateCastPrice() -> float

The Design diagram follows the material presented on the Analytical diagram while converting the syntax to the one used in C#, as well as imposing additional constraints on element length, contents, and more. Additionally, it converts things like many-to-many associations to programmatically possible concepts, like utilizing in-between association classes.

Use Case Scenario – Apply for Partnership

Name: Apply for Partnership

Actor: Mage

Precondition: The Actor is a verified Mage.

Basic Path:

1. The Actor starts the use case.
2. The System displays a panel containing fields for the application data.
3. The Actor fills in the data and submits the form.
4. The System validates the data and creates the application.
5. The System closes the use case.

Alternative Paths:

3a. The Actor clicks on one of the buttons outside the panel.

1. The System closes the use case.

3b. The Actor submits an empty or partially filled form.

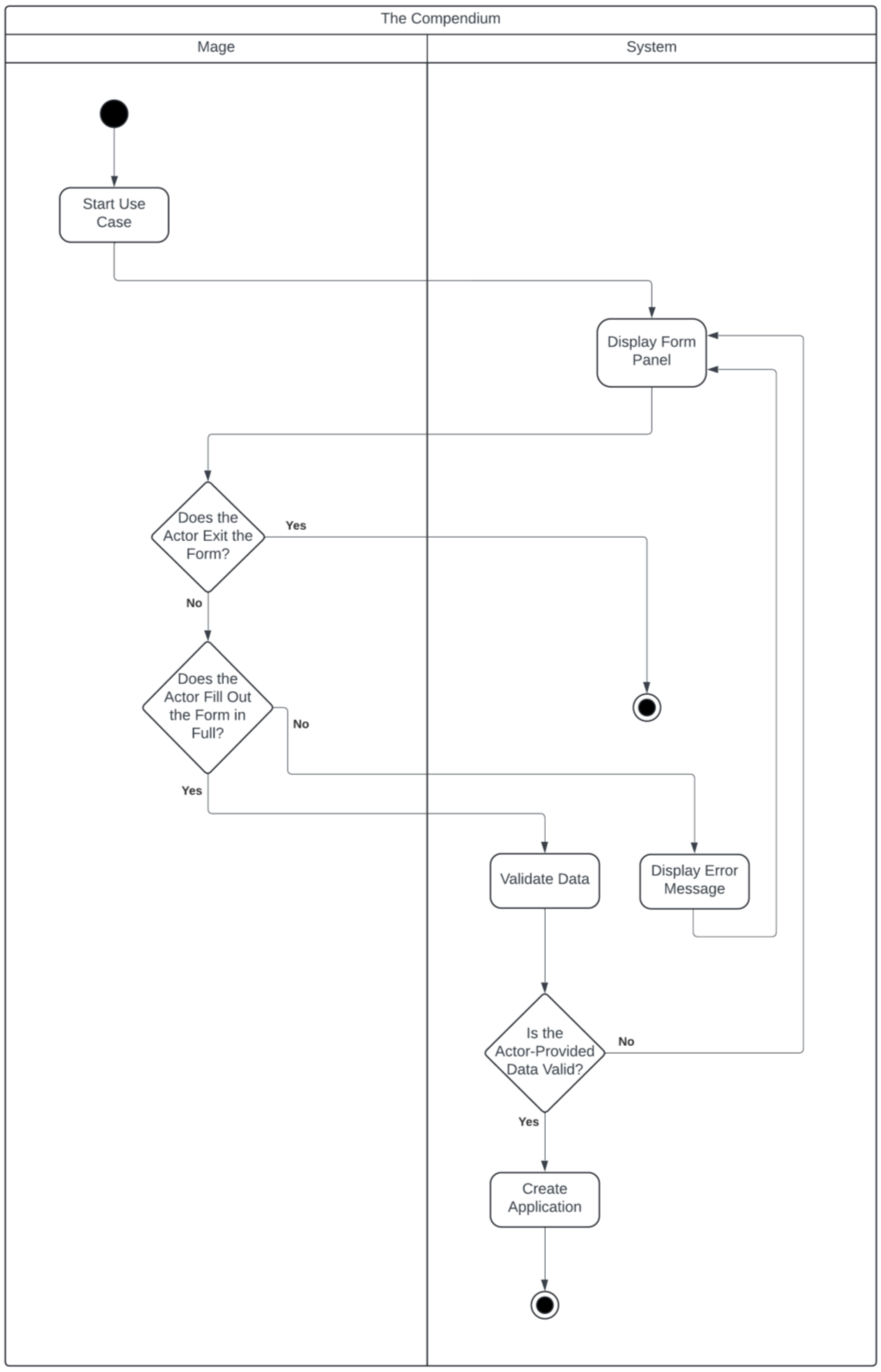
1. The System displays an error message asking to fill out the whole form.
2. The System goes to step 2

4a. The data provided by the Actor is incorrect.

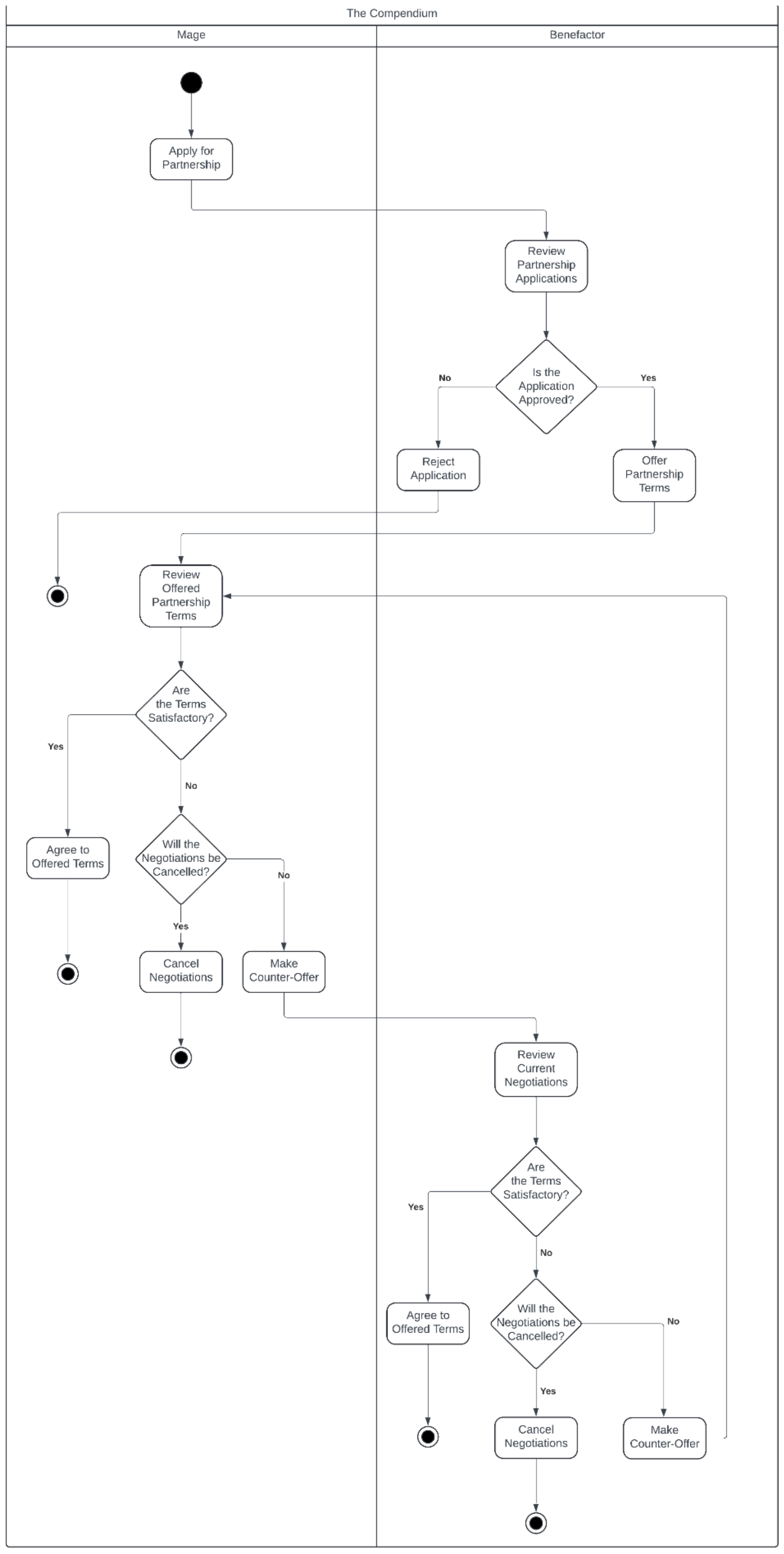
1. The System displays an error message asking to correct the mistakes.
2. The System goes to step 2.

Postcondition: the new application is successfully added to the database and the GUI is updated to reflect the new addition.

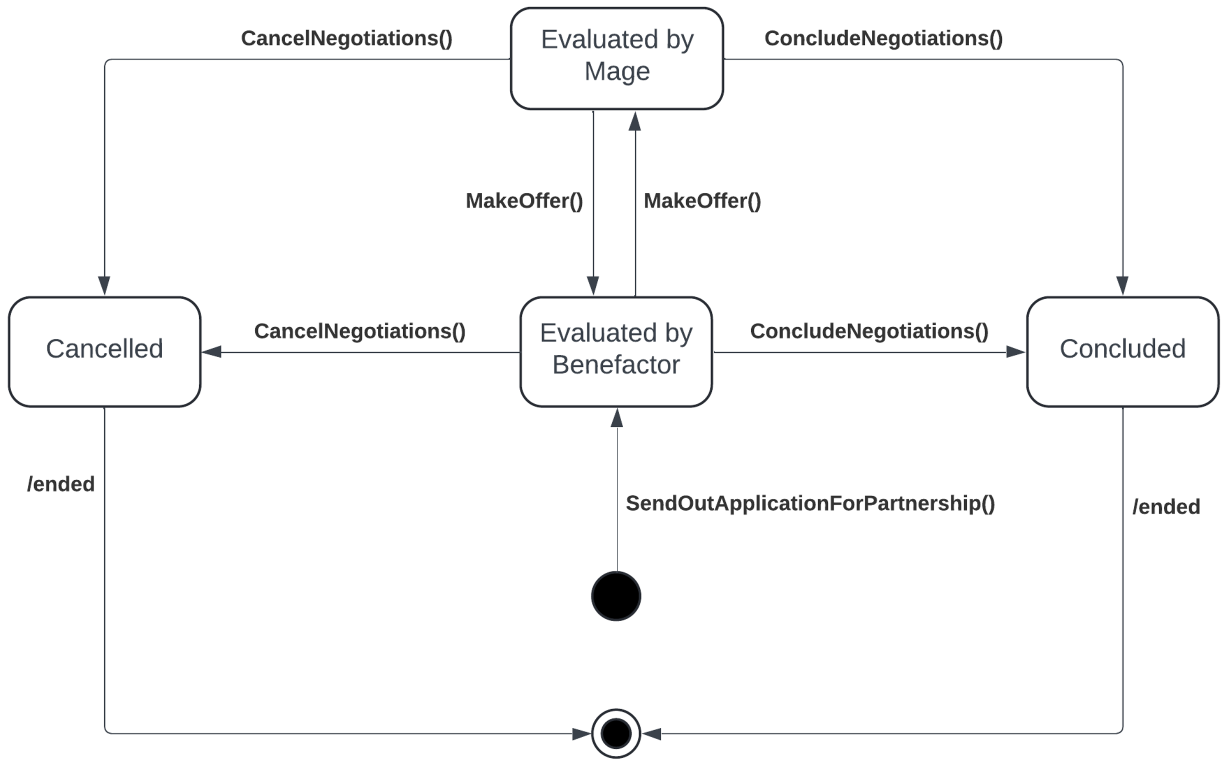
Activity Diagram – Apply for Partnership



Activity Diagram – Application to Conclusion


The Compendium
Mage
Benefactor
Apply for Partnership
Review Partnership Applications
Is the Application Approved?
No
Yes
Reject Application
Offer Partnership Terms
Review Offered Partnership Terms
Are the Terms Satisfactory?
Yes
No
Agree to Offered Terms
Will the Negotiations be Cancelled?
No
Yes
Cancel Negotiations
Make Counter-Offer
Review Current Negotiations
Are the Terms Satisfactory?
Yes
No
Agree to Offered Terms
Will the Negotiations be Cancelled?
No
Yes
Cancel Negotiations
Make Counter-Offer

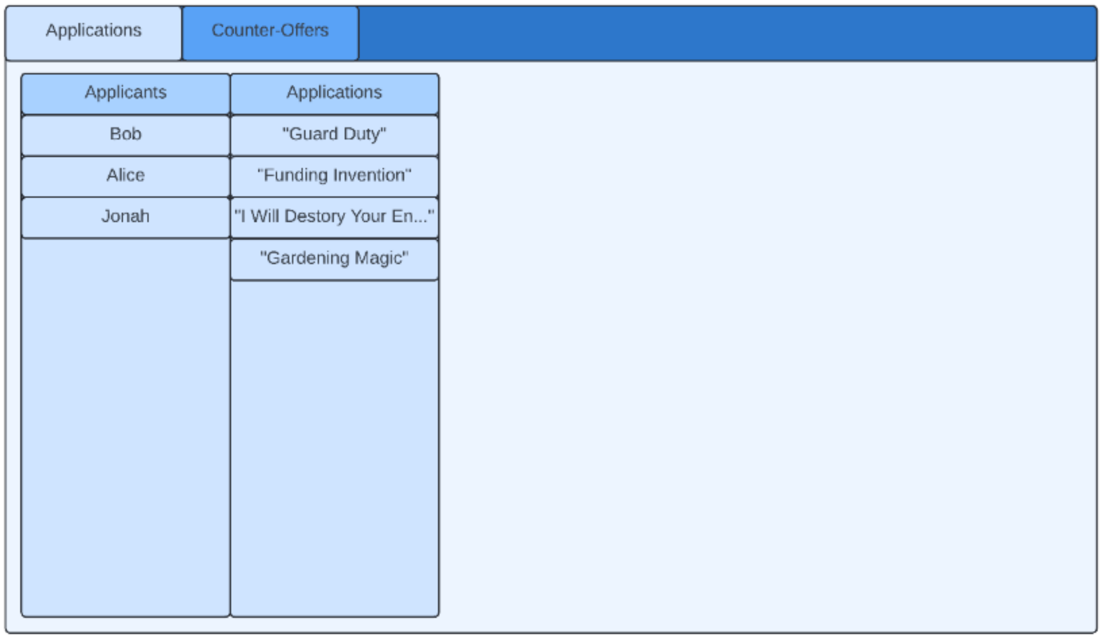
State Diagram – Application

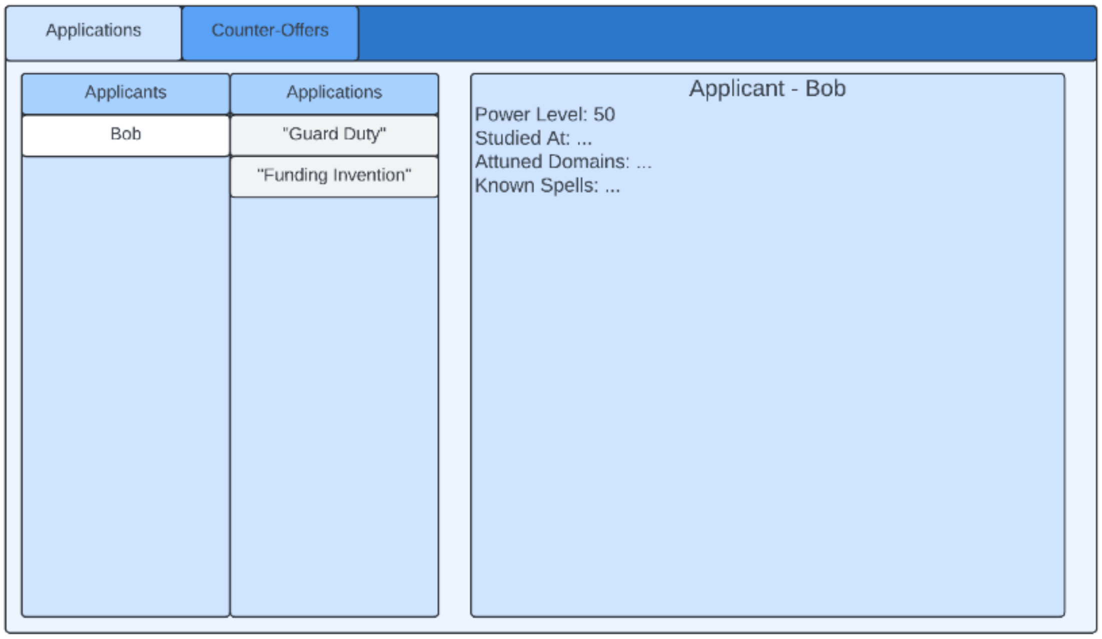


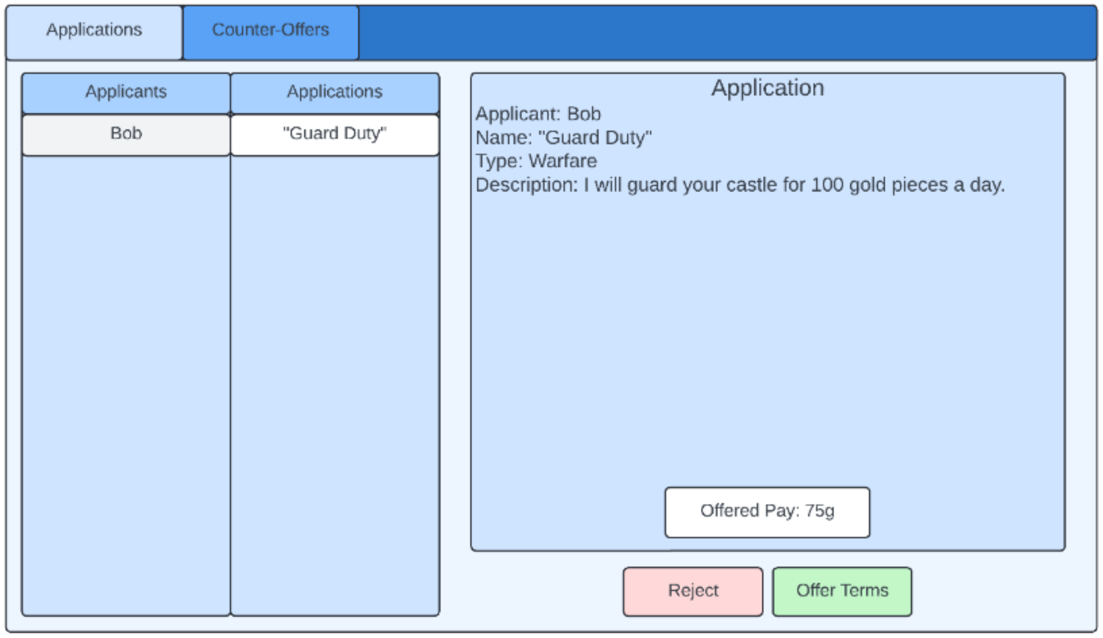
The State diagram shows the various states that the Application class can find itself in over the course of its lifetime. When first created following a Mage sending out an Application for Partnership, the class is the “evaluated” by the associated Benefactor, who can either reject it, agree to the terms, or make a counter-offer, in which case the Application is now “evaluated” by the original sender, and can find itself in the same states as with the Benefactor. Upon either being cancelled or concluded, the Application’s contents are transferred to a Partnership object, and it is destroyed.

GUI Design

GUI Design – Benefactor



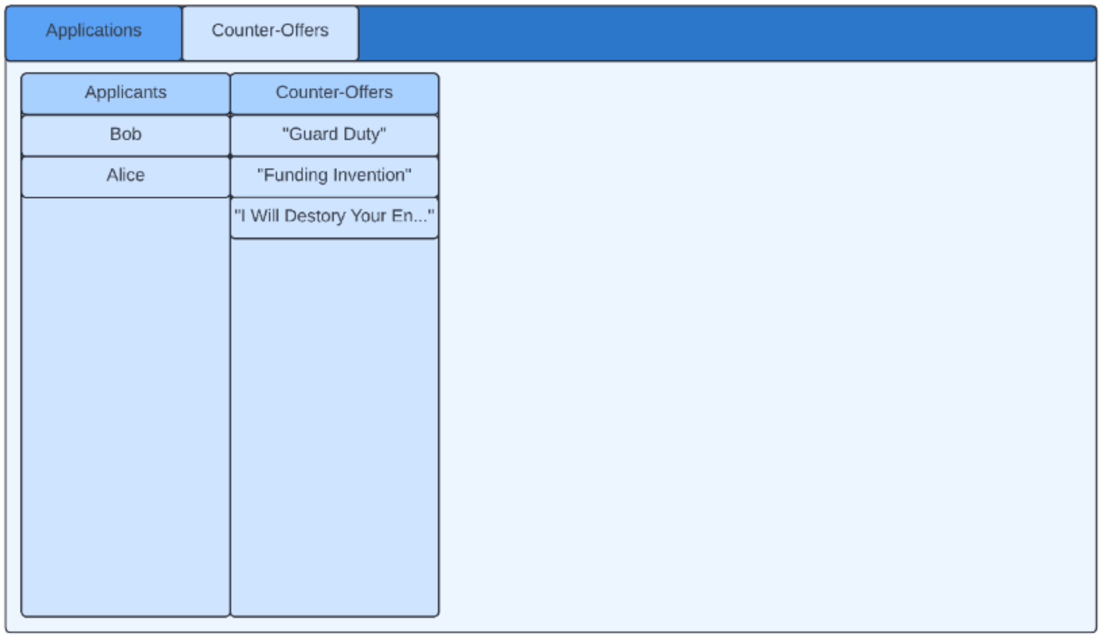


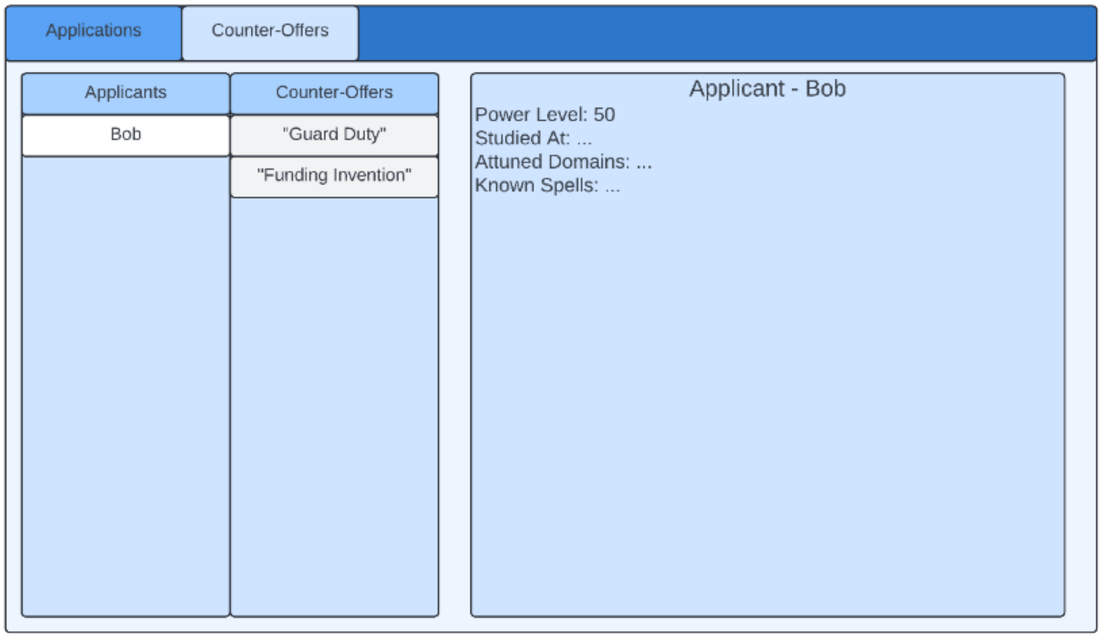


Application

Applicant: Bob
Name: "Guard Duty"
Type: Warfare
Description: I will guard your castle for 100 gold pieces a day.

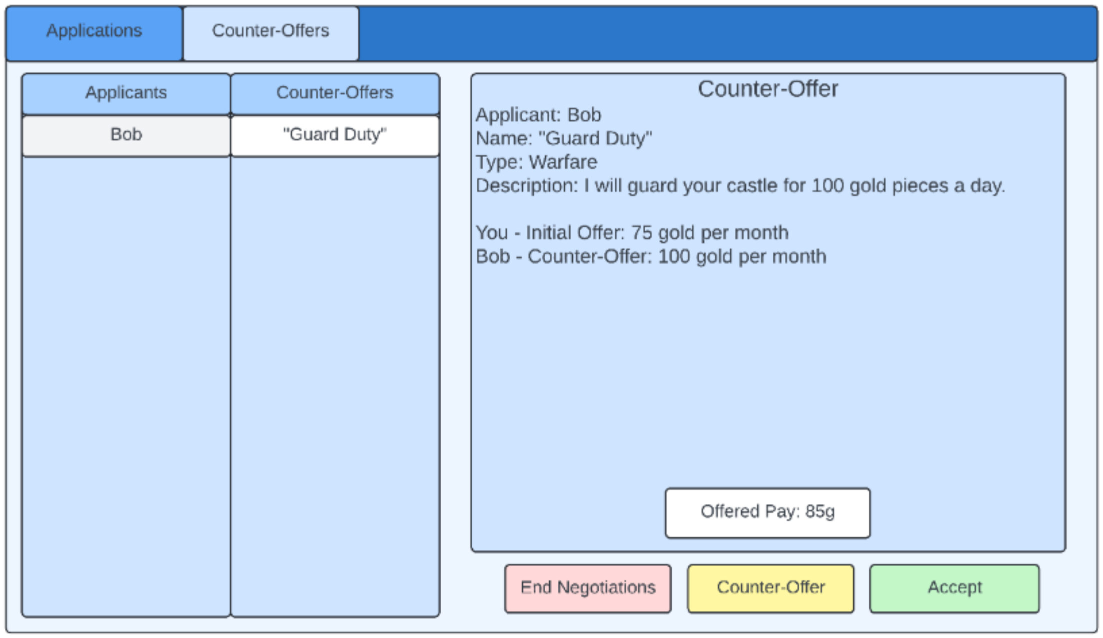
Offered Pay: 75g





Applicant - Bob

Power Level: 50
Studied At: ...
Attuned Domains: ...
Known Spells: ...



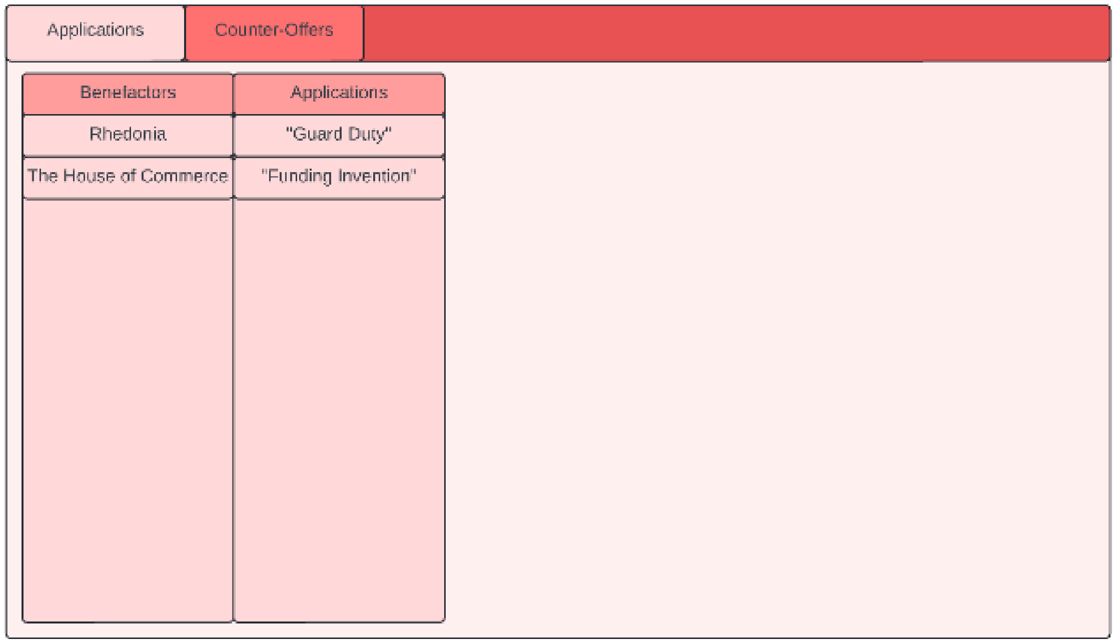
Counter-Offer

Applicant: Bob
Name: "Guard Duty"
Type: Warfare
Description: I will guard your castle for 100 gold pieces a day.

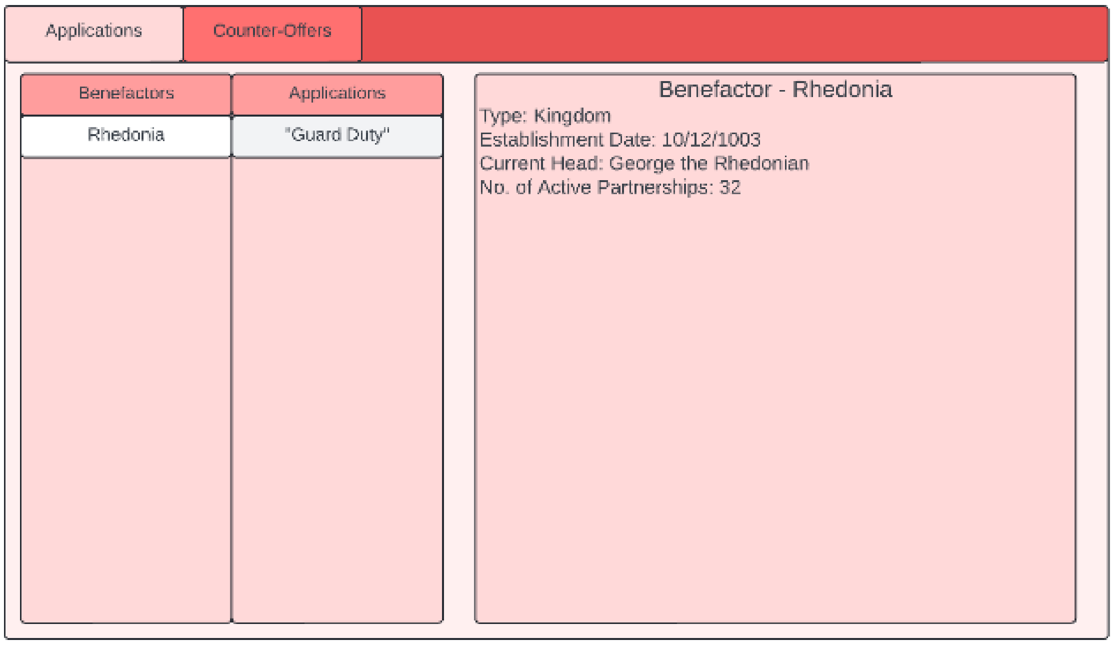
You - Initial Offer: 75 gold per month
Bob - Counter-Offer: 100 gold per month

Offered Pay: 85g

GUI Design – Mage

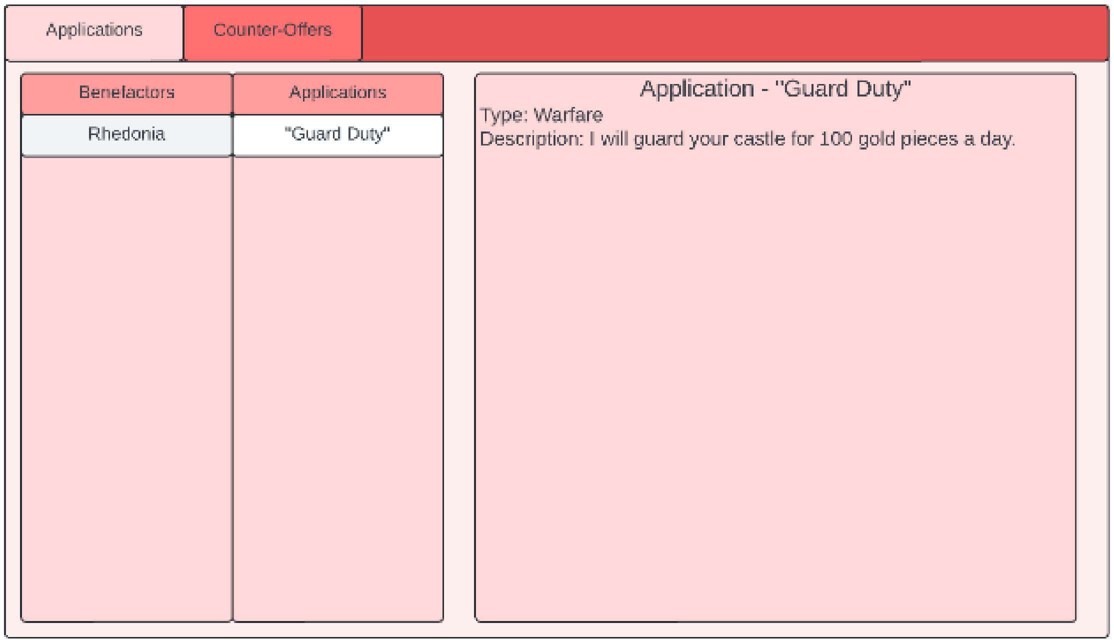


Benefactors	Applications
Rhedonia	"Guard Duty"
The House of Commerce	"Funding Invention"



Benefactors	Applications
Rhedonia	"Guard Duty"

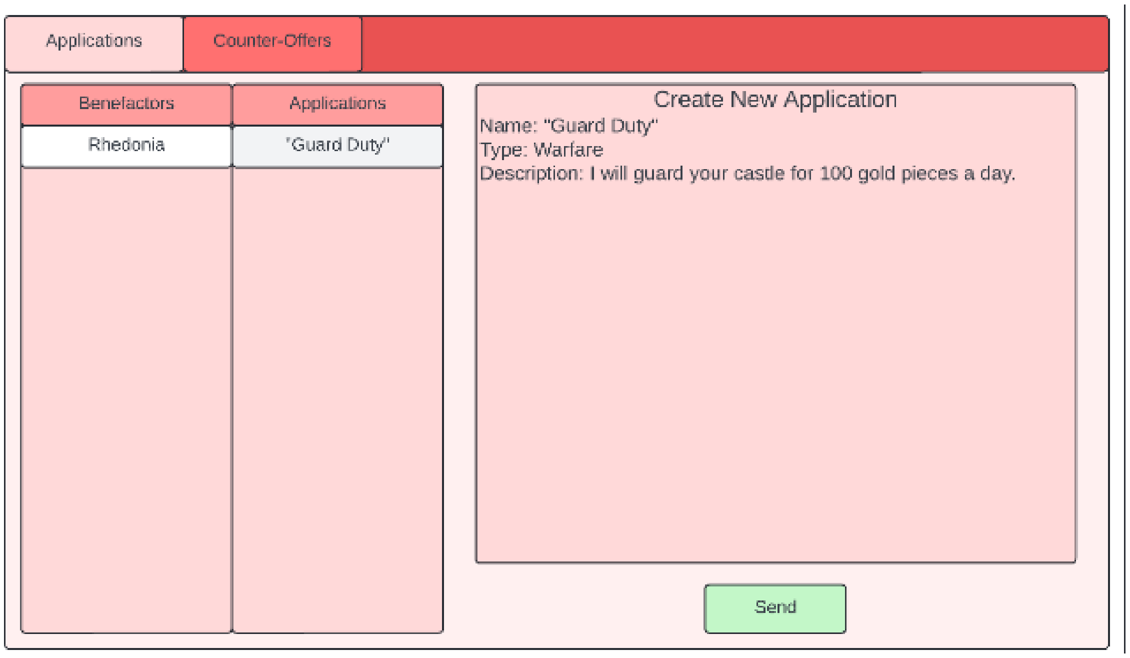
Benefactor - Rhedonia
Type: Kingdom
Establishment Date: 10/12/1003
Current Head: George the Rhedonian
No. of Active Partnerships: 32



Application - "Guard Duty"

Type: Warfare

Description: I will guard your castle for 100 gold pieces a day.

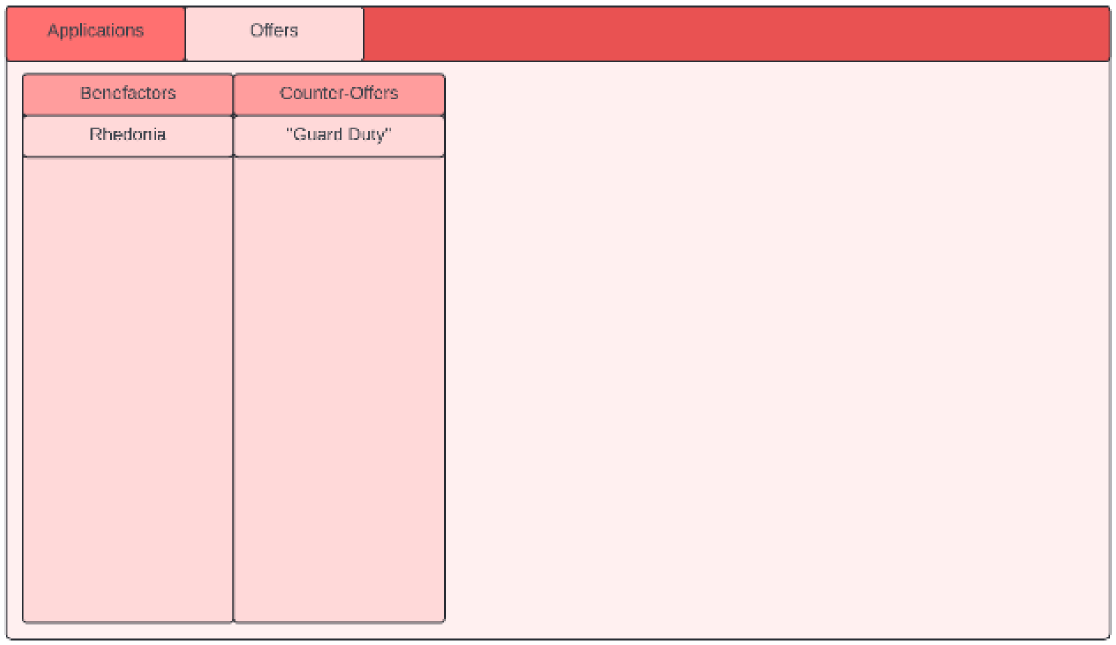


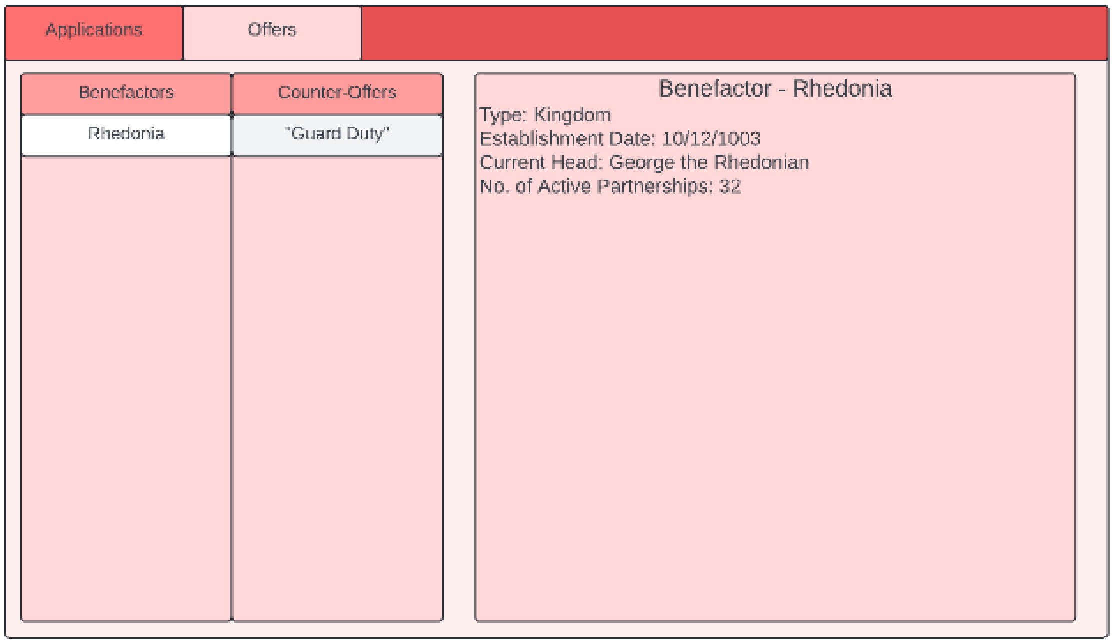
Create New Application

Name: "Guard Duty"

Type: Warfare

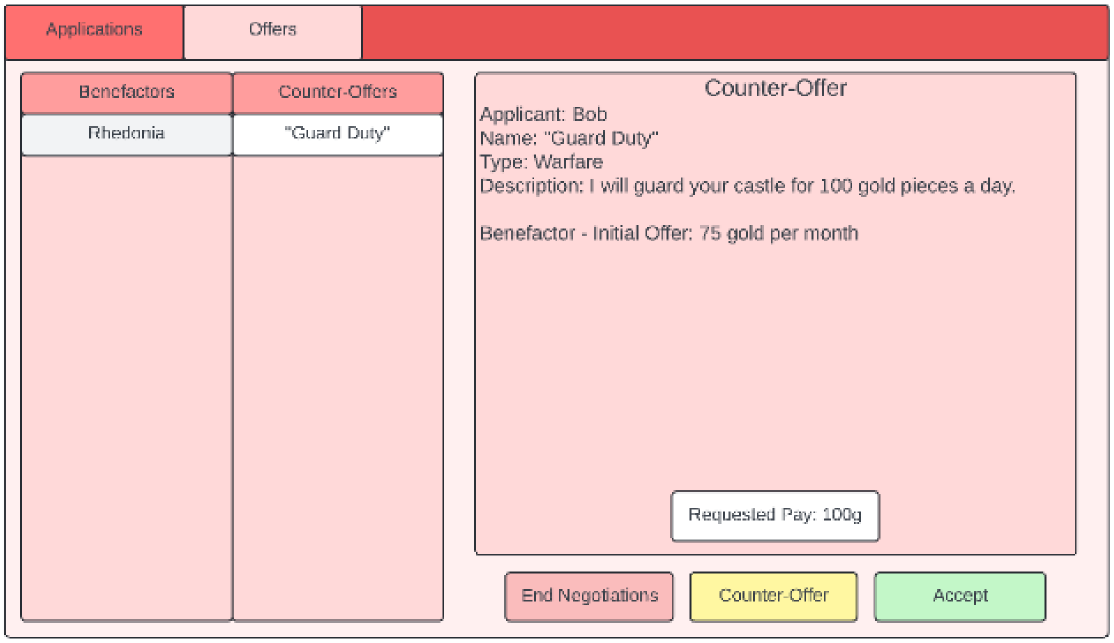
Description: I will guard your castle for 100 gold pieces a day.





Benefactor - Rhedonia

Type: Kingdom
Establishment Date: 10/12/1003
Current Head: George the Rhedonian
No. of Active Partnerships: 32

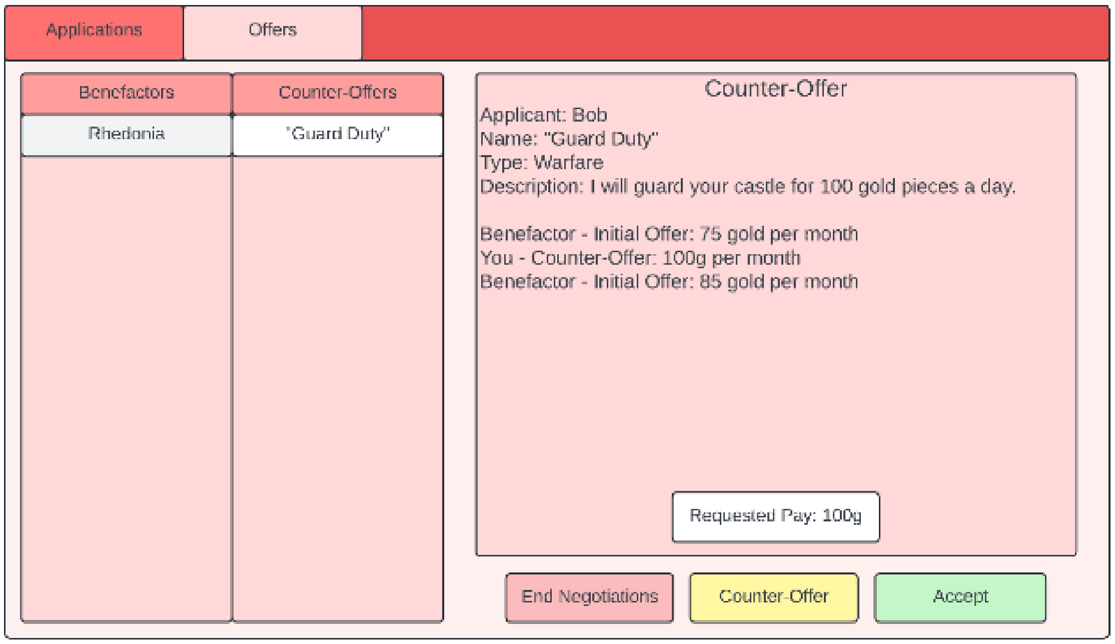


Counter-Offer

Applicant: Bob
Name: "Guard Duty"
Type: Warfare
Description: I will guard your castle for 100 gold pieces a day.

Benefactor - Initial Offer: 75 gold per month

Requested Pay: 100g



Counter-Offer

Applicant: Bob
Name: "Guard Duty"
Type: Warfare
Description: I will guard your castle for 100 gold pieces a day.

Benefactor - Initial Offer: 75 gold per month
You - Counter-Offer: 100g per month
Benefactor - Initial Offer: 85 gold per month

Requested Pay: 100g

Design Decisions Discussion

The program will be implemented in C#, using EFCore for the database side of the system, and Unity to build the GUI aspect. The data will be stored in a persistent SQLite database on the user's computer.

Every class represents a model within the system. Its' methods will be implemented using repositories within EFCore. The constraints will be implemented both on the model level in getters and setters, and on the database level using configurations of the model builder. The extents will be handled by DbSets in EFCore. Basic CRUD functionality will be handled by them as well. Derived attributes will be handled as simple getter properties returning a function or calling a method.

Composition associations will be handled using cascade deletes within EFCore. Inheritance will follow the TPC pattern. XOR and Subset validation will be handled on the model level. Many-to-many associations will be handled through creating intermediate association classes to convert them to two one-to-many associations instead.