CPSC 2720 [FALL 2021]

Wrexorcist

Team Wrex

Brayden Carlson, Angela De Sousa Costa, Rocky Au, Juhyoung Park

November 6, 2021

TABLE OF CONTENTS

INTRODUCTION

In this text-based adventure game you will take the role of a young exorcist in training. One night on your travels, you fall asleep in the woods and wake up in a strange world. A mysterious figure approaches you and informs you that there are three trapped souls who are keeping you here and it is your duty to release them. As you continue your adventure, you realize that there is more to this world than you think, and it is your duty to investigate what is truly going on. The actions that you take will have an impact on whether or not you escape.

The player will be able to explore this mysterious world and interact with their environment. The player will examine the environment and enemies to discover clues and items to aid them on their adventure. There are also puzzles throughout the environment for unlockable items in order to get the best ending. Other locations include areas for the player to rest at and others that provide useful functions. There are also enemies present in this world, however, the player does not always need to fight the enemy, and in fact, may be able to befriend some of them. The player will need to figure out how to obtain the right items to appease some of these enemies (namely, the trapped souls). The player must be careful though, as some items may make the trapped souls permanently hostile. The player will also be able to find equipment and consumables should they desire to take the more combative route and the game will feature the possibility to fight the enemies.

The environment will be initialized as a 4 x 4 area to explore; however it will be randomly generated. There will still be a natural progression to the exploration and enemies encountered, however the environment will not always be the same with each play through. The areas will be themed after the specific trapped boss in that area. This adds to the replayability of the game. There will be a variety of rooms throughout the game. Some rooms will contain generic enemies and items. There will also be utility rooms, such as for healing and teleporting. There will also be rooms with puzzles and traps that the player must solve to navigate. The items the players need will also be found in the rooms. Items include consumable items (health potions), equipment, and key items to exorcise the trapped souls.

The player completes the game if they are able to fulfill the conditions to exit the world. The final boss is the mysterious figure at the start of the game. The player must defeat the final boss before exiting the mysterious world. The game is only beatable if they do not kill every trapped soul. If they do, the final boss will become undefeatable, and the player will eventually run out of health. The player may also run out of health via combat or failing certain puzzles/traps enough times. If the player kills some of the trapped souls, but frees the others, the game goes towards a neutral end. If the player releases all of the souls, the player achieves the good ending, provided they defeat the final boss. The trapped souls can be set free by reducing their health to a certain amount and then using an item from their past on them. This will pacify the trapped soul and they will move on to the after life.

PROJECT MANAGEMENT

TEAM ROLES

Team Member	**Design - Draft**	**Design – Final**	**Implementation - Basic**	**Implementation - Final**
Rocky	Phase Lead	Design Lead	QA Lead	Reporting Lead
Brayden	Design Lead	QA Lead	Reporting Lead	Phase Lead
Angela	QA Lead	Reporting Lead	Phase Lead	Design Lead
Juhyoung	Reporting Lead	Phase Lead	Design Lead	QA Lead

TEAM ROLE RESPONSIBILITIES

Phase Lead

- Keeps team on track during meetings.
- Has a good understanding of the phase requirements.
- If necessary, seeks clarification from the instructor.
- Ensures that the team's tasks are completed on time.
- Helps to set sub-goals for the phase in consultation with the group.
- Makes sure that all members are aware of their responsibilities and duties.
- Ensures that all team members are heard and have a voice in discussions.
- Ensures that tasks are divided evenly and makes adjustments as needed.

Design Lead

- Suggests design ideas to the group and collects them.
- Changes design if necessary.
- Ensures the design is being followed with the proper SOLID+DRY principles.
- Modifies the UML diagram and sequence diagram to reflect the current state of the software.

Quality Assurance Lead

- Ensures that errors haven't occurred.
- Issues are high quality.
    - Bug reports use the proper bug report template.
- Fixes any style or static errors.
- Implements tests are ensures they are working.
- Establishes document contribution deadlines.

Reporting Lead

- Writes the team reports, along with the team members.
- Ensures that all team members know what's expected for the team report.
- Sets deadlines for team report.
- Ensures bugs are in the process of getting fixed.
    - If not, they fix it themselves or remind the team members about it.

RISK MANAGEMENT

Requirements/Design/Estimation

1. The design didn't work out as intended.
    - We will meet together and decided how we can move forward and how we can change certain things to make the design work.
2. There must be major changes to the design when implementing the code.
    - We will meet together and work out what each person has to do so it can be a smooth and fast change. We could also see if there's a work around.
3. The team forgot to add a key requirement of the group project.
    - We will get together and discuss how we can implement that requirement into our design and do the necessary changes.
4. The team planned a project that is too large.
    - If this happens, we will get together and decide how we can narrow the project down so it's not as large and equally distribute the work needed.
5. The team underestimated how long parts of the project would take.
    - We will meet as a group and communicate how we would be able to get this done. If needed, we can assign two people to work on the long part, especially if it's an essential part of the project.

People

1. If a team member isn't meeting deadlines
    - The group will talk to the person and ask why they aren't meeting the deadlines
        - If the person is loaded with work or had an emergency, the group will come up with a plan; the 3 members will take on what that person was supposed to do, and that person will be reassigned.
        - If they don't respond and go M.I.A, we will try to ask them in class. If the behaviour doesn't change, we will meet with Dr. Anvik and let him know what that team member is doing.
2. If a team member leaves
    - We will get together as a group and discuss how we can equally distribute the work
    - If a member is okay with taking on more work, they will be allowed to, as long as they keep open communication and let the team know if they aren't able to do it.
3. If a team member doesn't have the technical skills for that specific task
    - They will get reassigned to a different task
    - They will be asked to watch another member of the team to gain those technical skills

Learning & Tools

1. Lack of understanding of lab tools
    - If the tool is taught in the lab, but no one in the group has a clear understanding of it, we will make an appointment with Nicole to ask for clarification
2. Tools don't work together
    - We will discuss on which tool is more important and find alternatives for the other tool

DEVEVLOPMENT PROCESS

CODE REVIEW PROCESS

- If any team member has a proposal to change the code design, they will email whoever is the phase leader, and the phase leader will bring it up in the next meeting.
  - We will see if this change will be easy and/or necessary.
- Any changes that are made to the design plan, without consultation of the group, we will have a discussion in a group meeting.
- The Quality Assurance Lead will make sure that there are no conflicts in the code before committing it to the master branch and/or will reject a merge request if it doesn't follow our design plan.

COMMUNICATION TOOLS

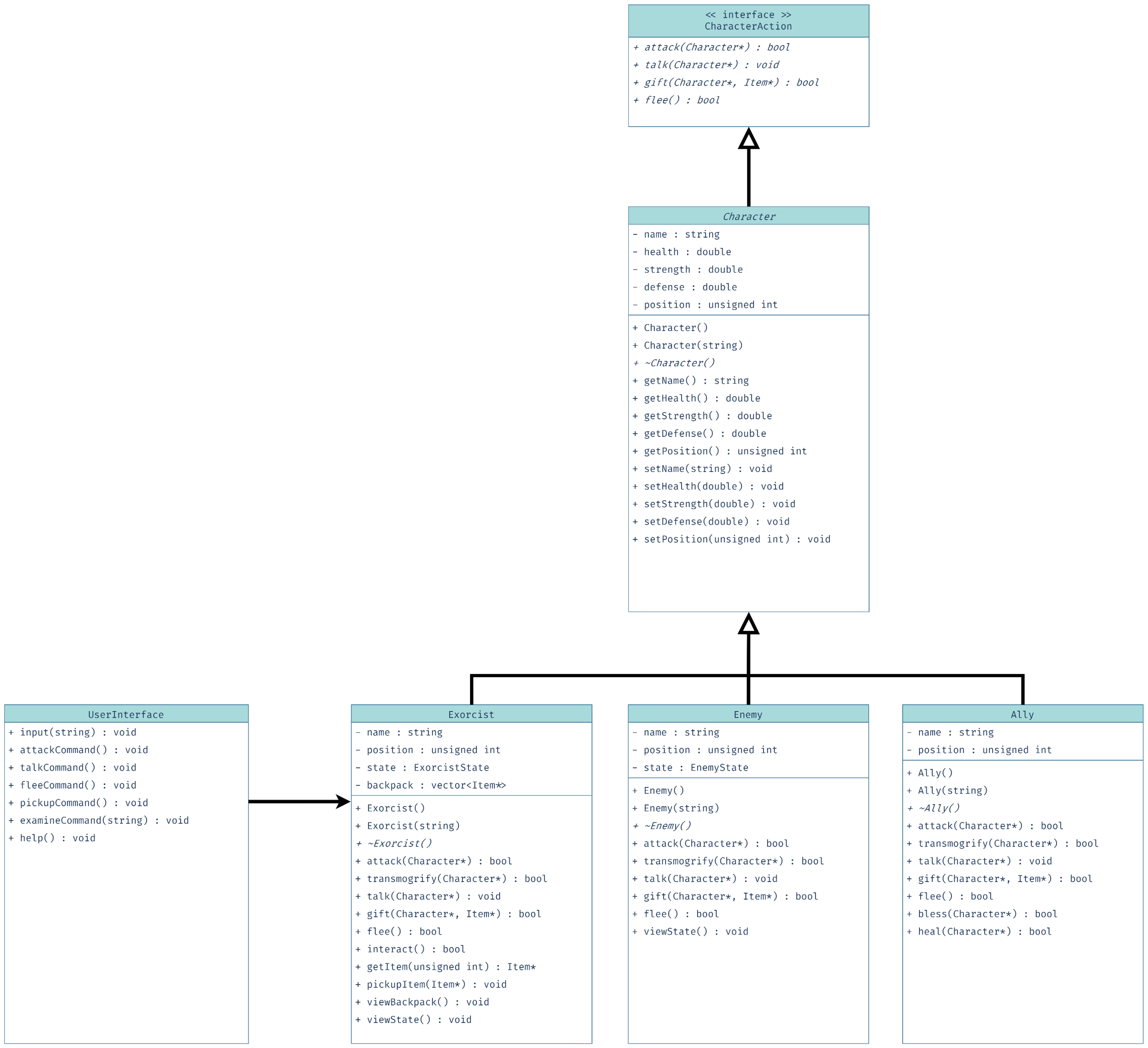
We are using Discord to communicate everything to do with the project. This is where we will have our meetings, share any resources, help each other out, and many other things.

CHANGE MANAGEMENT

- If a bug report shows an error in a method, the Quality Assurance Lead is responsible to identify which team member wrote that method and assign them to fix it.
- If a bug report shows an error from the interaction of code written by one or more people and it's not clear as to why the bug appeared, the team will meet up and discuss how we can fix it and the Quality Assurance Lead will assign one person to fix it.

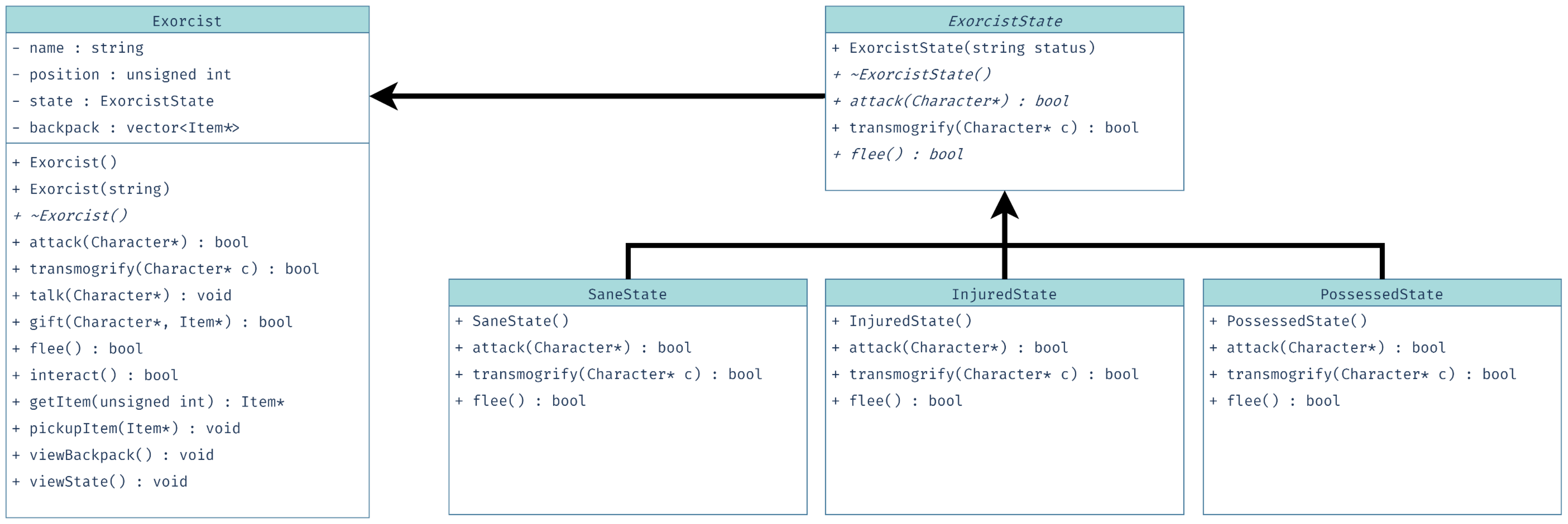
SOFTWARE DESIGN

DESIGN – CLASS DIAGRAMS

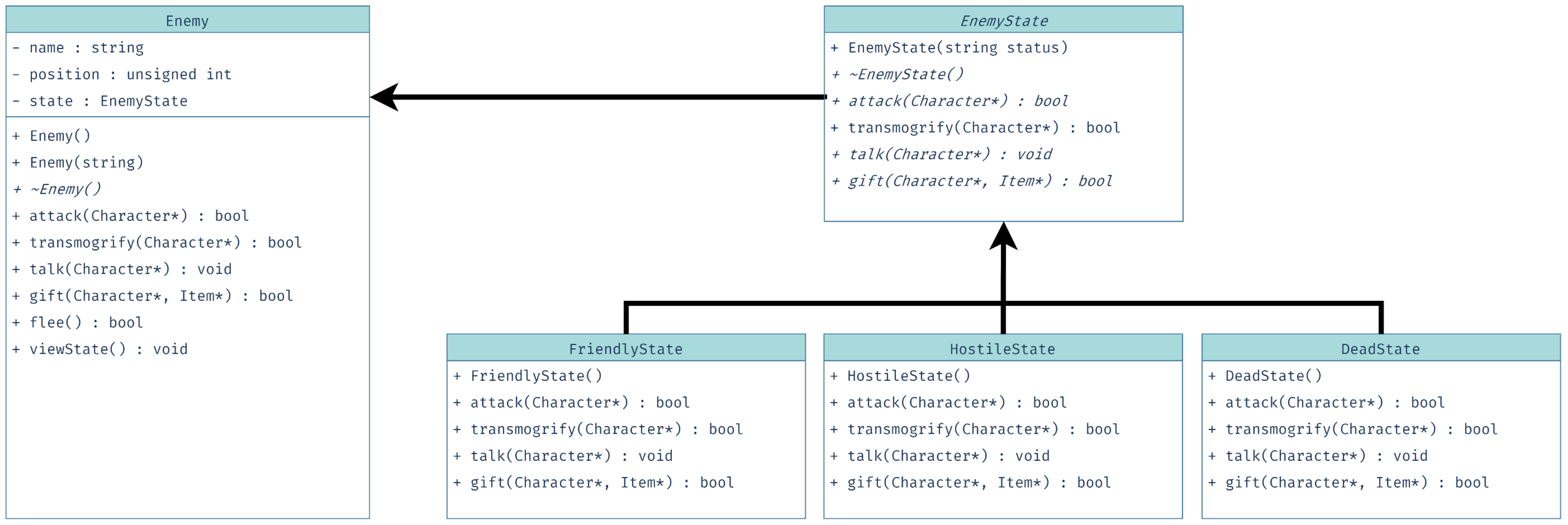


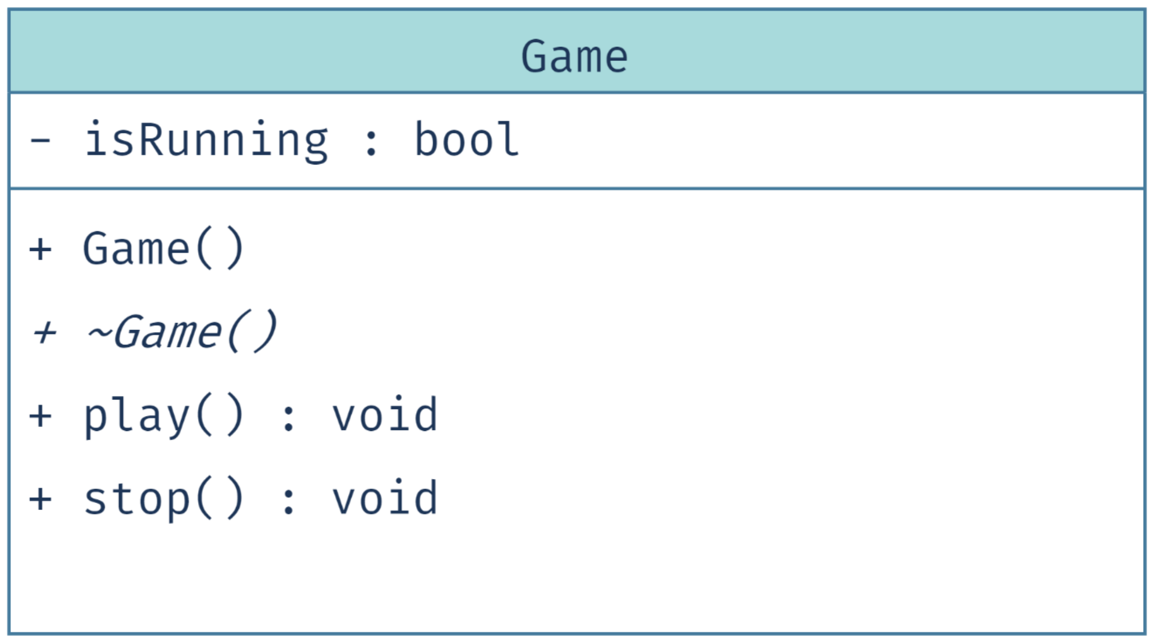
<< interface >>
CharacterAction
+ attack(Character*) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool
+ flee() : bool
Character
- name : string
- health : double
- strength : double
- defense : double
- position : unsigned int
+ Character()
+ Character(string)
+ ~Character()
+ getName() : string
+ getHealth() : double
+ getStrength() : double
+ getDefense() : double
+ getPosition() : unsigned int
+ setName(string) : void
+ setHealth(double) : void
+ setStrength(double) : void
+ setDefense(double) : void
+ setPosition(unsigned int) : void
UserInterface
+ input(string) : void
+ attackCommand() : void
+ talkCommand() : void
+ fleeCommand() : void
+ pickupCommand() : void
+ examineCommand(string) : void
+ help() : void
Exorcist
- name : string
- position : unsigned int
- state : ExorcistState
- backpack : vector<Item*>
+ Exorcist()
+ Exorcist(string)
+ ~Exorcist()
+ attack(Character*) : bool
+ transmogrify(Character*) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool
+ flee() : bool
+ interact() : bool
+ getItem(unsigned int) : Item*
+ pickupItem(Item*) : void
+ viewBackpack() : void
+ viewState() : void
Enemy
- name : string
- position : unsigned int
- state : EnemyState
+ Enemy()
+ Enemy(string)
+ ~Enemy()
+ attack(Character*) : bool
+ transmogrify(Character*) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool
+ flee() : bool
+ viewState() : void
Ally
- name : string
- position : unsigned int
+ Ally()
+ Ally(string)
+ ~Ally()
+ attack(Character*) : bool
+ transmogrify(Character*) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool
+ flee() : bool
+ bless(Character*) : bool
+ heal(Character*) : bool

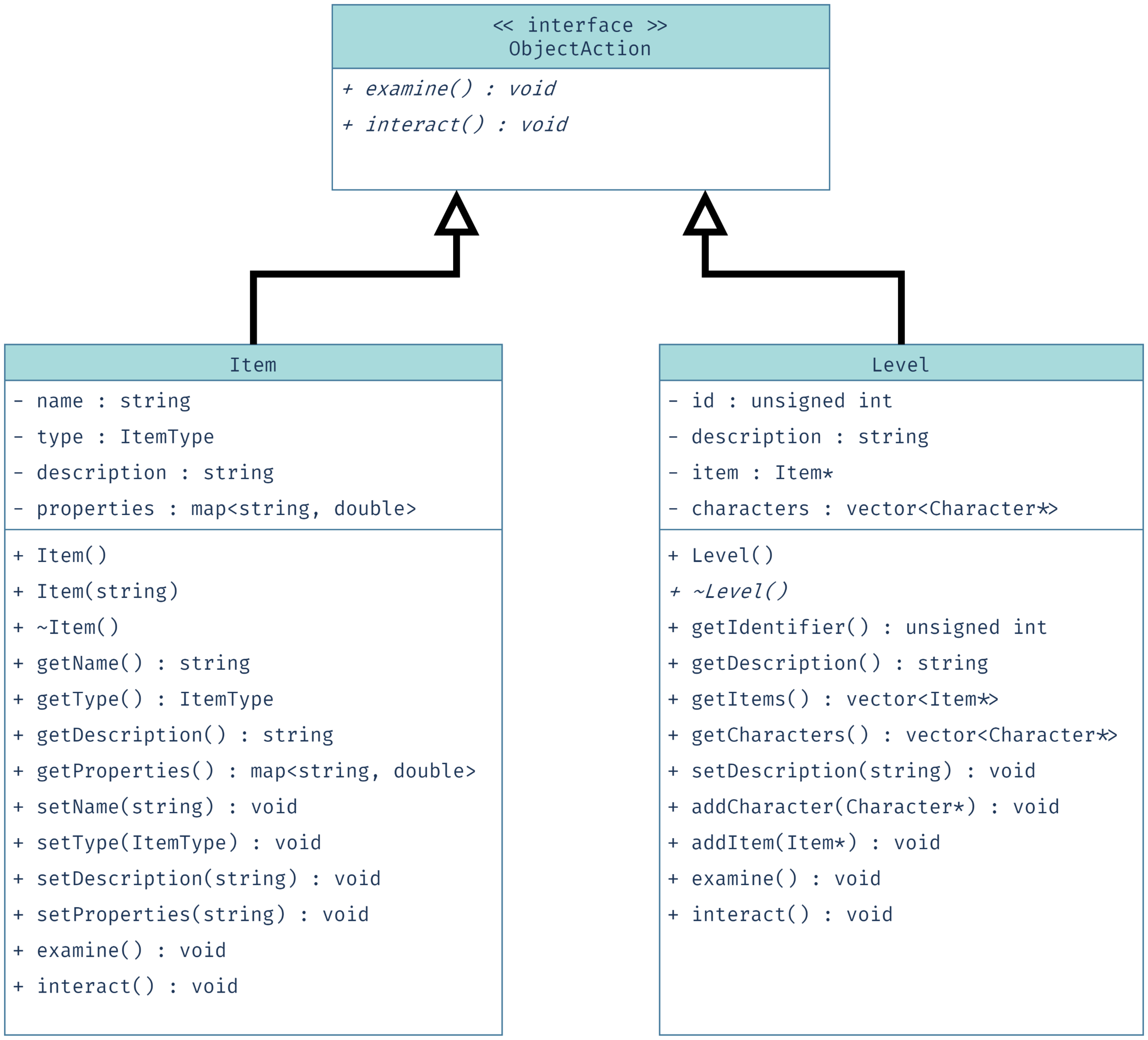
State Pattern


Exorcist
- name : string
- position : unsigned int
- state : ExorcistState
- backpack : vector<Item*>
+ Exorcist()
+ Exorcist(string)
+ ~Exorcist()
+ attack(Character*) : bool
+ transmogrify(Character* c) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool
+ flee() : bool
+ interact() : bool
+ getItem(unsigned int) : Item*
+ pickupItem(Item*) : void
+ viewBackpack() : void
+ viewState() : void
ExorcistState
+ ExorcistState(string status)
+ ~ExorcistState()
+ attack(Character*) : bool
+ transmogrify(Character* c) : bool
+ flee() : bool
SaneState
+ SaneState()
+ attack(Character*) : bool
+ transmogrify(Character* c) : bool
+ flee() : bool
InjuredState
+ InjuredState()
+ attack(Character*) : bool
+ transmogrify(Character* c) : bool
+ flee() : bool
PossessedState
+ PossessedState()
+ attack(Character*) : bool
+ transmogrify(Character* c) : bool
+ flee() : bool

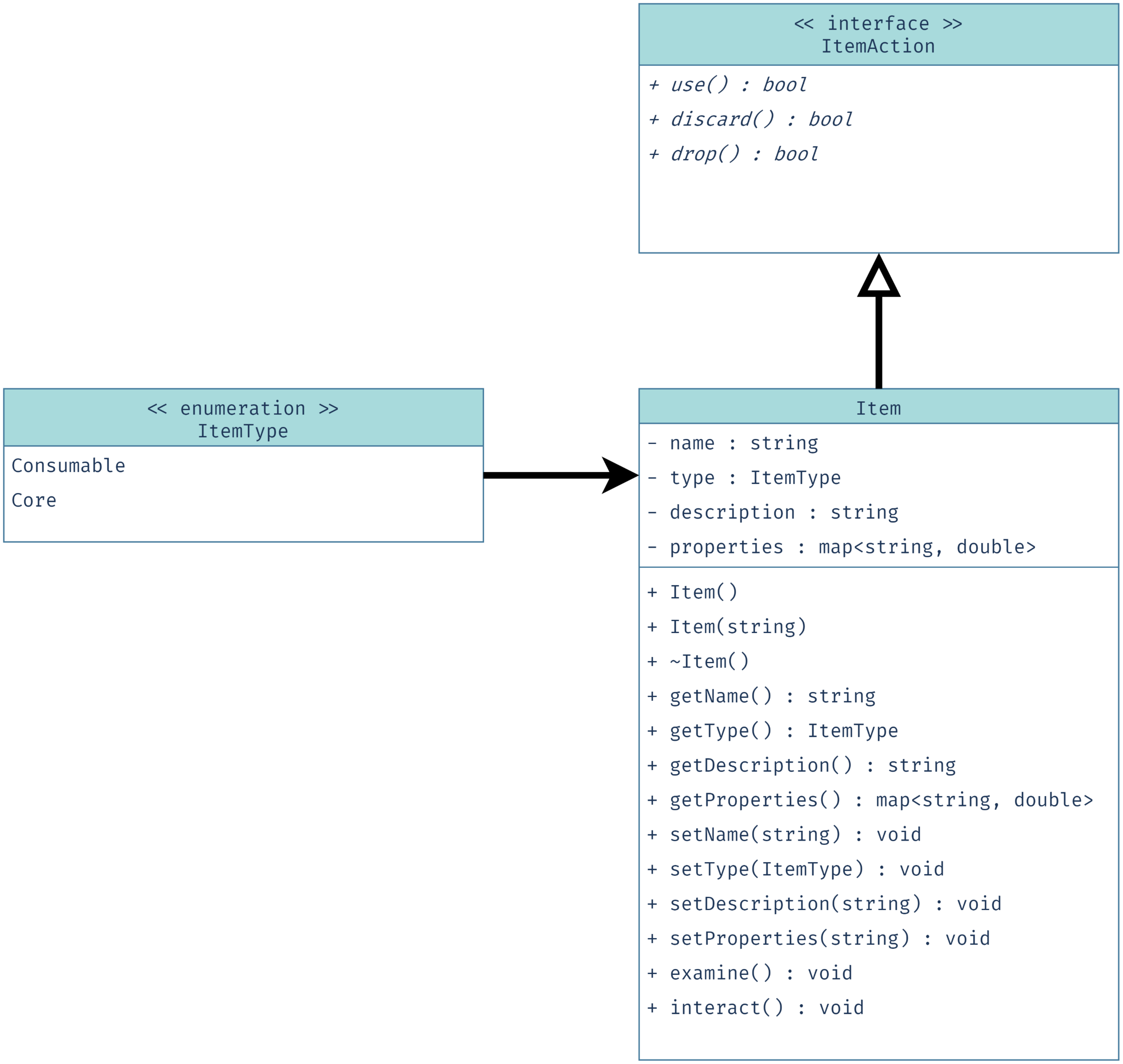
State Pattern


Enemy
- name : string
- position : unsigned int
- state : EnemyState
+ Enemy()
+ Enemy(string)
+ ~Enemy()
+ attack(Character*) : bool
+ transmogrify(Character*) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool
+ flee() : bool
+ viewState() : void
EnemyState
+ EnemyState(string status)
+ ~EnemyState()
+ attack(Character*) : bool
+ transmogrify(Character*) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool
FriendlyState
+ FriendlyState()
+ attack(Character*) : bool
+ transmogrify(Character*) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool
HostileState
+ HostileState()
+ attack(Character*) : bool
+ transmogrify(Character*) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool
DeadState
+ DeadState()
+ attack(Character*) : bool
+ transmogrify(Character*) : bool
+ talk(Character*) : void
+ gift(Character*, Item*) : bool

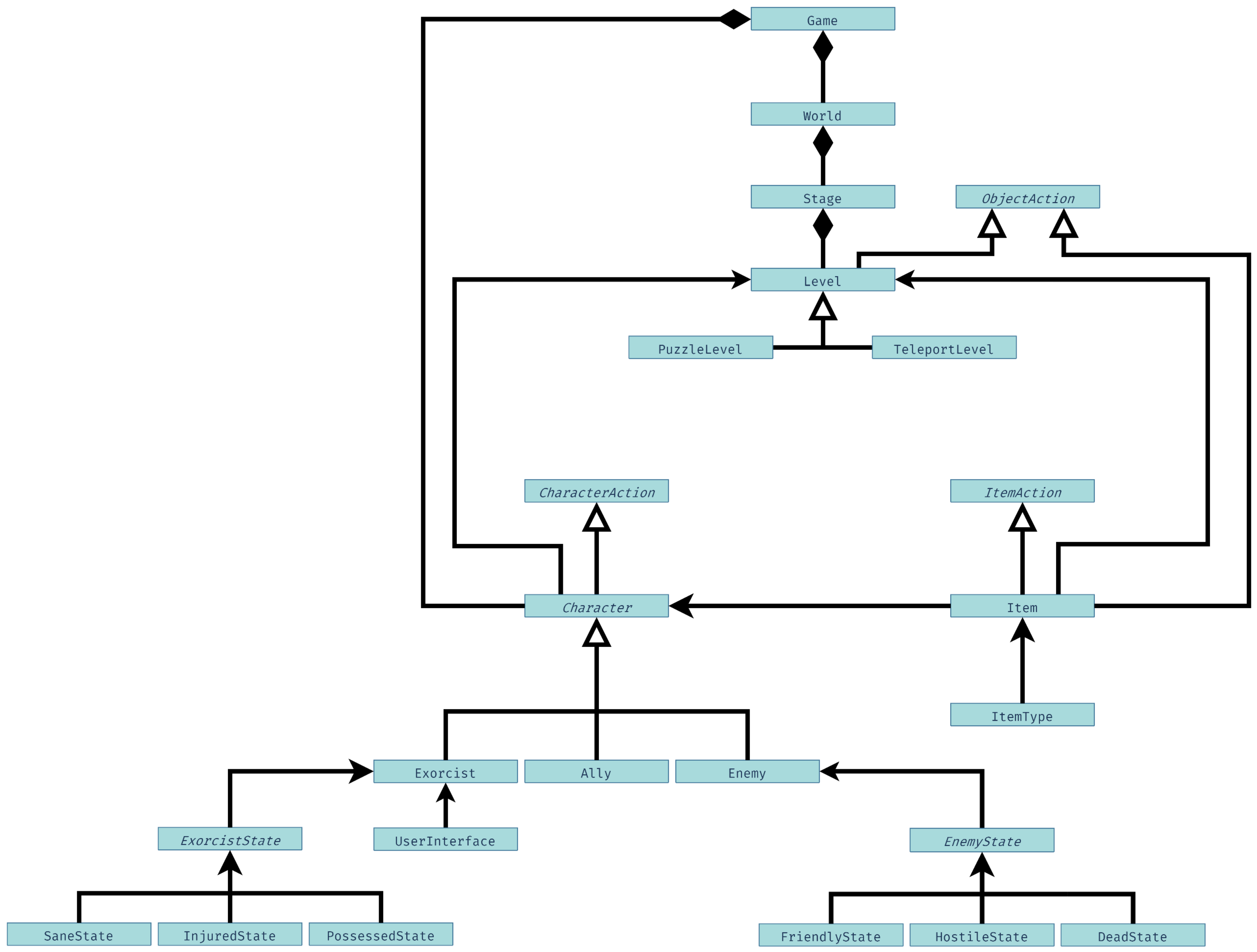

Game
- isRunning : bool
+ Game()
+ ~Game()
+ play() : void
+ stop() : void

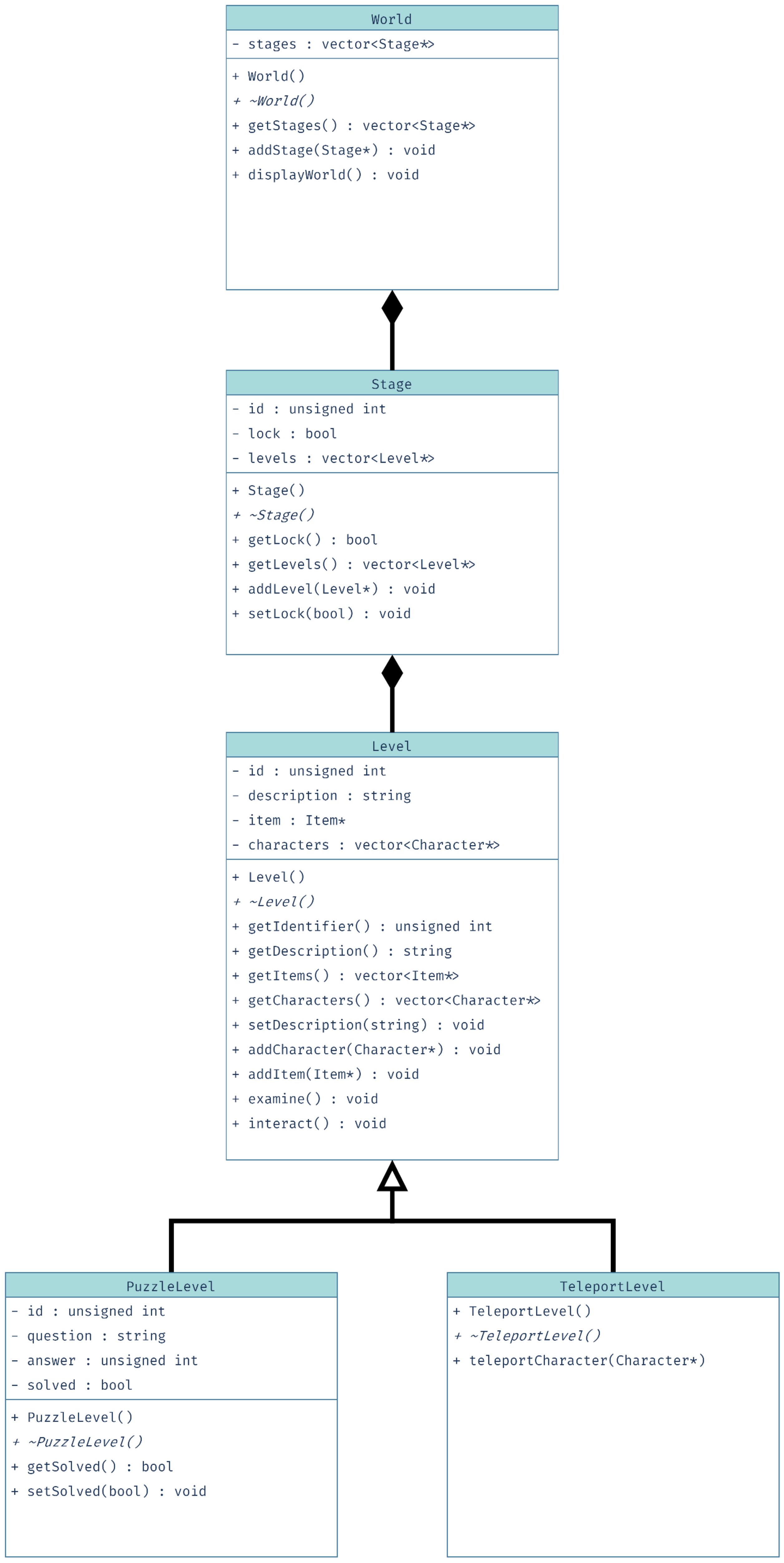


<< interface >>
ObjectAction
+ examine() : void
+ interact() : void
Item
- name : string
- type : ItemType
- description : string
- properties : map<string, double>
+ Item()
+ Item(string)
+ ~Item()
+ getName() : string
+ getType() : ItemType
+ getDescription() : string
+ getProperties() : map<string, double>
+ setName(string) : void
+ setType(ItemType) : void
+ setDescription(string) : void
+ setProperties(string) : void
+ examine() : void
+ interact() : void
Level
- id : unsigned int
- description : string
- item : Item*
- characters : vector<Character*>
+ Level()
+ ~Level()
+ getIdentifier() : unsigned int
+ getDescription() : string
+ getItems() : vector<Item*>
+ getCharacters() : vector<Character*>
+ setDescription(string) : void
+ addCharacter(Character*) : void
+ addItem(Item*) : void
+ examine() : void
+ interact() : void

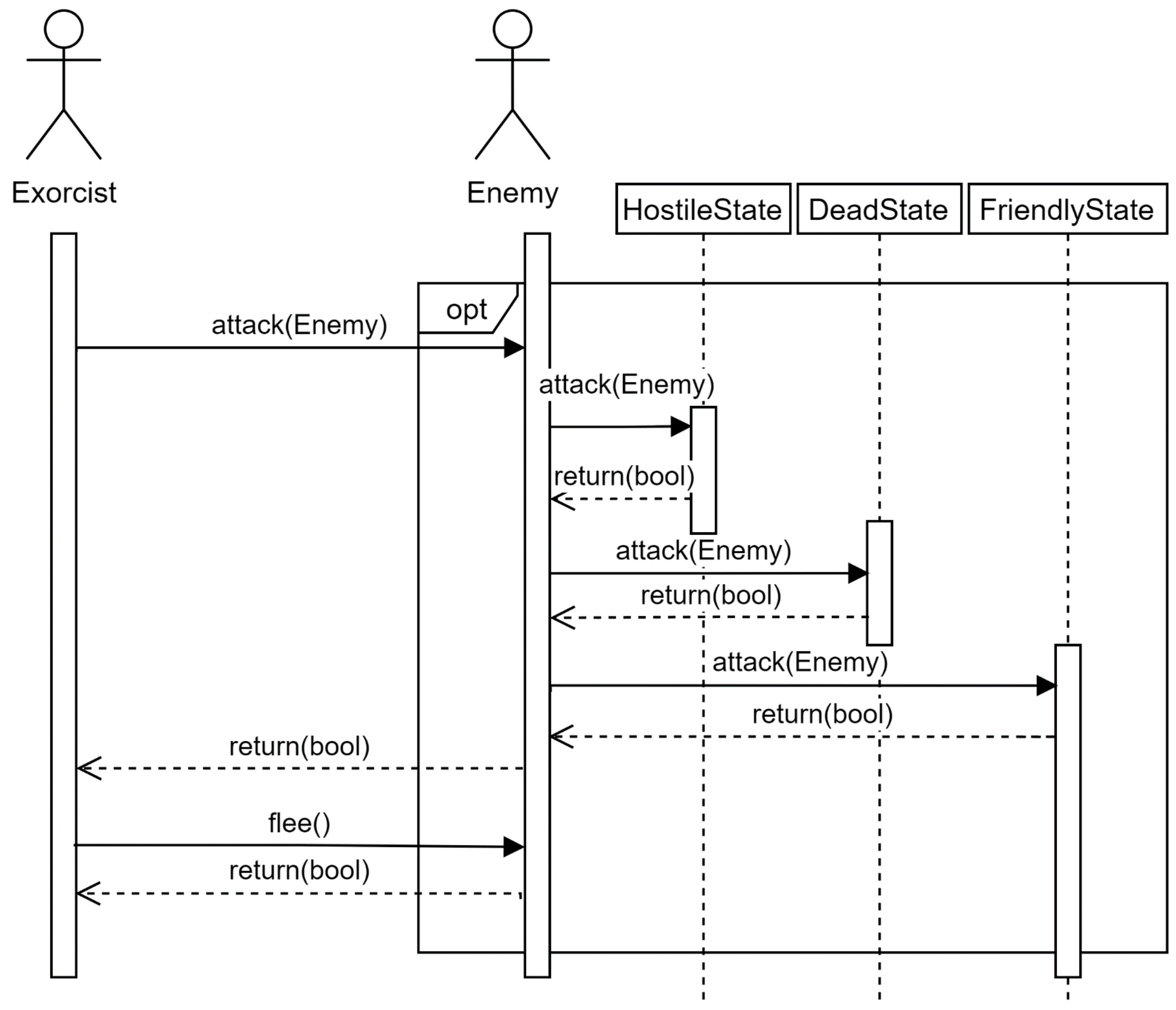


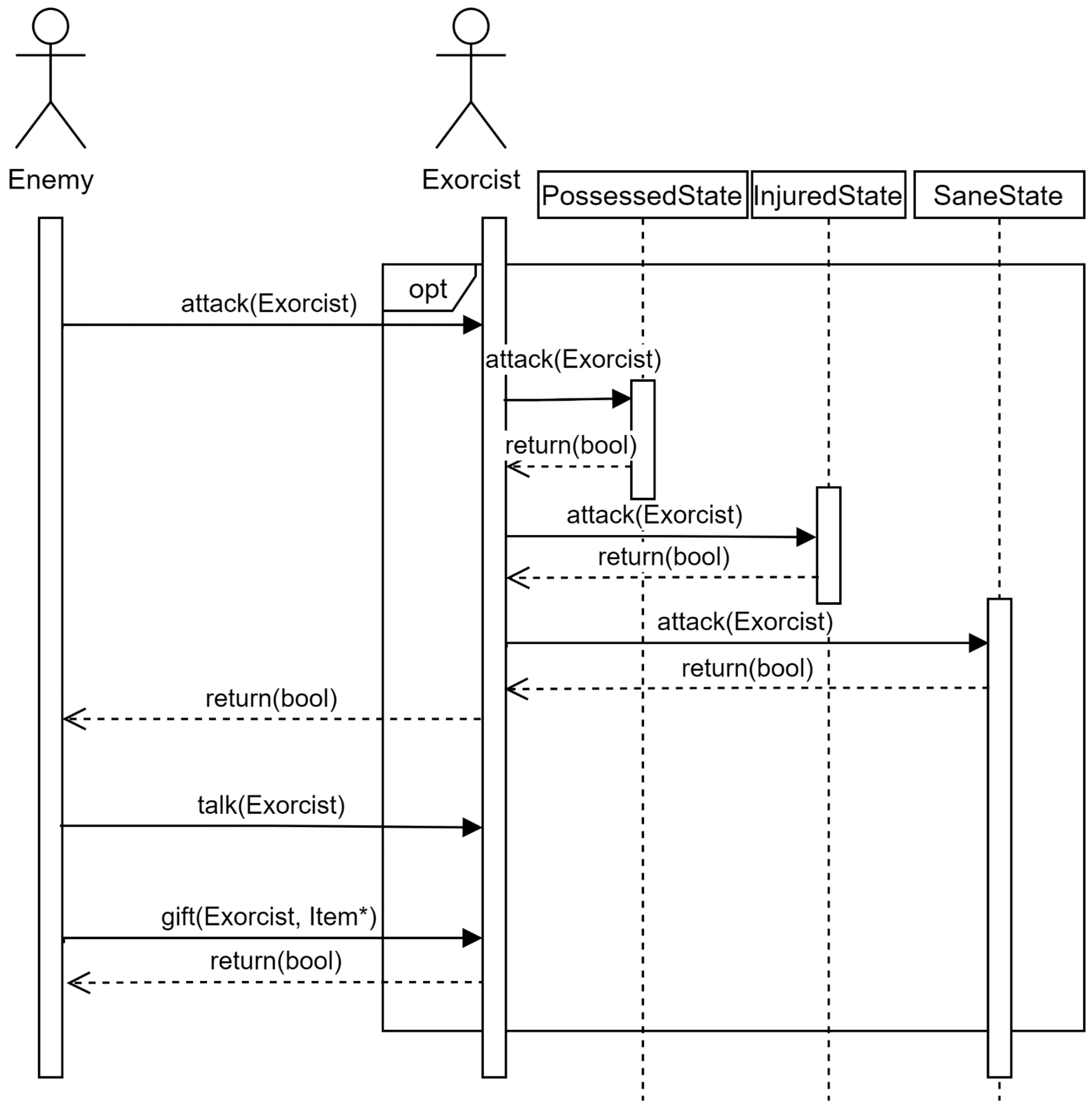
<< interface >>
ItemAction
+ use() : bool
+ discard() : bool
+ drop() : bool
<< enumeration >>
ItemType
Consumable
Core
Item
- name : string
- type : ItemType
- description : string
- properties : map<string, double>
+ Item()
+ Item(string)
+ ~Item()
+ getName() : string
+ getType() : ItemType
+ getDescription() : string
+ getProperties() : map<string, double>
+ setName(string) : void
+ setType(ItemType) : void
+ setDescription(string) : void
+ setProperties(string) : void
+ examine() : void
+ interact() : void

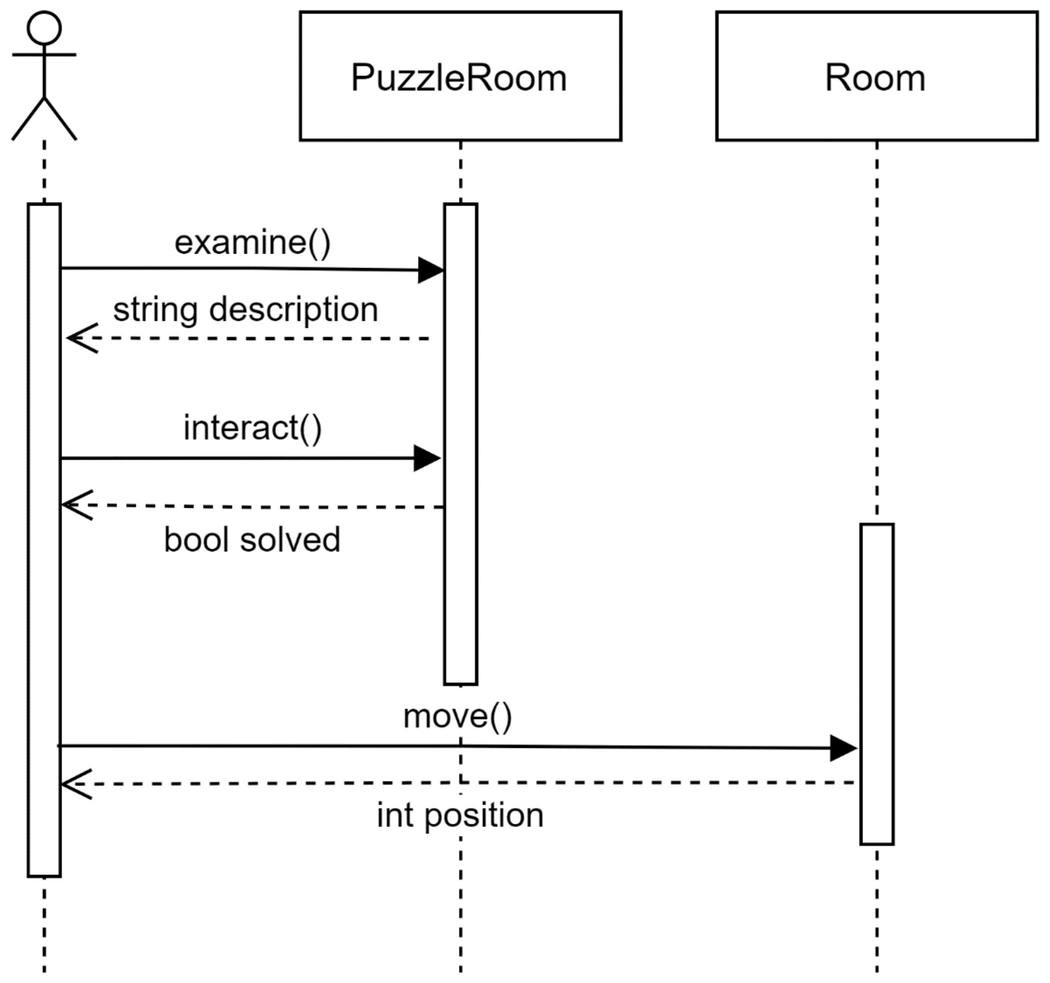

Game
World
Stage
ObjectAction
Level
PuzzleLevel
TeleportLevel
CharacterAction
ItemAction
Character
Item
ItemType
Exorcist
Ally
Enemy
ExorcistState
UserInterface
EnemyState
SaneState
InjuredState
PossessedState
FriendlyState
HostileState
DeadState

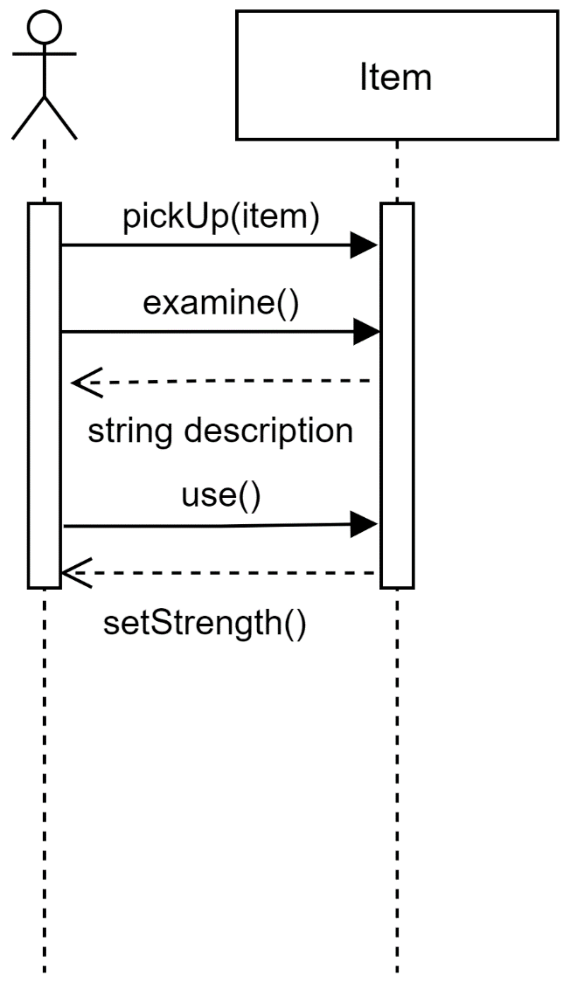

World
- stages : vector<Stage*>
+ World()
+ ~World()
+ getStages() : vector<Stage*>
+ addStage(Stage*) : void
+ displayWorld() : void
Stage
- id : unsigned int
- lock : bool
- levels : vector<Level*>
+ Stage()
+ ~Stage()
+ getLock() : bool
+ getLevels() : vector<Level*>
+ addLevel(Level*) : void
+ setLock(bool) : void
Level
- id : unsigned int
- description : string
- item : Item*
- characters : vector<Character*>
+ Level()
+ ~Level()
+ getIdentifier() : unsigned int
+ getDescription() : string
+ getItems() : vector<Item*>
+ getCharacters() : vector<Character*>
+ setDescription(string) : void
+ addCharacter(Character*) : void
+ addItem(Item*) : void
+ examine() : void
+ interact() : void
PuzzleLevel
- id : unsigned int
- question : string
- answer : unsigned int
- solved : bool
+ PuzzleLevel()
+ ~PuzzleLevel()
+ getSolved() : bool
+ setSolved(bool) : void
TeleportLevel
+ TeleportLevel()
+ ~TeleportLevel()
+ teleportCharacter(Character*)

DESIGN – SEQUENCE DIAGRAMS




Enemy
Exorcist
PossessedState
InjuredState
SaneState
opt
attack(Exorcist)
attack(Exorcist)
return(bool)
attack(Exorcist)
return(bool)
attack(Exorcist)
return(bool)
return(bool)
talk(Exorcist)
gift(Exorcist, Item*)
return(bool)


PuzzleRoom
Room
examine()
string description
interact()
bool solved
move()
int position


Item
pickUp(item)
examine()
string description
use()
setStrength()

CLASS DESCRIPTIONS

Ally

A concrete class derived from the Character base class. A friendly companion to assist the Exorcist in the game.

The default constructor for Ally.

```
Ally();
```

A constructor for the Ally class.
n is the name of the Ally to be created.

```
Ally(std::string n);
```

The destructor for Ally.

```
virtual ~Ally();
```

The ally attempts to attack a Character.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
virtual bool attack(Character* c);
```

Alter a Character's modifier(s).
c is the pointer to a Character to transmogrify.
true if the transmogrification was successful, false otherwise.

```
virtual bool transmogrify(Character* c);
```

The ally attempts to talk to a Character.
c is the pointer to a Character to talk to.

```
virtual void talk(Character* c);
```

The ally attempts to gift a Character an item.
c is the pointer to a Character to gift
i is the pointer to an Item to gift
true if gifting was successful, false otherwise.

```
virtual bool gift(Character* c, Item* i);
```

The ally attempts to flee from combat.
true if able to flee, false otherwise.

```
virtual bool flee();
```

The ally attempts to bless a Character.
c is the pointer to a Character to bless.
true if the blessing was successful, false otherwise.

```
bool bless(Character* c);
```

The ally attempts to heal a Character.
c is the pointer to a Character to heal.
true if the healing was successful, false otherwise.

```
bool heal(Character* c);
```

Character

An abstract base class for Exorcist, Enemy and Ally to inherit from.

The default constructor for Character.

```
Character();
```

A constructor for the Character class.
n is the name of the Character to be created.

```
Character(std::string n);
```

The destructor for Character.

```
virtual ~Character() {}
```

Get the name of the Character.
returns the name of the Character as a string.

```
std::string getName();
```

Get the current health of the Character.
returns the current health of the Character as a double.

```
double getHealth();
```

Get the amount of strength the Character can do.
returns the amount of strength the Character can do as a double.

```
double getStrength();
```

Get the defensive modifier for a Character.
returns the defensive modifier of a Character as a double.

```
double getDefense();
```

Get the modifier percentage for a Character.
returns the modifier percentage of a Character as a double.

```
double getModifier();
```

Get the position of the Character in the map.
returns the position of the Character as an unsigned int.

```
unsigned int getPosition();
```

Set the name of the Character.
n is the name to be set for the Character.

```
void setName(std::string n);
```

Set the health of the Character.
h is the health to be set for the Character.

```
void setHealth(double);
```

Set the strength of the Character.
d is the strength to be set for the Character.

```
void setStrength(double);
```

Set the defensive modifier of the Character.
d is the defensive modifier to be set for the Character.

```
void setDefense(double);
```

Set the modifier percentage for a Character.
m is the modifier to be set for the Character.

```
void setModifier(double m);
```

Set the position of the Character within the map.
p is the position to be set for the Character.

```
void setPosition(unsigned int);
```

CharacterAction

An interface for the action(s) of a Character.

The Character base class will not have a default
implementation due to concrete classes using the
State Pattern to alter the implementation.

Attack a Character, friend or foe.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
virtual bool attack(Character* c) = 0;
```

Alter the modifier(s) of the Character.
c is the pointer to a Character to transmogrify.
true if the modification was successful, false otherwise.

```
virtual bool transmogrify(Character* c) = 0;
```

Talk to a Character, friend or foe.
c is the pointer to a Character to talk to.

```
virtual void talk(Character* c) = 0;
```

Gift a Character an item, friend or foe.
c is the pointer to a Character to gift.
i is the pointer to an Item to gift.
true if gifting was successful, false otherwise.

```
virtual bool gift(Character* c, Item* i) = 0;
```

Flee from a Character if in combat.
true if able to flee, false otherwise.

```
virtual bool flee() = 0;
```

Enemy

A concrete class derived from the Character base class. An Enemy could be generic or a boss, and will be spawned in a hostile state. An item or defeat could change it's state.

The default constructor for Enemy.

```
Enemy();
```

A constructor for the Enemy class.
n is the name of the Enemy to be created.

```
Enemy(std::string n);
```

The destructor for Enemy.

```
virtual ~Enemy();
```

The Enemy attempts to attack a Character.
This method is affected by EnemyState.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
virtual bool attack(Character* c);
```

Alter the Enemy's modifier(s).
This method is affected by EnemyState.
c is the pointer to a Character to transmogrify.
true if the transmogrification was successful, false otherwise.

```
virtual bool transmogrify(Character* c);
```

The Enemy attempts to talk to a Character.
This method is affected by EnemyState.
C is the pointer to a Character to talk to.

```
virtual void talk(Character* c);
```

The Enemy attempts to gift a Character an item.
This method is affected by EnemyState.
c is the pointer to a Character to gift
i is the pointer to an Item to gift

true if gifting was successful, false otherwise.

```
virtual bool gift(Character* c, Item* i);
```

The Enemy attempts to flee from combat.
true if able to flee, false otherwise.

```
virtual bool flee();
```

Get the state of the Enemy.
the state of the Enemy as an EnemyState.

```
EnemyState getState();
```

Set the state of the Enemy.
s is the state to be set for the Enemy.

```
void setState(EnemyState s);
```

View the state of the Enemy.

```
void viewState();
```

EnemyState

EnemyState is used to control various states of the Enemy, depending on the context.

The default constructor for the EnemyState class.
n is the name of the state.

```
EnemyState(std::string n) : name(n) {}
```

The destructor for EnemyState.

```
virtual ~EnemyState()
```

The Enemy attempts to attack a Character.
It depends on the context and state of the Enemy.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
virtual bool attack(Character* c) = 0;
```

The Enemy's modifier(s) are altered.
It depends on the context and state of the Enemy.
c is the pointer to a Character to transmogrify.
true if the transmogrification was successful, false otherwise.

```
virtual bool transmogrify(Character* c) = 0;
```

The Enemy attempts to talk to a Character.
It depends on the context and state of the Enemy.
c is the pointer to a Character to talk to.

```
virtual void talk(Character* c) = 0;
```

The Enemy attempts to gift a Character.
It depends on the context and state of the Enemy.
c is the pointer to a Character to attack.
i is the pointer to an Item to gift
true if the Enemy gifted an item, false otherwise.

```
virtual bool gift(Character* c, Item* i) = 0;
```

FriendlyState

FriendlyState is a state that indicates that the Enemy is friendly, and will no longer attack the Exorcist, depending on context or item received.

The Enemy will not attack a Character.
The attempt will fail.
c is the pointer to a Character to attack.
returns false.

```
bool attack(Character* c);
```

The Enemy's modifier(s) are altered.
The modifier will be higher than the default value.
c is the pointer to a Character to transmogrify.
returns true.

```
bool transmogrify(Character* c);
```

The Enemy will talk to a Character in a friendly manner.
c is the pointer to a Character to talk to.

```
void talk(Character* c);
```

The Enemy may gift a Character an item.
c is the pointer to a Character to attack.
i is the pointer to an Item to gift
true if the Enemy gifted an item, false otherwise.

```
bool gift(Character* c, Item* i);
```

HostileState

HostileState is a state that indicates that the Enemy is hostile, and will attack and engage in combat with the Exorcist, depending on context or until the Enemy is defeated.

The default constructor for the HostileState class.

```
HostileState();
```

The Enemy attempts to attack a Character.

The attempt will often be successful in this state.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
bool attack(Character* c);
```

The Enemy's modifier(s) are altered.
The modifier will be higher than the default value.
c is the pointer to a Character to transmogrify.
returns false.

```
bool transmogrify(Character* c);
```

The Enemy will talk to a Character in a hostile manner.
c is the pointer to a Character to talk to.

```
void talk(Character* c);
```

The Enemy will never gift a Character in this state.
c is the pointer to a Character to attack.
i is the pointer to an Item to gift
returns false.

```
bool gift(Character* c, Item* i);
```

DeadState

DeadState is a state that indicates that the Enemy is dead through combat or an item/spell, and will no longer attack or converse with a Character.

The default constructor for the DeadState class.

```
DeadState();
```

The Enemy is unable to attack while dead.
The attempt will always fail in this state.
c is the pointer to a Character to attack.
returns false.

```
bool attack(Character* c);
```

The Enemy's modifier(s) are altered.
The modifier will be set to zero.

```
c is the pointer to a Character to transmogrify.
returns false.

bool transmogrify(Character* c);


The Enemy may or may not speak in this state.
The attempt to speak may result in nonsense.
c is the pointer to a Character to talk to.

void talk(Character* c);


The Enemy may gift a Character with an item upon defeat.
It is dependent on the Enemy having an item.
c is the pointer to a Character to attack.
i is the pointer to an Item to gift
true if the Enemy gifted an item, false otherwise.

bool gift(Character* c, Item* i);
```

Exorcist

A concrete class derived from the Character base class. The Exorcist is the main character, and consists of three states that will determine how likely an action will succeed and the amount of stat bonuses applied.

The default constructor for Exorcist.

```
Exorcist();
```

A constructor for the Exorcist class.
n the name of the Exorcist to be created.

```
Exorcist(std::string n);
```

The destructor for Exorcist.

```
virtual ~Exorcist();
```

The Exorcist attempts to attack a Character, friend or foe.
This method is affected by ExorcistState.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
virtual bool attack(Character* c);
```

Alter the Exorcist's modifier(s).
This method is affected by ExorcistState.
c is the pointer to a Character to transmogrify.
true if the transmogrification was successful, false otherwise.

```
virtual bool transmogrify(Character* c);
```

The Exorcist attempts to talk to a Character, friend or foe.
c is the pointer to a Character to talk to.

```
virtual void talk(Character* c);
```

The Exorcist attempts to gift a Character an item, friend or foe.
c is the pointer to a Character to gift
i is the pointer to an Item to gift
true if gifting was successful, false otherwise.

```cpp
virtual bool gift(Character* c, Item* i);
```

```
The Exorcist attempts to flee from combat with an Enemy.
This method is affected by ExorcistState.
true if able to flee, false otherwise.
```

```cpp
virtual bool flee();
```

```
Get an item from the Exorcist's backpack.
i is the index of the item.
returns a pointer to the item.
```

```cpp
Item* getItem(unsigned int i);
```

```
Pick-up an Item to add to the Exorcist's backpack.
I is the pointer to the Item.
```

```cpp
void pickupItem(Item* i);
```

```
Get the state of the Exorcist.
returns the state of the Exorcist as an ExorcistState*.
```

```cpp
ExorcistState* getState();
```

```
Set the state of the Exorcist.
s the state to be set for the Exorcist.
```

```cpp
void setState(ExorcistState* s);
```

```
The Exorcist attempts to interact with the environment.
true if there is an interaction to be had, false otherwise.
```

```cpp
bool interact();
```

```
View the content of the Exorcist's backpack.
```

```cpp
void viewBackpack();
```

```
View the state of the Exorcist.
```

```cpp
void viewState();
```

ExorcistState

ExorcistState is used to control various states of the Exorcist, depending on the context.

The default constructor for the ExorcistState class.
n is the name of the state.

```
ExorcistState(std::string n) : name(n) {}
```

The destructor for ExorcistState.

```
virtual ~ExorcistState() {}
```

The Exorcist attempts to attack a Character.
It depends on the context and state of the Exorcist.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
virtual bool attack(Character* c) = 0;
```

The Exorcist's modifier(s) are altered.
It depends on the context and state of the Exorcist.
c is the pointer to a Character to transmogrify.
true if the transmogrification was successful, false otherwise.

```
virtual bool transmogrify(Character* c) = 0;
```

The Exorcist attempts to flee from combat with an Enemy.
It depends on the context and state of the Exorcist.
true if able to flee, false otherwise.

```
virtual bool flee() = 0;
```

SaneState

SaneState is a state that indicates that the Exorcist is in a normal state, which is the default. As the game progresses, the Exorcist may lose their sanity.

The default constructor for the SaneState class.

```
SaneState();
```

The Exorcist attempts to attack a Character.
The attempt will often be successful in this state.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
bool attack(Character* c);
```

The Exorcist's modifier(s) are altered.
The modifier is reset to the default in this state.
c is the pointer to a Character to transmogrify.
returns false.

```
bool transmogrify(Character* c);
```

The Exorcist attempts to flee from an Enemy.
The attempt will often be successful in this state.
true if able to flee, false otherwise.

```
bool flee();
```

InjuredState

InjuredState is a state that indicates that the Exorcist is injured from running into a trap, answering a puzzle incorrectly or engaging in combat with enemies.

The default constructor for the InjuredState class.

```
InjuredState();
```

The Exorcist attempts to attack a Character.
The attempt will sometimes be successful in this state.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
bool attack(Character* c);
```

The Exorcist's modifier(s) are altered.
The modifier will be lower than the default value
c is the pointer to a Character to transmogrify.
true if the transmogrification was successful, false otherwise.

```
bool transmogrify(Character* c);
```

The Exorcist attempts to flee from an Enemy.
The attempt will sometimes be successful in this state.
true if able to flee, false otherwise.

```
bool flee();
```

PossessedState

PossessedState is a state that indicates that the Exorcist is possessed and will require an item or blessing from an ally to go back to a Sane state. The game will be harder, and lower the Exorcist's stats.

The default constructor for the PossessedState class.

```
PossessedState();
```

The Exorcist attempts to attack a Character.
The attempt will often fail in this state.
c is the pointer to a Character to attack.
true if the attack was successful, false otherwise.

```
bool attack(Character* c);
```

The Exorcist's modifier(s) are altered.
The modifier will be lower than the default value
c is the pointer to a Character to transmogrify.
true if the transmogrification was successful, false otherwise.

```
bool transmogrify(Character* c);
```

The Exorcist attempts to flee from an Enemy.
The attempt will often be fail in this state.
true if able to flee, false otherwise.

```
bool flee();
```

Game

A game session can be created, played and stopped, depending on the user's input.

The default constructor for Game.

```
Game();
```

The destructor for Game.

```
virtual ~Game() {}
```

Play the game

```
void play();
```

Stop the game

```
void stop();
```

Item

The structure and functionality of an item that can be used, modified or consumed.

The default constructor for the Item class.

```
Item();
```

A constructor for the Item class.
n the name of the item.

```
Item(std::string n);
```

The destructor for Item.

```
virtual ~Item() {}
```

Get the name of the item.
returns the name of the item as a string.

```
std::string getName();
```

Get the type of the item.
returns the type of the item as an enumeration.

```
ItemType getType();
```

Get the description of the Item.
returns the description of the Item as a string.

```
std::string getDescription();
```

Get the properties of the item.
returns the properties of the item as a map<string, double>.

```
std::map<std::string, double> getProperties();
```

Set the name of the item.
n is the name to be set for the item.

```
void setName(std::string);
```

Set the type of the item.
t is the type to be set for the item

```
void setType(ItemType t);
```

Set the type of the Item.
d is the description to be set for the Item

```
void setDescription(std::string d);
```

Set the properties of the item.
p is the properties to be set for the item

```
void setProperties(std::string p);
```

Examine and get detailed information about the Item.

```
virtual void examine();
```

Interact with the Item.

```
virtual void interact();
```

ItemAction

An interface for the action(s) of an Item.

The Item base class will not have a default implementation for the following methods.

Use an item.

```
virtual bool use() = 0;
```

Discard an item.

```
virtual bool discard() = 0;
```

Drop an item on the ground.

```
virtual void drop() = 0;
```

Level

The structure of a Level. A Level may contain Puzzle(s), Item(s), Enemies or Allies.

The default constructor for Level.

```
Level();
```

A constructor for Level.
i is the identifier for the Level

```
Level(unsigned int i);
```

The destructor for Level.

```
virtual ~Level() {}
```

Get the identifier of the Level.
returns the identifier of the Level as an unsigned int.

```
unsigned int getIdentifier();
```

Get the description of the Level.
returns the description of the Level as a string.

```
std::string getDescription();
```

Get the Item(s) contained in the Level.
returns a vector of Item(s) in the Level.

```
std::vector<Item*> getItems();
```

Get the Character(s) contained in the Level.
returns a vector of Character(s) in the Level.

```
std::vector<Character*> getCharacters();
```

Set the description of the Level.
d is the description for the Level.

```
void setDescription(std::string d);
```

Add a Character to the Level.
c is the pointer of the Character to add to the Level.

```
void addCharacter(Character* c);
```

Add an item to the Level.
i is the pointer of the Item to add to the Level.

```
void addItem(Item* i);
```

Examine and get detailed information about the Level.

```
virtual void examine();
```

Interact with the Level.

```
virtual void interact();
```

ObjectAction

An interface for the action(s) of an object.

Interact with an Item or Level.

```
virtual bool interact() = 0;
```

Examine and get detailed information about the Item or Level.

```
virtual bool examine() = 0;
```

PuzzleLevel

A PuzzleLevel is a unique level where the Exorcist must solve a puzzle in order to proceed. In doing so, a reward could be given or the ability to unlock the next Stage.

The default constructor for PuzzleLevel.

```
PuzzleLevel();
```

The destructor for PuzzleLevel.

```
virtual ~PuzzleLevel() {}
```

Get the status of the PuzzleLevel.
returns a boolean indicating if the Level was solved.

```
bool getSolved();
```

Set the status of the PuzzleLevel.
s is a boolean indicating if the Level was solved.

```
void setSolved(bool s);
```

TeleportLevel

A TeleportLevel is a unique level where the Exorcist will be allowed to teleport to a new Stage/Level if a condition is met.

The default constructor for TeleportLevel.

```
TeleportLevel();
```

The destructor for TeleportLevel.

```
virtual ~TeleportLevel() {}
```

The Exorcist can teleport to a certain level.
c is the pointer of the Character to be teleported.

```
teleportCharacter(Character* c);
```

World

A container for each stage. The World will be used to indicate a Character's position in the World.

The default constructor for World.

```
World();
```

The destructor for World.

```
virtual ~World();
```

Get the Stage(s) contained in the World.
returns a vector of Stage(s) in the World.

```
std::vector<Stage*> getStages();
```

Add a Stage to the World.
s is the pointer of the Stage to add to the World.

```
void addStage(Stage* s);
```

Display a text representation of the World.

```
void displayWorld();
```

APPENDICES

APPENDIX A: FIGURES AND TABLES