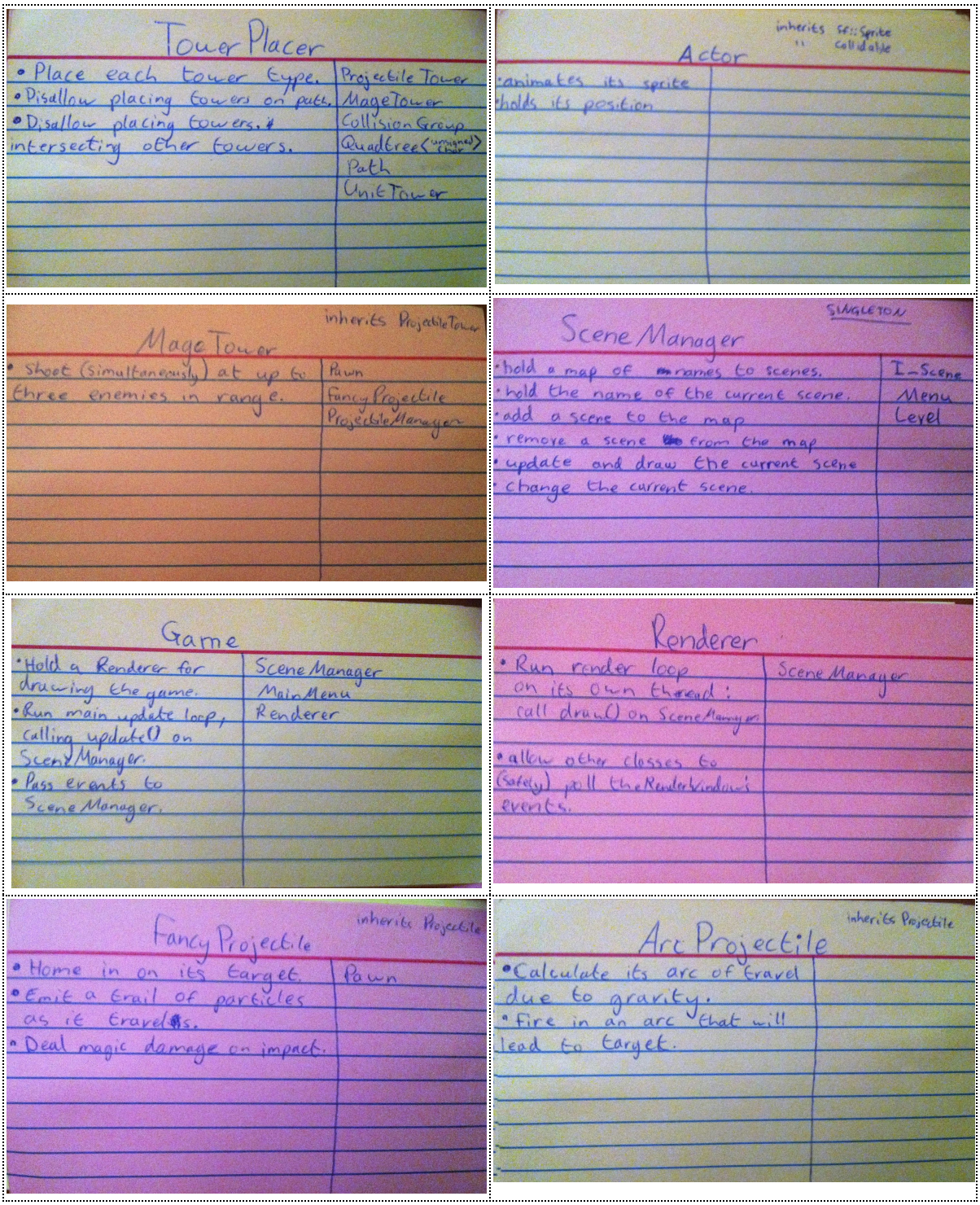
Technical Design Document (TDD)

CRC cards



TowerPlacer

- Place each tower type.
- Disallow placing towers on path.
- Disallow placing towers intersecting other towers.

Projectile Tower
MageTower
CollisionGroup
QuadTree<unsigned char>
Path
UnitTower

Actor

inherits sf::Sprite
collidable

- animates its sprite
- holds its position

MageTower

inherits ProjectileTower

- Shoot (simultaneously) at up to three enemies in range.

Pawn
FancyProjectile
ProjectileManager

SceneManager

SINGLETON

- hold a map of names to scenes.
- hold the name of the current scene.
- add a scene to the map
- remove a scene from the map
- update and draw the current scene
- change the current scene.

I-Scene
Menu
Level

Game

- Hold a Renderer for drawing the game.
- Run main update loop, calling update() on SceneManager.
- Pass events to SceneManager.

SceneManager
MainMenu
Renderer

Renderer

- Run render loop on its own thread: call draw() on SceneManager.
- allow other classes to (safely) poll the RenderWindow's events.

SceneManager

FancyProjectile

inherits Projectile

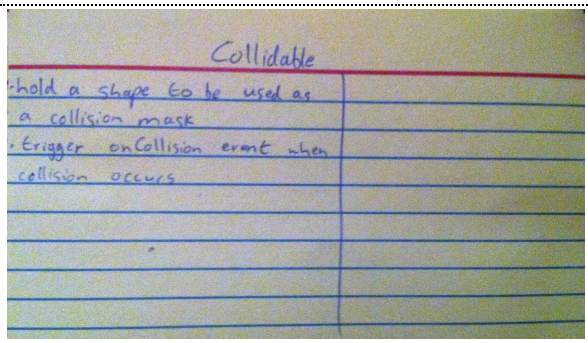
- Home in on its target.
- Emit a trail of particles as it travels.
- Deal magic damage on impact.

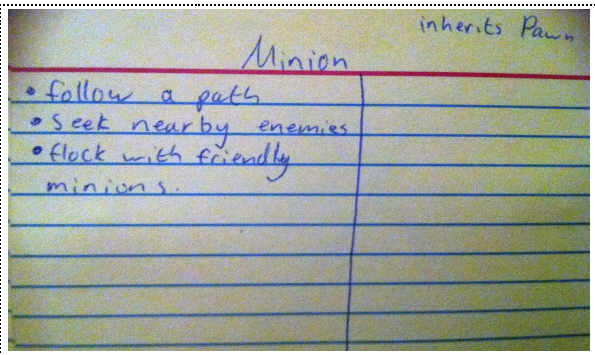
Pawn

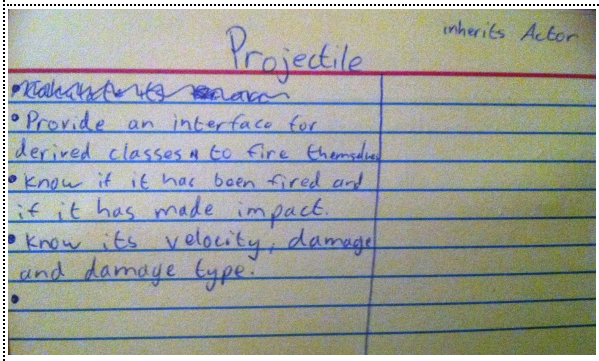
ArcProjectile

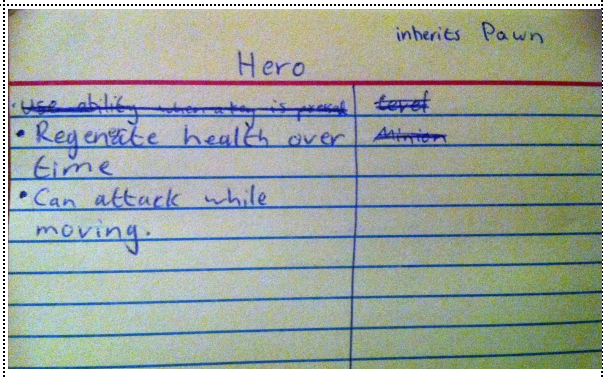
inherits Projectile

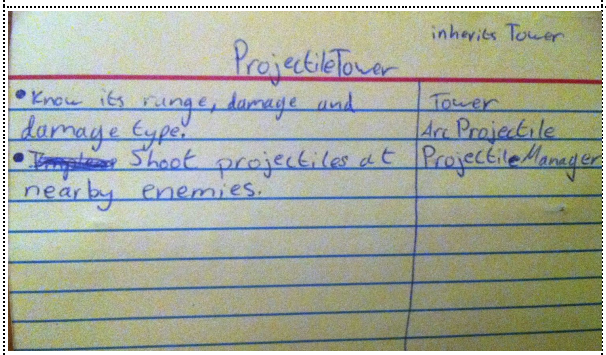
- Calculate its arc of travel due to gravity.
- Fire in an arc that will lead to target.

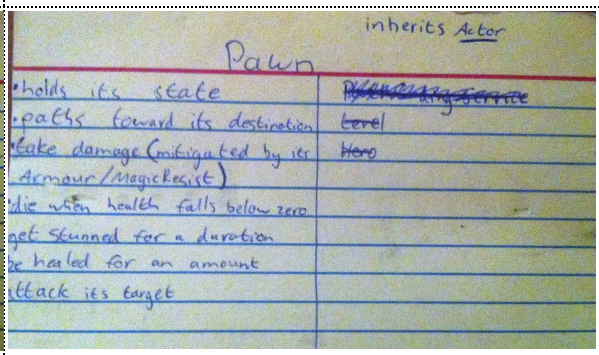

Collidable
hold a shape to be used as a collision mask
trigger onCollision event when collision occurs

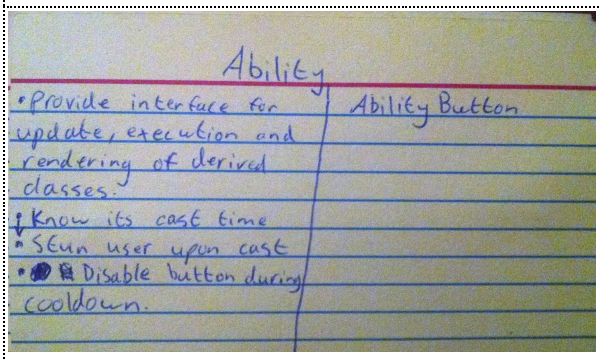

inherits Pawn
Minion
• follow a path
• Seek nearby enemies
• flock with friendly minions.

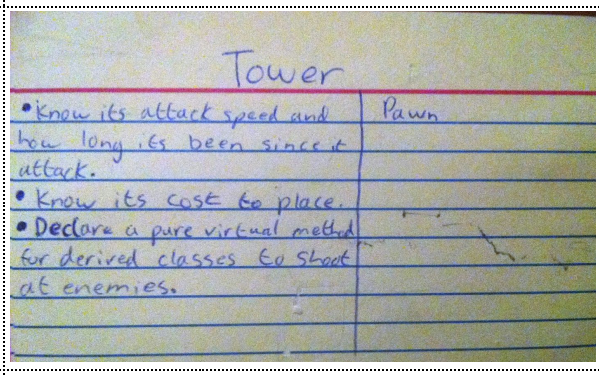

inherits Actor
Projectile
• Provide an interface for derived classes to fire themselves
• know if it has been fired and if it has made impact.
• know its velocity, damage and damage type.

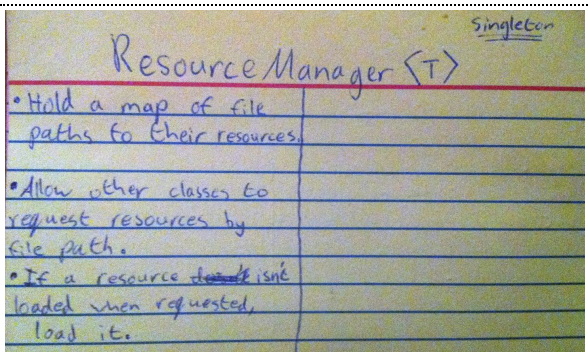

inherits Pawn
Hero
• Regenerate health over time
• Can attack while moving.

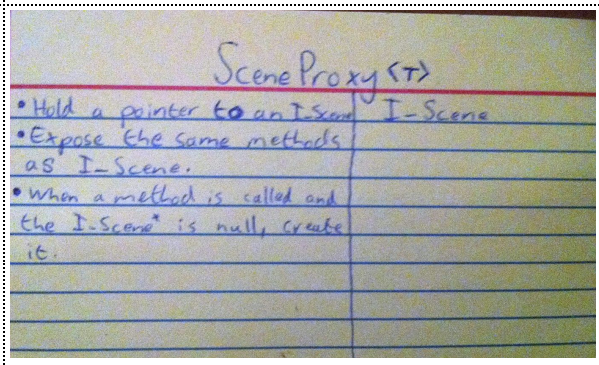

inherits Tower
ProjectileTower
• Know its range, damage and damage type.
• Shoot projectiles at nearby enemies.
Tower
Arc Projectile
Projectile Manager

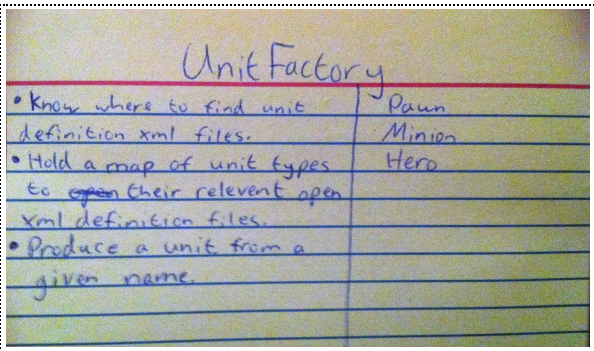

inherits Actor
Pawn
• holds its state
• paths toward its destination
• take damage (mitigated by its Armour/MagicResist)
• die when health falls below zero
• get stunned for a duration
• be healed for an amount
• attack its target
Hero


Ability
• Provide interface for update, execution and rendering of derived classes.
• Know its cast time
• Stun user upon cast
• Disable button during cooldown.
Ability Button


Tower
• Know its attack speed and how long its been since it attack.
• Know its cost to place.
• Declare a pure virtual method for derived classes to shoot at enemies.
Pawn

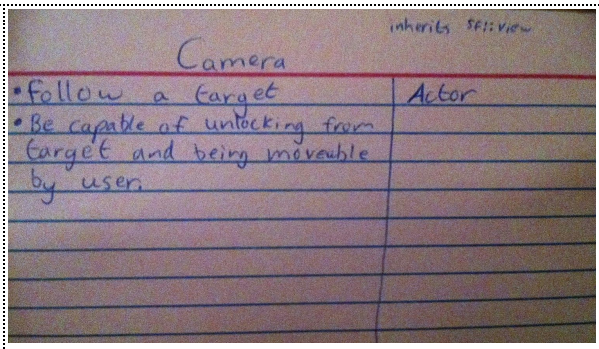

Singleton
ResourceManager <T>
• Hold a map of file paths to their resources.
• Allow other classes to request resources by file path.
• If a resource isn't loaded when requested, load it.


Scene Proxy <T>
• Hold a pointer to an I-Scene
• Expose the same methods as I-Scene.
• When a method is called and the I-Scene* is null, create it.
I-Scene



UnitFactory

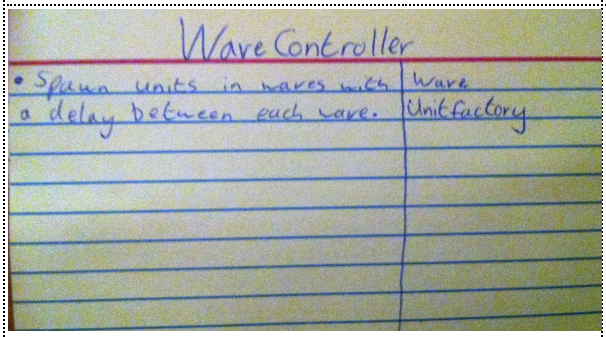
• Know where to find unit definition xml files.	Pawn
• Hold a map of unit types to ~~open~~ their relevent open xml definition files.	Minion
• Produce a unit from a given name.	Hero



inherits SFML View

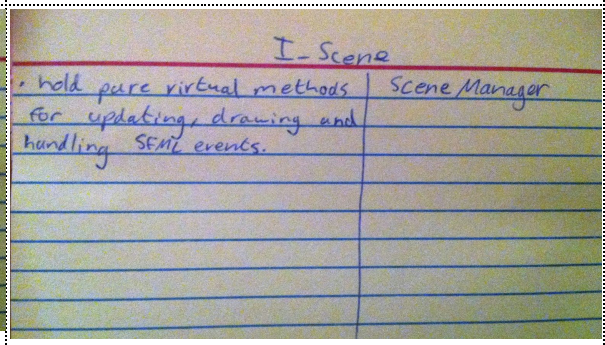
Camera

• Follow a target	Actor
• Be capable of unlocking from target and being moveable by user.	



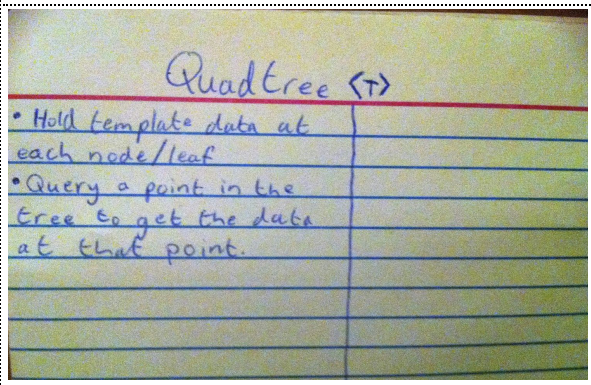
WaveController

• Spawn units in waves with a delay between each wave.	Wave
	UnitFactory



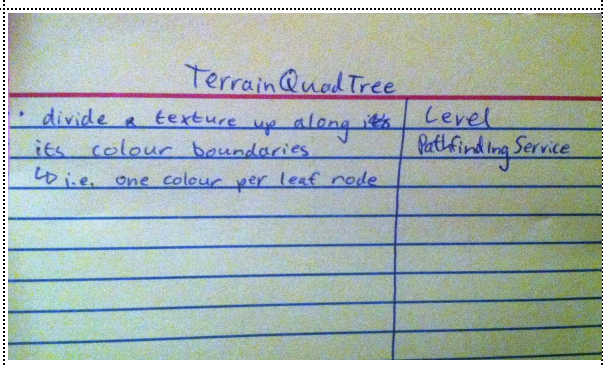
I_Scene

• hold pure virtual methods for updating, drawing and handling SFML events.	SceneManager



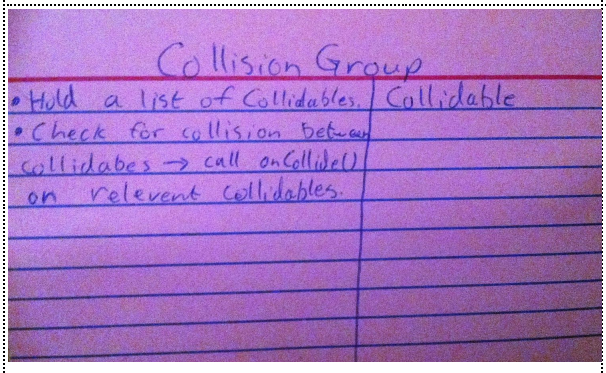
QuadTree <T>

• Hold template data at each node/leaf	
• Query a point in the tree to get the data at that point.	



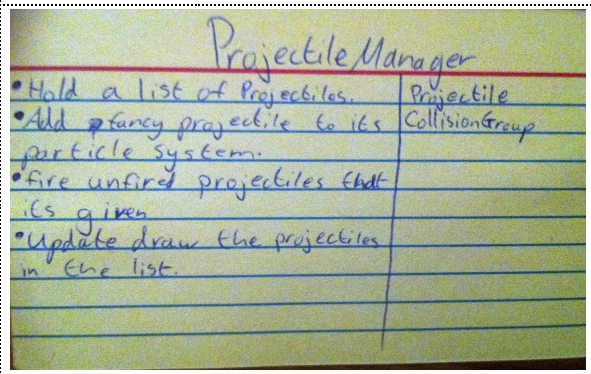
TerrainQuadTree

• divide a texture up along its colour boundaries	Level
↳ i.e. one colour per leaf node	PathfindingService



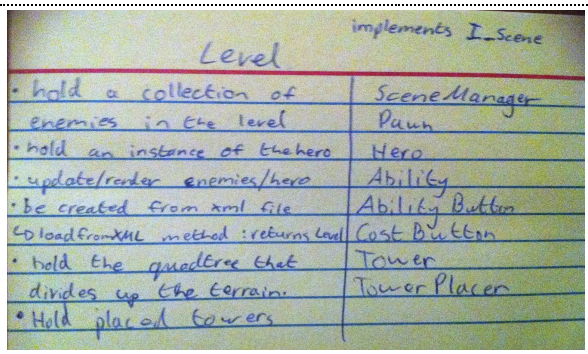
CollisionGroup

• Hold a list of Collidables.	Collidable
• Check for collision between collidabes → call onCollide() on relevent collidables.	



ProjectileManager

• Hold a list of Projectiles.	Projectile
• Add a fancy projectile to its particle system.	CollisionGroup
• fire unfired projectiles that it's given	
• Update draw the projectiles in the list.	

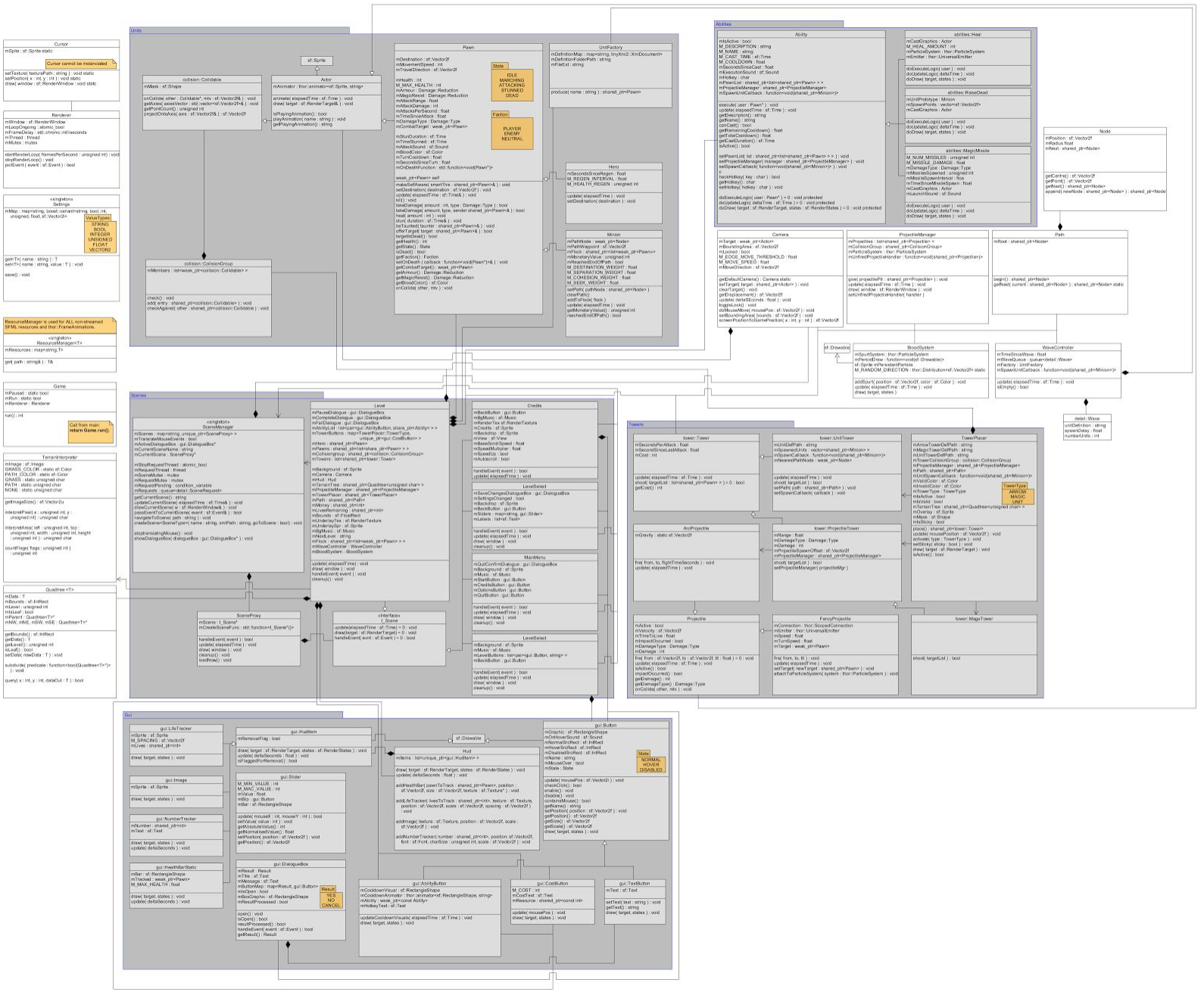


implements I_Scene

Level

• hold a collection of enemies in the level	SceneManager
• hold an instance of the hero	Pawn
• update/render enemies/hero	Hero
• be created from xml file	Ability
↳ loadFromXML method : returns Level	AbilityButton
• hold the quadtree that divides up the terrain.	CostButton
• Hold placed towers	Tower
	TowerPlacer

Architecture



Approaches

XML Levels:
Levels will be stored as xml files. This makes debugging easier as no code needs to be recompiled if only the level changes. Additionally, units, animations and abilities will also be stored as xml definitions.

Smart Pointers:
There should be no raw pointers in the code. All pointers should be one of the following c++11 smart pointers: unique_ptr, shared_ptr, weak_ptr. This takes away some of the memory management responsibility and reduces the chance of human error.

Factory Pattern:
When producing units, the factory pattern will be used. This will allow us to easily produce units as needed. The factory will know where to find the xml definition files and will produce units by name.

Quadtree terrain:
Terrain data, which is used to check for valid tower placement, will be stored in a quadtree structure. The tree will be subdivided on the borders between valid and invalid areas. This will allow us to easily query the terrain data at a particular location while also using minimal memory.

Thor library:
Thor provides some handy vector algebra as well as an easy to use particle system. Using Thor means we won't have to write our own vector functions such as length, normalise or dot product.

Proxy pattern:
The SceneManager will make use of SceneProxies. SceneProxies hold an I_Scene pointer and the file path use to create the scene. When a method is called on the proxy: if the pointer is null, the scene is initialised. This is also known as Lazy-Loading and it means that the scenes don't need to exist before they're used.

Feature Design

Setup Tasks

Task 1: Install required middleware

- SFML
- Thor
- Boost

Task 2: Initial Application stub

- Create application stub
- Create local repo & github repo
- Push Repo to remote

Setup basic game system

Task 1: Create Scene interface

The I_Scene interface class will expose four pure virtual methods, allowing us to program many different scenes but hold them in one container and not worry about the derived type. The methods that the scene interface must expose are:

- handleEvent(event : sf::Event&) : bool
  - Allows derived classes to handle sfml events.
  - Returns true if the event was handled (i.e. shouldn't be processed by Game class. An example is the Closed event, which should only be handled by the Game class).
- update(elapsedTime : sf::Time const&) : void
  - Allows derived classes to update according to delta time.
- draw(window : sf::RenderWindow&) : void
  - Allows derived classes to draw themselves to the window.
- cleanup() : void
  - Allows derived classes to stop any sounds or music that they are playing.

Task 2: Create SceneProxy class

The SceneProxy class will expose the same public methods as the Scene interface. When any of these methods are invoked the proxy will check if the underlying scene has been created. If the scene has not been created the the scene will first be created and then the appropriate method will be invoked. The SceneProxy must be a template class: this will allow us to use it for any type of scene.

Additionally, the SceneProxy's constructor will accept the path to the xml that defines the scene. The SceneProxy will hold this path and use the xml document to create its underlying scene object.

Task 3: Create SceneManager class

The SceneManager class should be a singleton; there is no use to having two scene managers and navigation and creation requests should be submittable from anywhere. The SceneManager must hold a map of strings to pointers to I_Scenes. The manager will also have to keep track of the current scene as a string field. The SceneManager should lock a mutex before calling any methods on the managed scenes (and unlock it after). The methods that the SceneManager must expose are:

- instance() : *SceneManager
  - Static. Provides singleton access to all other public methods.
- updateCurrentScene(elapsedTime : sf::Time) : void
  - Calls update() on the current scene.
- drawCurrentScene(window : sf::RenderWindow&) : void
  - Calls draw() on the current scene.
- passEventToCurrentScene(event : sf::Event&) : void
  - Calls handleEvent() on the current scene.
- navigateToScene(name : std::string const&) : void
  - Changes current scene to the the new one if it exists in the map.
- createScene<SceneType>(name : std::string const&, xmlPath : std::string const&, goToScene : bool) : void
  - Template method.
  - Creates a SceneProxy<SceneType> with xmlPath and inserts it into the map of scenes.
  - If goToScene is true (defaults to true) then the current scene is set to be the newly created scene.

Task 4: Create Renderer class

The Render class creates and manages an sfml RenderWindow. It's constructor should take the same parameters as the sfml RenderWindow class. The Renderer will use a separate thread for rendering. The Renderer will need to hold an atomic boolean: if the boolean is true the render loop will continue, otherwise it will stop and the thread will be joinable. The class's methods should use a mutex to avoid multithreading complications. The Renderer needs one private method:

- render() : void
  - Main render loop. Uses while loop in tandem with the class' atomic boolean.
  - Implements a fixed timestep.
  - Calls drawCurrentScene() on the SceneManager instance.

The Renderer also need the following three public methods:

- startRenderLoop(framesPerSecond : unsigned int const) : void
  - Sets the frame delay field according to framesPerSecond.
  - Begins the rendering thread. The thread runs the render() method.
- stopRenderLoop() : void

    - Sets the atomic boolean to false (preventing render() from looping again).
    - Joins the rendering thread.
- pollEvent(event : sf::Event&) : bool
    - Wrapper for sfml RenderWindow's pollEvent method.

Task 5: Create Game class

The Game class will hold a Renderer and two static booleans (*mRun* and *mPaused*). It only needs one private method:

- handleEvent(event : sf::Event&) : void
    - Calls passEventToCurrentScene() on SceneManager instance.
    - If event is not handled by the current scene, then the Game class will try to handle it.
    - If the event type is Closed: set the running bool to false so that the game ends.
    - If the event type is LostFocus or GainedFocus: set mPaused appropriately so that the game is paused while it doesn't have focus.

The Game class also needs the following two public methods:

- run() : int
    - Sets mRun to true.
    - Does the main update loop of the game.
    - Uses a while loop in conjunction with mRun so that the method returns when mRun is set to false.
    - If mPaused is false, calls updateCurrentScene() on the SceneManager instance.
    - Returns EXIT_SUCCESS (0)
    - Intended to be called from main() as "*return game.run();*"
- close() : void
    - Static. Sets mRun to false.
    - Allows the game to be closed from inside a scene in a way that ensures everything is cleaned up properly. We should never make a call to exit() or use atExit() in our program.

The constructor for the Game class take no arguments.

Task 7: Create Collidable class

The Collidable class (namespace *collision*) is used for anything that requires collision. The Collidable class holds a mask (sf::Shape), used to collision, and an offset (sf::Vector2f), positional offset from parent position. The constructor accepts an XMLElement as its only argument; the shape definition and offset are both read from xml. Collidable needs six public methods:

- getMask() : sf::Shape const*
    - Gets the collision mask (the returned mask is constant)
- onCollide(other : Collidable*, mtv : sf::Vector2f) : void
    - Virtual.
    - other is the Collidable we're colliding with and mtv stands for minimum translation vector.
- debug_draw(target : sf::RenderTarget&) : void
    - Draws the mask to the target.

- getAxies(axiesVector : std::vector<sf::Vector2f>&) : void
    - Gets the normals from each face of the mask and adds them to the vector.
- getPointCount() : size_t
    - Calls getPointCount() on the mask.
- projectOntoAxis(axis : sf::Vector2f&) : sf::Vector2f
    - Projects each point of the mask onto an axis and returns the min and max point it touches on that axis.

It also needs this protected method:

- updateCollidableMask(newPosition : sf::Vector2f const&) : void
    - Sets the position of the mask to the newPosition plus the offset.

Task 8: Create ResourceManager class

ResourceManager is a template class and should work with any sfml resource (except music which is streamed). The ResourceManager should be used whenever possible over manually loading resources. It holds a map of strings to resources, where the key of each entry is its relative path in the file system. The class also needs a mutex so that it can be accessed safely from any thread. The ResourceManager provides the following public method:

- get(path : std::string const&) : T&
    - Loads the resource from file if it's not already in the map.
    - Returns a reference to the resource.
    - Specialised template required for thor::FrameAnimation resources.

Task 9: Create Actor class

The Actor class is used for anything that has collision and animates. It holds a thor::Animator<std::string, sf::Sprite> which it uses to animate itself. It publicly inherits both sf::Sprite and collision::Collidable. The constructor take an XMLElement as its only parameter. It needs the following four public methods:

- animate(elapsedTime : sf::Time const&) : void
    - Updates the animator and applies it to our self.
- isPlayingAnimation() : bool
    - Calls isPlayingAnimation() on the animator and returns the result.
- getPlayingAnimation() : std::string
    - Calls getPlayingAnimation() on the animator and returns the result.
- playAnimation(name : std::string const&, loop : bool) : void
    - Begins playing the named animation.
    - If loop is true (defaults to false), the animation will loop.
- draw(target : sf::RenderTarget&)
    - Draws the sprite to the target.

Task 10: Create Damage namespace and members

Inside the Damage namespace, write a Reduction class. Reduction is constructed from a float and provides operators for multiplication and assignment with floats.
Add a Type enum to the namespace with the members PHYSICAL and MAGICAL, these are the types of damage that can be inflicted.

Task 11: Create Pawn class

A Pawn is an Actor (public inheritance) that can move, attack, be stunned, die, etc. Each Pawn belongs to a faction (PLAYER, ENEMY, NEUTRAL) and can be in one of five states (IDLE, MARCHING, ATTACKING, STUNNED, DEAD). The constructor accepts an XMLElement as its only argument and assigns all variables from the xml data. Pawn has the following public methods:

- getDestination() : sf::Vector2f
- setDestination(destination : sf::Vector2f const&) : void
- getMovementSpeed() : int
- kill() : void
    - Kills the Pawn outright.
    - Sets health to 0 and state to DEAD.
- takeDamage(amount : int , type : Damage::Type) : bool
    - Decrements health by amount.
    - Multiplies damage by the appropriate Damage::Reduction.
    - Sets state to DEAD if damage brought health below 0.
    - Returns true if the damage killed the Pawn.
- heal(amount : int) : void
    - Increments health by amount, capping it at maximum health.
- stun(duration : sf::Time const&) : void
    - If duration is greater than remaining stun time, set state to STUNNED and set remaining stun time to be duration.
- beTaunted(taunter : Pawn*)
    - Sets the combat target to be the taunter.
- offerTarget(target : Pawn*)
    - Offers a target to the Pawn, who can accept or reject it.
    - Pawn should decline target if it is null or dead.
    - Pawn should accept target if current target is null, dead or far away.
- targetIsDead() : bool
    - Returns true if combat target is null or dead
- getHealth() : int
- getState() : State (enum)
- isDead() : bool
    - Return true if state is DEAD
- getFaction() : Faction (enum)
- getCombatTarget() : Pawn*
- getArmour() : Damage::Reduction
- getMagicResist() : Damage::Reduction
- onCollide(other : Collidable*, mtv : sf::Vector2f)
    - Overrides method in Collidable
    - If other can be dynamically cast to a Pawn, then call offerTarget() and pass the Pawn as the argument.
    - Moves by the mtv.
- update(elapsedTime : sf::Time const&) : void

- If state is STUNNED, then decrement stun time and skip rest of update.
- State becomes MARCHING if destination is further than attack range.
- State becomes ATTACKING if has a target and it is within attack range.
- Else state becomes IDLE.
- Moves toward goal if MARCHING.
- Attacks target if ATTACKING, taking into account attacks per second.
- Calls updateCollidableMask() with current position.

Task 12: Create UnitFactory class

The UnitFactory hold a map of strings to open tinyXml2 XMLDocuments. The constructor specifies where to look for unit definitions and what file extension the definitions use. It exposes only one method:

- produce(name : std::string const&) : Pawn*
  - If name doesn't exist in the map, find the definition file in the folder and add it to the map.
  - Passes the root element of the the definition to the Pawn constructor.
  - Returns a pointer to the newly created unit.

Task 13: Create Level class

The level class is derived from the scene interface. It holds a list of Pawn* as well as a separate Pawn*, mHero. The Level class must implement the i_Scene pure virtual methods:

- handleEvent(event : sf::Event&)
  - Does nothing at this point.
- update(elapsedTime : sf::Time const&) : void
  - Calls update() on each pawn in the list.
  - Does not call update() on mHero separately. mHero is also held in the vector, so its update is already taken care of.
- draw(window : sf::RenderWindow&)
  - Calls draw() on all pawns in the list.

Player movement

Task 1: Write Hero definition file

Create an xml file that defines a Pawn. This will be given to the constructor of the Pawn. Within the xml definition, nested inside the Pawn tag, should be an Actor tag. The Actor tags, in turn, should have a Collidable tag inside them. The xml definition should define all fields for their respective classes.

Task 2: Read xml definition inside constructors

Update constructors of Pawn, Actor and Collidable to read from their xml elements and initialise their variables.

Task 3: Write Level definition file and read in constructor

Similar to task 1 and 2, write an xml definition for Level and update its constructor to parse the xml into variables. Add a unit tag whose type attribute is the hero xml definition file name without extension, this will allow the unit factory to find the correct definition. The level constructor should take each unit tag and use an instance of the UnitFactory to create and add pawns to the vector.

Task 4: Set Hero's destination on mouse click

In the level's handleEvent() method, check if the event is a mouse click; if it is, then call setDestination() on mHero (which has been initialised from xml definition and added to the vector of pawns), passing the mouse position as the argument.
The hero should now be moveable.

Basic enemy

Task 1: Write basic enemy xml definition

Similar to the hero definition, write an xml definition file for a basic enemy.

Task 2: Add enemy to level xml definition

Similar to the hero, add the basic enemy to the xml file for the level.
There should now be a basic enemy in the game that the player can fight.

Tower

Task 1: Write Projectile class

Projectile as abstract and inherits publicly from Actor. It has two private variables: mDamage (const int) and mDamageType (const Damage::Type). It has four protected variables: mActive (bool), mVelocity (sf::Vector2f), mTimeToLive (float), mImpactOccurred (bool). It needs to defined the following seven public methods:

- update(elapsedTime : sf::Time const&) : void
  - Pure virtual.
- isActive() : bool
- impactOccurred() : bool
- onCollide(other : Collidable*, mtv : sf::Vector2f)
  - Sets impactOccurred to true
- getDamage() : int
- getDamageType() : Damage::Type
- fire(from : sf::Vector2f const&, to : sf::Vector2f const&, ttl : float) : void
  - Pure virtual.

Task 2: Write ArcProjectile class

ArcProjectile holds a static constant sfml vector *mGravity* (0, 100). This projectile implementation overrides both the fire and update methods of Projectile:

- fire(from : sf::Vector2f const&, to : sf::Vector2f const&, ttl : float) : void
  - Sets mTimeToLive to match ttl parameter.
  - Sets mActive to true.
  - Calculates mVelocity so that the projectile will reach the target under the influence of mGravity when mTimeToLive reaches zero.
- update(elapsedTime : sf::Time const&) : void
  - If mActive is true:
    - Applies mGravity (by elapsed time in seconds) to mVelocity.
    - Moves the position by mVelocity (by elapsed time in seconds)

Task 3: Write ProjectileManager class

Holds a list of Projectile* (*mProjectiles*) and has the following public methods:

- give(projectilePtr : Projectile*) : void
  - Adds projectilePtr to the mProjectiles.
  - ProjectileManager gains ownership of projectile and accepts responsibility of managing its memory.
- update(elapsedTime : sf::Time const&) : void
  - Calls update() on all managed projectiles.
- draw(target : sf::RenderTarget&) : void
  - Calls draw() on all managed projectiles.

Task 4: Write Tower class

Write the base tower class, which publicly inherits from Actor. Tower has three protected member variables: *mSecondsPerAttack* (const float), *mSecondsSinceLastAttack* (float) and *mCost* (const int). The constructor accepts two arguments: position (sf::Vector2f const&) and the xml element that defines the tower. Tower has the following three public methods:

- update(elapsedTime : sf::Time const&) : void
  - Virtual.
  - Increases mSecondsSinceLastAttack by the elapsedTime as seconds.
- shoot(targetList : std:list<Pawn*>&)
  - Pure virtual.
- getCost() : int

Task 5: Write ProjectileTower class

ProjectileTower publicly inherits from Tower. It takes the same arguments in its constructor. It holds a pointer to a ProjectileManager in addition to mRange (float) and mProjectileSpawnOffset (sf::Vector2f, the offset from the tower's position at which to spawn projectiles). It defines the following two public methods:

- setProjectileManager(projectileMgr : ProjectileManager*) : void
- shoot(possibleTargets : std::list<Pawn*>&)

- If mSecondsSinceAttack is greater than mSecondsPerAttack:
    - Iterate through possibleTargets, if an ENEMY Pawn is found and it is within mRange distance of the tower, create an ArcProjectile and fire it at the unit.
    - If the Pawn is MARCHING, fire at where the Pawn will be when the projectile lands.

Task 6: Implement Tower-Terrain collision

In order to restrict the placement of towers as little as possible, the only place the player cannot place a tower is on the path of the enemies. Therefore, we'll use a quadtree of bitfields, where the value of each node is the terrain at that node. We'll need a basic quadtree template class to accomplish this. The quadtree must expose the following methods:

- subdivide(predicate : std::function<bool(QuadTree<T>*)
    - Takes a predicate and subdivides each node of the tree if the predicate returns true for that node.
- query(x : int, y : int, dataOut : T&)
    - Returns the data of the leaf node under the position (x,y).

We will need a TerrainInterpreter class. The TerrainInterpreter must store an sf::Image (whose relative file system path will be passed to the constructor). The TerrainInterpreter has two static constant unsigned chars: GRASS and PATH in addition to their corresponding constant static sf::Color fields: mGrassColour (pure green) and mPathColour (pure white). The class requires the following method for interpreting terrain:

- interpretArea(left : unsigned, top : unsigned, width : unsigned, height : unsigned) : unsigned
    - Interprets each pixel of the image inside the rectangle specified.
    - Returns the combination of all the bitflags from each pixel.

In the level class, add a Quadtree<unsigned char> member: mTerrainTree. In the constructor for Level, create a TerrainInterpreter from the terrain image specified in the level xml. Call subdivide() on mTerrainTree with the following predicate lambda:

- [interpreter&](node : Quadtree<unsigned char>*) : bool
    - Call setData() on the node, passing the returned value of interpreter.interpretArea() with the node's bounds.
    - Return true if the node's data contains both GRASS and PATH and if its level is less than 10.
    - Else return false.

The next thing we need is a TowerPlacer class. The constructor takes a pointer to a Quadtree<unsigned char>, which it will use to check for valid tower placements. The TowerPlacer hold a bool for if it is active (mIsActive) as well as a bool for if the current location is valid (mIsValid). It also needs an sfml Sprite (mOverlay) for graphical feedback and an sfml Shape (mMask) to represent the tower base when checking for valid terrain. It needs the following public methods:

- place() : Tower*
    - If active and current location is valid for tower placement, returns a valid tower, placed at the current location.
    - Else returns nullptr.

    - Sets mIsActive to false.
- update(mousePosition : sf::Vector2f const&) : void
    - If active, updates position to match mousePosition and calculates validity.
- activate() : void
    - Sets mIsActive to true.
- draw(target : sf::RenderTarget&) : void
    - If active, draws the tower overlay to the target.
- isActive() : bool

Add an instance for TowerPlacer to the Level class, it should be a private member.
Modify Level::handleEvent() so that mouse move events calls update() on the tower placer, mouse press events call place() on the tower placer (if it is active) and key down events where the key is 'T' call activate() on the tower placer. The tower returned from place() should be put into a list belonging the the Level if it's not a nullptr. If TowerPlacer::place() is called then don't call setDestination() for mHero. Add calls to update() and draw() so that all towers are updated and drawn respectively. Finally, as each tower is updated, call its shoot() method and pass the list of Pawn*s.
Towers should now be functional and placeable.

Hero combat

Task 1: Implement SAT collision between units

Write colliision::CollisionGroup class which must hold a list of collision::Collidable*s. The class requires the following public methods:

- check(deltaTime : float) : void
    - Checks all Collidables against each other and calls onCollide on all colliding pairs.
    - Uses AABBs for broad phase.
    - Uses SAT for narrow phase.
- add(entry : Collidable*) : void
- checkAgainst(other : Collidable*) : void
    - Checks *other* against all members of the group without adding *other* to the group. Useful for projectiles later.

Add an instance of CollisionGroup to Level and add all Pawns to the group upon creation in the constructor. Call check() on the group in the Level's update().
The hero and other pawns should now fight each other when they collide.

Mage Tower

Task 1: Write FancyProjectile class

FancyProjectiles inherit publicly from Projectile. They keep track of their target and home in on its position. They also need to emit a trail of particles as they travel. It overrides Projectile::fire() so that the projectile's initial velocity shoots it away from its target. It overrides Projectile::update() so that it turns toward its target over time.

Task 2: Write MageTower class

MageTower inherits publicly from ProjectileTower. It overrides ProjectileTower::shoot() so that it fires FancyProjectiles instead of ArcProjectiles. It can also should a number of projectiles at a time and at different enemies.

Task 3: Modify TowerPlacer

Add the TowerType enum to TowerPlacer. The activate() method needs to accept a TowerType as its argument and store it. When place() is called, the placer needs to try and place the corresponding tower to the TowerType.

Task 4: Modify Level::handleEvent()

When the 'Y' key is pressed, activate the tower placer with TowerType::MAGIC. When the 'T' key is pressed, the tower placer should be activated with TowerType::ARROW.
Both Arrow and Mage towers should now be functional and placeable.

Level Path

Task 1: Write Path and Node classes.

Write the Path class. The Path class implements a simple singly-linked list of Nodes. The Path's begin() method returns a pointer to the first Node in the list.

Task 2: Write Minion class

Minion inherits from Pawn. It follows a path given to it via setPath() and flocks (sans alignment) with other members of its flock (flock is set by addToFlock()). Override Pawn::update() to follow the path waypoints then call the Pawn's update().

Task 3: Change non-hero unit Pawn instances to Minions.

Change UnitFactory to produce Minions.

Unit tower

Task 1: Write UnitTower class.

UnitTower inherits publicly from Tower. Instead of firing projectiles, it creates PLAYER faction units and adds them to the game. The UnitTower will need to hold a UnitFactory to produce its units.

Task 2: Modify TowerPlacer

Add UNIT to the TowerType enum. In Level::handleEvent(), pressing U should activate the TowerPlacer with TowerType::UNIT.