Map Generation

Direction Enum

NORTH,
EAST,
SOUTH,
WEST.

getOpposite(Direction Enum) returns South/North if North/South or East/West is West/East

Room Object

Private Room north,south,east,west
Private RoomTemplate layout
Private EnemyTemplate enemies
Private Item reward
Private int[1] Coordinates

Public Room(RoomTemplate layout, EnemyTemplate enemies, Item reward) returns Room object

setRoom(direction enum, Room Object)
getRoom(direction enum) returns Room Object
getAvailableDirections() returns List<Direction>
getCoords() returns Coordinates[]

Depth Based Generation (Basic)

Used on Room Object as generateFrom(new rootRoom)
Start with root room (x=0,y=0).
While generated rooms < room capacity
    //Identify number of available rooms using room map list
        Check for room (x+1,y), (x,y+1), (x-1,y), (x,y-1)
            From 0 to (Random number between 1 and (available rooms)):
                Pick unused direction
                Create new Room(choose random layout and enemies)
                generateFrom(newRoom)
                currentRoom.setRoom(direction)
            End loop
End loop

There needs to eventually be a version of this with any required rooms implemented. Boss room with end generation at that room, so perhaps generate a boss room after a minimum depth of 2 or 3? Items rooms should exist, a minimum of 1 per floor, and a maximum of 2 (changeable).

Room templates soon.

In theory the main difference between this and keyed generation will be that decisions made by the generator aren't random, and are instead based on a key, but devising the logic to base and generate from a key is something I haven't done before

Keyed Generation Algorithm

Enumerate decisions that need to be made:
For every floor:
- For every room:
  - How many connecting rooms should be made (up to 4)
  - Which template to use (some x amount, definitely a maximum of 32)
  - Which enemy template to use (some y amount, probably a maximum of 32 as well)
  - Which reward should drop (number of common and uncommon items)
- Which items should spawn in item rooms (total number of uncommon to rare items)
- Which rooms will be item rooms
- Which rooms with be boss rooms

Items

StatEffects Interface

Private EffectType type

getEffectType() returns EffectType type
setEffectType(Type enum)

StatEffectsInt Object implements StatEffects

Private EffectType type
Private int value

getEffectType() returns EffectType type
setEffectType(Type enum)
getValue() returns int value
setValue(int value)

StatEffectsString Object implements StatEffects

Private EffectType type
Private String value

getEffectType() returns EffectType type
setEffectType(Type enum)
getValue() returns String value
setValue(String value)

Item Object implements Entity

Private int id
Private String name
Private EntityType type
Private String spriteFile

Private StatEffects[] buffs
Private String description

Public Item(String name, StatEffects[])
getName() returns name
setName(String name)
getStatEffects() returns StatEffects
setStatEffects(StatEffects[] buffs)

Item Decomposition

Items will give some key components, which will be a raw numerical/discrete value (likely as a String) and %values. The final effect of the stat change will be the raw value multiplied by the %value. A discrete stat change could be something like an on-hit effect, with some %chance of happening, or a form that player takes, for example having equidistant or parallel weapons. There must be a list of possible effects/stats a user can have.
Stats that can change:

- Discrete [Possible Values] *multiple possible:
  - Player On-Hit: [Poison/Frost/Fire/Vampiric/Thorns]*
    - A list of [effect,%chance] values
  - Turret Form: [Parallel/Equidistant]
  - Turret Type: [Basic/Piercing/Shotgun]
  - Turret On-Hit: [Poison/Frost/Fire/Vampiric]*
    - A list of [effect,%chance] values

- Continuous (Range of values):
  - Maximum Health (1-15) - Measured as quarter hearts - Integer only
  - Damage (1 - 15) - Measured as quarter hearts - Integer only
  - Rate of Fire (0.1 - 10) - Measured as projectiles per second
  - Projectile Speed (0.5 - 10) - Measured as tiles per second
  - Movement Speed (0.5 - 10) - Measured as tiles per second
  - Rotation Speed (10 - 180) - Measured as degrees per second

All stats will be stored as their value and %value terms, and then re-calculated when either of these stats are affected. This means rounding each time does not affect the long term effect of %value increases. Every item that has effects will have a combination of the above, making the number of methods I have to write to handle a potentially infinite number of items minimal.

Projectiles will likely work like visitors, where colliding with another entity that has health will cause it to take damage by visiting the object, and applying its damage calculation. Any object this projectile hits should be visited. If the projectile is not a piercing projectile, it should self-destruct upon first collision.

Entities

Entity Interface

Private int id
Private String name
Private EntityType type
Private String spriteFile

getId() returns id
setId(int id)
getName() returns name
setName(String name)
getType() returns type
setType(EntityType type)
getSprite() returns spriteFile
setSprite(String spriteFIle)