Class Write-up

Class 1: **ActionMap**

1. Class Description: This class serves as a command registry that maps string action names (like “Jump”, “UsePotion”, “OpenChest”, etc.) to callable functions. Main goal of the class is to let other parts of the game query what actions exist and then trigger an action by name without hard-coding switch statements everywhere.

   High-level behavior would include:
   a. Being able to register actions under a string key
   b. Trigger an action by providing its name and potential arguments
   c. Listing available actions
   d. Safe and predictable behavior for missing, overwritten, or invalid actions

2. Similar Classes:
   a. std::map/std::unordered_map – storing the name and function mapping
   b. std::function – type-erased callable wrapper
   c. std::string – action keys
   d. std::vector – listing action names
3. Key functions:
   bool AddFunction(std::string name, std::function<void()> fn)
      Adds a new action mapping its name with a function callable.
   bool RemoveFunction(std::string name)
      Removes an action from the map
   bool Trigger(std::string name)
      Calls the action if present
   std::vector<std::string> ListActions()
      Lists all the actions

   Above are the functions that will likely be the core of the class. Other functions likely to be implemented are Size(), Clear(), and Rename(std::string oldName, std::string newName). Depending on the direction of the game, descriptions could also be added to actions. This could introduce functions like SetDescription(std::string name, std::string description) and GetDescription(std::string name).

4. Error Conditions:
   Programmer Errors (asserts) – AddFunction(name, fn) where fn is empty or there is an invalid name format like an empty string.

   Recoverable Errors (exceptions) – Memory allocation failure during insert / copy

   User/Runtime Errors (return special condition) – Trigger(name) when action doesn’t exist, AddFunction(name, fn) when the name already exists, RemoveFunction(name) when action being removed doesn't exist, Rename(old, new) when the old is missing or the new already exists.

5. Challenges: This class is relatively straightforward in terms of implementation. The main challenge with this class will be keeping the interface clean and flexible while avoiding overengineering, since the exact company project and gameplay requirements have not yet been decided.
6. Other Groups Classes: ActionMap will likely most coordinate with the GUI and Web interface modules. These interfaces can use ActionMap as the backing logic for menus, buttons, and other interactive controls by querying available actions at runtime and triggering them by name when a user interacts with the interface. This class may also coordinate with classes from the Interactive World group particularly with DataMap or FunctionSet, if parameterized actions are introduced in development. Finally, within our own Classic Agents module, ActionMap can integrate with the BehaviorTree class by providing a centralized mechanism for triggering named actions from leaf nodes.

Class 2: **BehaviorTree**

1. Class Description: This system provides the ability to build complex agent decision-making logic with mainly reusable components. The behavior tree itself will have nodes arranged in a hierarchy that are ticked constantly. For memory the tree will have a "blackboard" that allows nodes to read and write their state during execution

   It is made out of three different types of nodes:
   a. Composite nodes - these control the flow of the tree I.E. Sequence and Selector
   b. Decorator nodes - which modifies the behavior of a child node I.E. Inverter
   c. Leaf nodes - these represents the actions the agent might take and/or conditions that need to be met to take those actions

2. Similar Classes:
   Std::function - represents actions and conditions for the leaf nodes
   Std::unordered_map - for blackboard memory data
   Std::vector - for holding the child nodes
   Std::shared_ptr & std::unique_ptr - for node ownership
3. Key functions: The key functions depend on what part of the tree is being used
   BehaviorTree
      Void Tick() - Ticks the root for each decision
      Void Reset() - Resets tree to initial Node
   Composite Nodes
      AddChild(std::shared_ptr<Node>)
      Sequence::Tick("Node")
      Selector::Tick("Node")
   Decorator Nodes
      Void SetChild(std::shared_ptr<Node>)
         Inverter::Tick("Node")

Leaf Nodes

ActionNode()

ConditionNode()

4. Error Conditions:
   Programmer Error - Ticking the tree with a root node, and the potential of adding null child nodes to composites. This can be solved with asserts

   Recoverable Errors - Memory allocation, infinite loops and so on with the tree structure. Can be handled with std::bad_alloc

   User/Logic Errors - Action/Condition returns an invalid state. Could just return failure or just a default value that the parent node will handle.

5. Challenges:The biggest challenge would be how tightly coupled the tree should be with the other systems, if the current design of it with the goal being lightweight and a generalized usage backfires and causes issues in the future it won't be efficient for anyone. So we need to decide how much of the other groups resources do we really need to produce our classes
6. Other Groups Classes: As of right now the BehaviorTree will use a majority of Group2's own classes and setup. This is due to the behavior tree being more of a system structure with set conditions and actions that will be dependent on the info that is already known.

Class 3: **WorldPath**

1. Class Description: WorldPath stores an ordered sequence of 2D points representing an agent's planned movement through the world. It is meant to be a simple, reusable path container that other systems can create, inspect, and follow. It supports fast access to path information for agent movement and debugging, including total path length, detecting whether the path crosses itself, and computing a useful derived statistic such as which two points are furthest apart. It may optionally provide a short debug summary to help during development.

2. Similar Classes:
   std::vector (ordered storage)

   std::optional (no-result returns)

   std::pair (return index pairs)

   / (derived computations)

3. Key functions:
   AddPoint(Point p)

   Adds a new point to the end of the path. This is the main way a path is built up (from a generator, a planner, or during debugging).

Length() const

Returns the total path distance by summing distances between consecutive points. Used for quick evaluation (ex: “is this path too long?”).

SelfIntersects() const

Checks whether any segment of the path crosses another segment. Useful for debugging and for preventing weird “looping” movement if the game design cares about that. This is one of the more algorithm-heavy parts.

FurthestPair() const

Returns the two points in the path that are farthest apart (as indices or as a pair of points). Useful as a derived statistic for analysis/debugging, and can help approximate the “span” of a path. Return std::nullopt if there are fewer than 2 points.

4. Error Conditions:
   operator[] out of range
   Programmer error. Assert in debug builds (or throw std::out_of_range if the project uses exceptions).

   Empty / too-small path queries
   Runtime condition. Return std::nullopt for queries that require at least 1–2 points (ex: Start/End on empty, FurthestPair when size < 2).

5. Challenges: Implementing robust 2D segment intersection for SelfIntersects(), especially collinear segments and deciding whether touching at endpoints counts as intersection. If coordinates are floating point, selecting comparison tolerances. Avoiding unnecessary recomputation for expensive queries while keeping common queries like Length() fast. Confirming the project’s point/position type and units so stored points match the rest of the codebase.

6. Other Groups Classes: PathGenerator (or any pathfinding/path creation module) should output a WorldPath so agents can consume a consistent path format. WorldGrid/StateGrid (or equivalent) defines the coordinate system, valid tiles, and movement constraints that produced the path. Agent movement/path-following logic should consume WorldPath through an agreed interface, and debugging/visualization tools can read WorldPath to render or log paths.

Class 4: **PathGenerator – Logan Rimarcik**

1. Class Description: This class needs to take a set of parameters (a PathRequest-like object) that describes – a list of tile positions (*start, middle$_1$,...,middle$_n$, goal),* which *walkable tiles to avoid, etc* and an AgentAbility-type class which is a set of parameters if the agent can *climb, swim/surf, go through specific tiles…* to know if the path can be made. It also has a *const reference* to a *World* class or to a *WorldGrid* type class that holds a list of tiles in memory with positions. Most/all of its functions will return a WorldPath class. This class has various functions depending on what is needed for path traversal: radial/circular, manhattan, shortest, and more. WorldGrid will need to have a set of more parameters based on what movement options are possible in this particular world. (certain spaces like a puzzle space – i.e. traveling through a dungeon with a locked door so you need to play a minigame to open it and you are your character but need to walk through interacting with objects to solve it and you can't travel diagonally here).

2. Similar Classes: Nothing crazy but *std::optional* to return path/not path could be used. I'll end up using a priority queue for A* search as well then vectors throughout and std::maps where possible to make something more efficient

3. Key functions:
    - PathRequest: struct (maybe class) that contains *start, end, middle points, tiles to avoid, type (enum: Circular, manhattan, shortest, circular_manhattan) const AgentAbility&, float circular radius)*
    - Static std::optional<WorldPath> CreateSimplePath(const PathRequest& request, const WorldGrid& world_grid)
    - Then the rest are probably private functions dedicated to generatingManhattan, circular, shortest, circularManhattan
    - The highlighted YELLOW portion is 'optional' – it depends on the game though many could use it and some may not. It was described in the description why it is needed
    - Note: Patrol path is made by having a behavior tree to loop and create the simple path a middle point and probably manhattan if moving in L shape or just a line then two points gives you a path.
4. Error Conditions:

    - Error condition 1 (User Error): It will return a bad path (std::optional) if the path can't be made to the end point because of some obstacles or a distance is too great (imagine there is a chasm then has to prevent making a mile long path) – the distance max for creation is just some static constexpr variable
    - Error Condition 2 (Programmer Error) - Programming a circular path without a radius ⇒ assert statement or default value.
    -
5. Challenges:

    - NEED a function somewhere that takes the returned path from this class and helps agent updating to follow the path – given a path = list of points then with

information of character movement speed then calculate where to be after 't' milliseconds
- Need a lot of supporting classes like A* and knowledge of world map, character movement abilities, etc. → Hard to know currently because we still haven't decided our project
- I don't really need to learn about other things. I think I can do this and leave room to modify it when we know our project.

6. Other Groups Classes:
    - WorldPath - path information and how our characters will eventually start moving and such
    - WorldGrid → Need to know CellTypes, If this world is possible to move diagonally or not – can have multiple worlds and some worlds with diagonal vs nondiagonal (see description)
    - StateGrid - need to know the properties and what is navigable by the agent
    - Challenge/Other: Need to know if we can move diagonally or not. This dictates whether I make a static constexpr step size and return checkpoint positions for the character to move along or I return a list of tiles directly where the character just follows the tiles. There is probably a way to generalize both but I'd just be nice to know.
        - Or Bresenham's line algorithm for 8 directional movement

Comments: Class Description Comment: (I'm not sure if I should separate out each of the enum values: shortest, manhattan, radial/circular. Different ones may require different params so maybe space it out and decide higher up)

Class 5: **MemoFunction**

1. Class Description: This class is designed to speed up performance through memoization, preventing redundant function calls. It works as a function wrapper, storing the function to be run and an initially empty cache of outputs. When a function is called via MemoFunction, it searches the cache, with the key being the input for the function. If not found, the function then runs, returning the output as well as storing it within the cache. If found, it skips running the function altogether and simply returns the stored value associated with the input.

2. Similar Classes:
    - std::unordered_map: to be used as the cache (fast search and insert)
    - std::function: easy to use wrapper

3. Key functions:
    - returntype Memoize(const inputtype & input): The main function, which performs the actions described in the class description (likely needs a template for the types).

- The most important function I can think of is the one above, though more will likely need to be implemented for the sake of managing the cache and allowing certain input types to be hashed.

4. Error Conditions:
    - Programmer error:
        Not providing workaround ways for an unhashable input type to hash
    - Recoverable error:
        Memory error relating to growth of cache
    - User error:
        Providing invalid inputs to the wrapped function

5. Challenges:
    - Ensuring that the class works stably universally. This means making sure all the possible input types (including multiple inputs) for every possible function perform the function as expected, with the output being both stored and able to be accessed.
    - Handling the memory used by the cache and making sure it does not grow too large, is properly released, etc.

6. Other Group's Classes: Unlikely to need coordination with other group's classes, though I'm not completely sure of the scope MemoFunction will be used in. For the time being, I'm assuming it will just be used within Group 2 to memorize repeated function calls.

Vision for the Main Module:

Our overall vision for this module is all the basic systems that can be put in place for the main character and monster/npc characters to exist, move about, and interact. More specifically, we create agents being monsters, characters, npcs or others that have a specific behavior tree targeted to their class in which they can scan for players, move in a set WorldPath, or any other movement pattern. These agents will have stats, abilities, or any other necessary features relating to the game allowing them to interact with their environment such as abilities to climb and swim given certain unlocks acquired through the game. This module can be applied to any moving interacting object and not specifically the player or npcs as they may not exist when the group project gets decided.