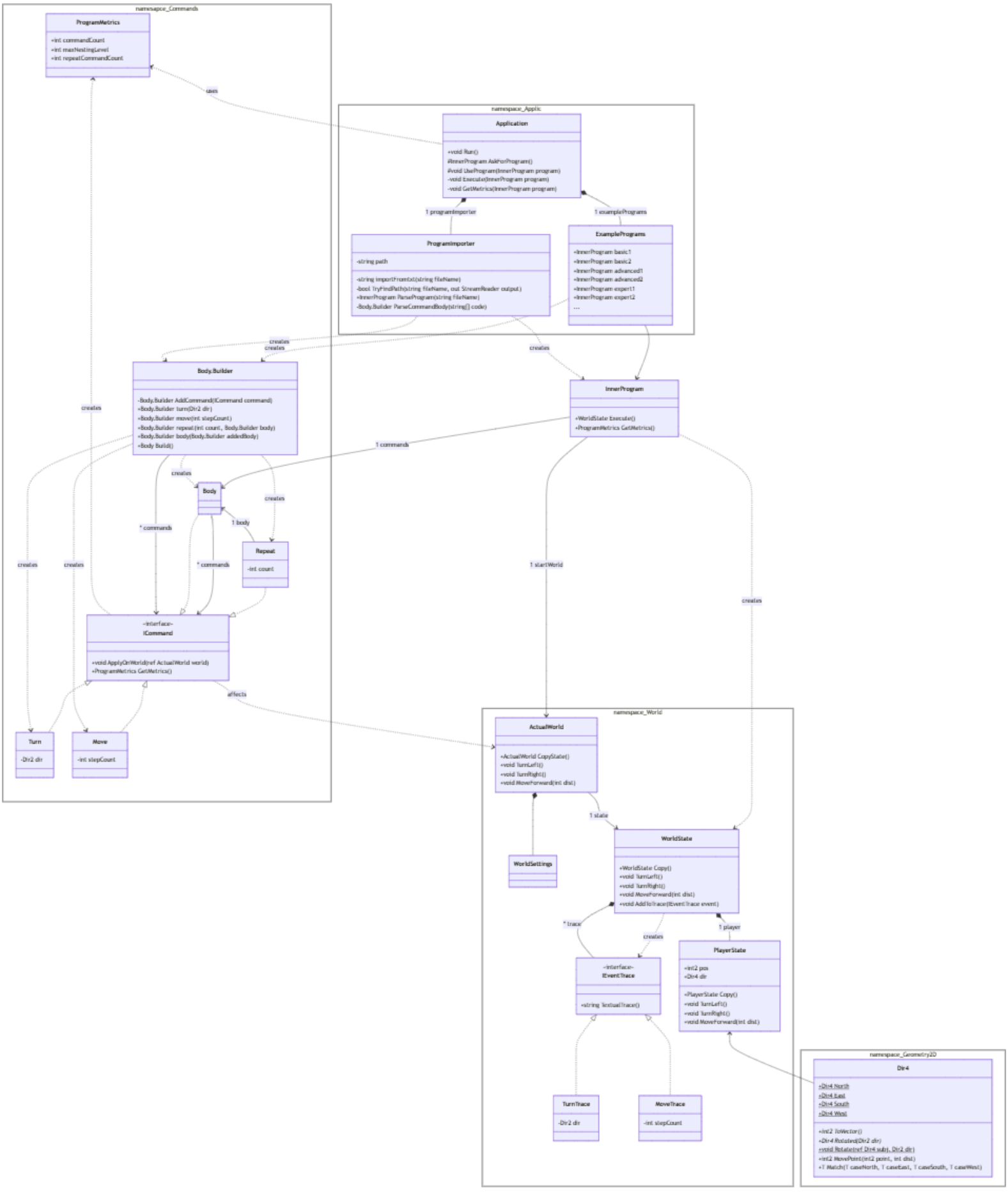
MSO assignment 2, Abracha Koens (9995153) and Aron Hopman (0296163)

Part 1 Software design and patterns

UML diagram:



Mermaid code in blue:

```
classDiagram
  %%region Application
    namespace namespace_Applic{
      class Application{
        +void Run()
        #InnerProgram AskForProgram()
        #void UseProgram(InnerProgram program)
        -void Execute(InnerProgram program)
        -void GetMetrics(InnerProgram program)
      }
      class ProgramImporter{
        -string path
        -string importFromtxt(string fileName)
        -bool TryFindPath(string fileName, out StreamReader output)
        +InnerProgram ParseProgram(string fileName)
        -Body.Builder ParseCommandBody(string[] code)
      }
      class ExamplePrograms{
        +InnerProgram basic1
        +InnerProgram basic2
        +InnerProgram advanced1
        +InnerProgram advanced2
        +InnerProgram expert1
        +InnerProgram expert2
        ...
      }
    }
    Application *-- ProgramImporter : 1 programImporter
    ProgramImporter ..> InnerProgram : creates
    Application *-- ExamplePrograms : 1 examplePrograms
    ExamplePrograms --> InnerProgram
    ProgramMetrics <.. Application : uses
  %%

  ProgramImporter ..> Body.Builder : creates
  ExamplePrograms ..> Body.Builder : creates

  %%region Commands
    namespace namesapce_Commands{
      class ProgramMetrics{
        +int commandCount
        +int maxNestingLevel
        +int repeatCommandCount
      }
      class Body.Builder{
        -Body.Builder AddCommand(ICommand command)
        +Body.Builder turn(Dir2 dir)
        +Body.Builder move(int stepCount)
        +Body.Builder repeat(int count, Body.Builder body)
        +Body.Builder body(Body.Builder addedBody)
        +Body Build()
      }
      class ICommand{
        +void ApplyOnWorld(ref ActualWorld world)
        +ProgramMetrics GetMetrics()
      }
      class Body
      class Repeat{
```

```
        -int count
      }
      class Turn{
        -Dir2 dir
      }
      class Move{
        -int stepCount
      }
    }
    <<interface>> ICommand
    ProgramMetrics <.. ICommand : creates
    Body <-- Repeat : 1 body
    Body.Builder ..> Body : creates
    Body.Builder --> ICommand : * commands
    Body.Builder ..> Repeat : creates
    Body.Builder ..> Turn : creates
    Body.Builder ..> Move : creates
    Body ..|> ICommand
    Body --> ICommand : * commands
    ICommand <|.. Turn
    ICommand <|.. Move
    Repeat ..|> ICommand
%%

ICommand ..> ActualWorld : affects

class InnerProgram{
    +WorldState Execute()
    +ProgramMetrics GetMetrics()
}
InnerProgram --> Body : 1 commands
InnerProgram --> ActualWorld : 1 startWorld
%%ProgramMetrics <.. InnerProgram

%%region World
    namespace namespace_World{
      class ActualWorld{
        +ActualWorld CopyState()
        +void TurnLeft()
        +void TurnRight()
        +void MoveForward(int dist)
      }
      class WorldSettings{
        %% Data that can't change while the Innerprogram is running
      }
      class WorldState{
        %% Data that can be changed by the program
        +WorldState Copy()
        +void TurnLeft()
        +void TurnRight()
        +void MoveForward(int dist)
        +void AddToTrace(IEventTrace event)
      }
      class PlayerState{
        +int2 pos
        +Dir4 dir
        +PlayerState Copy()
        +void TurnLeft()
        +void TurnRight()
        +void MoveForward(int dist)
```

```
    }
    class IEventTrace{
      +string TextualTrace()
    }
    class TurnTrace{
      -Dir2 dir
    }
    class MoveTrace{
      -int stepCount
    }
  }
  <<interface>> IEventTrace
  ActualWorld --> WorldState : 1 state
  ActualWorld *-- WorldSettings
  WorldState *-- PlayerState : 1 player
  WorldState *-- IEventTrace : * trace
  WorldState ..> IEventTrace : creates
  IEventTrace <|.. TurnTrace
  IEventTrace <|.. MoveTrace
%%

InnerProgram ..> WorldState : creates

%%region Geometry2D
  namespace namespace_Geometry2D{
    class Dir4{
      +Dir4 North $
      +Dir4 East $
      +Dir4 South $
      +Dir4 West $

      +int2 ToVector()*
      +Dir4 Rotated(Dir2 dir)*

      +void Rotate(ref Dir4 subj, Dir2 dir) $
      +int2 MovePoint(int2 point, int dist)
      +T Match<T>(T caseNorth, T caseEast, T caseSouth, T caseWest)
    }
  }
  PlayerState <-- Dir4
%%
```

We left Dir2 out of the UML diagram, because it's so basic and so unlikely to ever change, that we don't care how many classes depend on it. It would only make the UML look more messy.

Design patterns:

- We use the 'chain of responsibility'-pattern to pass commands through the world-classes to the player.
- Body uses the builder-pattern so classes can create Program-instances without needing to create all ICommand-instances themselves. This leads to looser coupling.
- Dir4 uses the state- and singleton-patterns, because there are only 4 directions. We'll never need a fifth.
- ExamplePrograms uses the singleton-pattern, because the programs it contains were supposed to be hard-coded, so we don't need multiple instances. They would only waste memory. We could save even more memory by only initializing the programs when asked by the user, but with our small set of programs that doesn't matter much.

Part 2 Evaluation

Likely future changes:

- A new type of command gets added, for example Face(x,y), which makes the player turn so that it faces a given point. To start, we can add this new command to ICommand's realizations and World's public methods. This wouldn't change anything to our program's behaviour yet. Then we add a method representing this command to Body.Builder and then we add it to ProgramImporter's Parse-function. Now we can create program files containing our new command.
- Walls get added to the world. These can be stored in the class WorldSettings, because the program doesn't affect the walls. World's MoveForward-method must be adapted so that it lets walls block player movement.
- We might want to be able to import programs from different formats. We could do that by making ProgramImporter abstract and creating different types of ProgramImporter, one for each format.
- Hard change: If the user asks for a program and makes a small typo, like 'exper2', the program should guess what the user meant to type. The way we implemented the program search, only exact matches are detected and the list of possible matches we stored would be way too big if we accounted for typos. If we want this feature, we should completely change the way we read input strings, using more logic.

High cohesion:

- We introduced interface IEventTrace where we could have reused ICommand. Now not every type of command needs to be traceable.
- We separated WorldState and WorldSettings, to remind the programmers which features of the world can be changed by a program and which can't.
- We introduced classes ProgramImporter and ExamplePrograms, because the code would have been more messy if we put all that data in the Application-class.

Low coupling:

- We introduced interface IEventTrace where we could have reused ICommand and we connected them only through methods in World, so the world-classes could depend less on ICommand.
- Body uses the builder-pattern so classes can use it to create Program-instances without needing to create all ICommand-instances themselves.

Part 4 Work distribution & retrospective

Task distribution:

Part 1: Aron.

Part 2: Aron.

Part 3:

Mostly Abracha: class ProgramImporter, folder ExampleFiles.

Mostly Aron: namespace Command, namespace World, class ExamplePrograms.

Part 4: Mostly Aron

What went well:

There was no disagreement on the way we would structure our program.

What could have been better:

One of us worked faster than the other, so it was hard for the other to keep up. We should have distributed the tasks more clearly, so we could both work on our own parts at our own time without worrying about causing conflicts by working on the same part.