MSO assignment 3, Abracha Koens (9995153) and Aron Hopman (0296163)

# **Part 1 Extending and adapting the software design**

Full UML (old version) in mermaid code here in blue:

  classDiagram

    %%region UI

        namespace namesapce\_GenericUI{

            class UI1{

                %% We'll probably have to split this class up for higher cohesion.

                +void ClickRun()

                +void ClickMetrics()

                -string ReadTextBoxProgram()

                ...()

            }

            class IDataBridge{

                +void SetTextBoxProgram(string text)

                +void SetTextBoxOutput(string text)

                +string ReadTextBoxProgram()

                ...()

            }

        }

        UI1 \*-- ProgramImporter : 1 programImporter

        UI1 \*-- ExamplePrograms : 1 examplePrograms

        IDataBridge <-- UI1 : 1 dataBridge

    %%

    %%region Forms

        namespace namespace\_MSO\_P3\_Forms{

            class Form1{

            }

            class Form1.DataBridge

        }

        <<partial>> Form1

        Form1 \*-- UI1

        Form1.DataBridge \*-- Form1

        Form1.DataBridge ..|> IDataBridge

    %%

    %%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 Bodied

            class Repeat{

                -int count

            }

            class RepeatUntil

            class If

            class ICondition{

                +Check(ActualWorld world)

            }

            class Turn{

                -Dir2 dir

            }

            class Move{

                -int stepCount

            }

        }

        <<interface>> ICommand

        <<abstract>> Bodied

        ProgramMetrics <.. ICommand : creates

        RepeatUntil --> ICondition

        If --> ICondition

        Body <-- Bodied : 1 body

        Body.Builder ..> Body : creates

        Body.Builder --> ICommand : \* commands

        Body.Builder ..> Repeat : creates

        Body.Builder ..> Turn : creates

        Body.Builder ..> Move : creates

        Body.Builder ..> RepeatUntil : creates

        Body.Builder ..> If : creates

        Body ..|> ICommand

        Body --> ICommand : \* commands

        ICommand <|.. Turn

        ICommand <|.. Move

        Repeat ..|> Bodied

        RepeatUntil ..|> Bodied

        If ..|> Bodied

        Bodied ..|> 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

                -WorldCell[,] worldGrid

                +WorldCell GetCell(int2 pos)

            }

            class BlockException

            class LeftGridException

            class WorldCell{

                Empty

                Blocked

            }

            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

        <<enumeration>> WorldCell

        ActualWorld ..> BlockException

        ActualWorld ..> LeftGridException

        ActualWorld \*-- WorldSettings

        WorldSettings --> WorldCell

        ActualWorld --> WorldState : 1 state

        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

    %%

**Split into namespaces:**

Namespace Commands:

  %% namespace Commands

  classDiagram

    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 ICommandAlg["ICommand.IAlgebra < Result, Cond >"]{

            +Result turn(Dir2 dir)

            +Result move(int stepCount)

            +Result repeat(int count, Result foldedBody)

            +Result body(Result[] foldedCommands)

            +Result repeatUntil(Cond foldedCondition, Result foldedBody)?

            +Cond facingBlock()

            +Cond facingGridEdge()

            +Cond not(Cond foldedInput)

        }

        class Body

        class Bodied

        class Repeat{

            -int count

        }

        class RepeatUntil

        class If

        class Turn{

            -Dir2 dir

        }

        class Move{

            -int stepCount

        }

    }

    <<interface>> ICommand

    <<interface>> ICommandAlg

    <<abstract>> Bodied

    ProgramMetrics <.. ICommand : creates

    Body <-- Bodied : 1 body

    Body.Builder ..> Body : creates

    Body.Builder --> ICommand : \* commands

    Body.Builder ..> Repeat : creates

    Body.Builder ..> Turn : creates

    Body.Builder ..> Move : creates

    Body.Builder ..> RepeatUntil : creates

    Body.Builder ..> If : creates

    Body ..|> ICommand

    Body --> ICommand : \* commands

    ICommand <|.. Turn

    ICommand <|.. Move

    ICommand ..> ICommandAlg : uses

    Repeat ..|> Bodied

    RepeatUntil ..|> Bodied

    If ..|> Bodied

    Bodied ..|> ICommand

    namespace namesapce\_Commands.Condition{

        class ICondition{

            +bool Check(ActualWorld world)

            +Result Fold < Result > (ICondition.Algebra < Result > algebra)

        }

        class IConditionAlg["ICondition.Algebra < Result >"]{

            +Result foldFacingBlock;

            +Result foldFacingGridEdge;

            +Func < Result, Result > foldNot;

        }

        class FacingBlock

        class FacingGridEdge

        class Not

    }

    RepeatUntil --> ICondition

    If --> ICondition

    ICondition ..> IConditionAlg : uses

    ICondition <|.. FacingBlock

    ICondition <|.. FacingGridEdge

    ICondition <|.. Not

Namespace World:

  classDiagram

    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

            -WorldCell[,] worldGrid

            +WorldCell GetCell(int2 pos)

        }

        class BlockException

        class LeftGridException

        class WorldCell{

            Empty

            Blocked

        }

        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

    <<enumeration>> WorldCell

    ActualWorld ..> BlockException

    ActualWorld ..> LeftGridException

    ActualWorld \*-- WorldSettings

    WorldSettings --> WorldCell

    ActualWorld --> WorldState : 1 state

    WorldState \*-- PlayerState : 1 player

    WorldState \*-- IEventTrace : \* trace

    WorldState ..> IEventTrace : creates

    IEventTrace <|.. TurnTrace

    IEventTrace <|.. MoveTrace

Namespace Geometry2D:

classDiagram

    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)

        }

    }

Namespace GenericUI:

classDiagram

    namespace namesapce\_GenericUI{

        class UI1{

            %% We'll probably have to split this class up for higher cohesion.

            +void ClickRun()

            +void ClickMetrics()

            -string ReadTextBoxProgram()

            ...()

        }

        class IDataBridge{

            +void SetTextBoxProgram(string text)

            +void SetTextBoxOutput(string text)

            +string ReadTextBoxProgram()

            ...()

        }

    }

    IDataBridge <-- UI1 : 1 dataBridge

Namespace Applic:

classDiagram

    namespace namespace\_Applic{

        class Application{

            +void Run()

            #InnerProgram AskForProgram()

            #void UseProgram(InnerProgram program)

            -void Execute(InnerProgram program)

            -void GetMetrics(InnerProgram program)

        }

        class ProgramImporter{

            -string codeFolderPath

            -string importFromtxt(string fileName)

            -bool TryFindPath(string fileName, out StreamReader output)

            +InnerProgram ImportProgram(string fileName)

        }

        class ExamplePrograms{

            +InnerProgram basic1

            +InnerProgram basic2

            +InnerProgram advanced1

            +InnerProgram advanced2

            +InnerProgram expert1

            +InnerProgram expert2

            ...

        }

    }

    Application \*-- ProgramImporter : 1 programImporter

    Application \*-- ExamplePrograms : 1 examplePrograms

Namespace MSO\_P3\_Forms:

classDiagram

    namespace namespace\_MSO\_P3\_Forms{

        class Form1{

        }

        class Form1.DataBridge

    }

    <<partial>> Form1

    Form1.DataBridge \*-- Form1

Dependencies Between namespaces:

  classDiagram

    %%region UI

        namespace namesapce\_GenericUI{

            class UI1

            class IDataBridge

            class ProgramParser

            class ProgramParser.BodyUnparser1

        }

        UI1 \*-- ProgramImporter : 1 programImporter

        UI1 \*-- ExamplePrograms : 1 examplePrograms

        ProgramParser ..> Body.Builder : creates

        ProgramParser.BodyUnparser1 ..|> ICommand.IAlgebra

    %%

    %%region Forms

        namespace namespace\_MSO\_P3\_Forms{

            class Form1

            class Form1.DataBridge

        }

        <<partial>> Form1

        Form1 \*-- UI1

        Form1.DataBridge ..|> IDataBridge

    %%

    %%region Application

        namespace namespace\_Applic{

            class Application

            class ProgramImporter

            class ExamplePrograms

        }

        ProgramImporter ..> InnerProgram : creates

        ExamplePrograms --> InnerProgram

        ProgramMetrics <.. Application : uses

    %%

    ProgramParser <-- ProgramImporter

    ProgramImporter ..> Body.Builder : creates

    ExamplePrograms ..> Body.Builder : creates

    %%region Commands

        namespace namesapce\_Commands{

            class Body.Builder

            class ProgramMetrics

            class ICommand

            class ICommand.IAlgebra

            class Body

            class ICondition

        }

        <<interface>> ICommand

    %%

    ICommand ..> ActualWorld : affects

    class InnerProgram{

        +WorldState Execute()

        +ProgramMetrics GetMetrics()

    }

    InnerProgram --> Body : 1 commands

    InnerProgram --> ActualWorld : 1 startWorld

    %%region World

        namespace namespace\_World{

            class ActualWorld

            class WorldState

            class PlayerState

        }

    %%

    InnerProgram ..> WorldState : creates

    %%region Geometry2D

        namespace namespace\_Geometry2D{

            class Dir4

        }

        PlayerState <-- Dir4

    %%

Design patterns:

* The use of UI1 and IDataBridge looks like a design pattern to me, but I don’t yet know which one. Maybe facade or mediator.
* ICommand and ICondition use a fold and algebra. This is a pattern taught in the course ‘languages and compilers’. Maybe it counts as a design pattern here.

Deviations from practical 2 design:

* See ‘Refactoring examples’ in part 2 for some of these deviations.
* Extracted abstract class ‘Bodied’ from the repeat-command, because there are now multiple command types that use a body in a similar way.
* …

# **Part 2 Implementation and code quality**

Measures taken:

* …

Refactoring examples:

* Use of class Bodied. …
* Use of IOutputLanguage. …
* Extracted the ProgramParser-class for ProgramImporter and split its parse-function into multiple parse-functions, one for each command-type, for higher cohesion.
* Created IOutputLanguage, so the programmer has more freedom in where and how to display textual output. It no longer needs to be by the console, and it no longer needs to use the exact sentence structure that used to be hard-coded in the old methods.

Changes inspired by metrics:

* …

# **Part 3 Evaluation**

Likely future changes:

* …

High cohesion:

* …
* ICommand now has method Fold, so not every method that needs to distinguish between different types of ICommand-realizations needs to be inside ICommand anymore. Otherwise, ICommand could get too many different responsibilities.

Low coupling:

* Every action of every UI-element in the form is directly connected to a method of class UI1, so our model handles basically every somewhat complex responsibility. This makes it easy to replace the form with any other type of UI, as long as that other UI has similar elements.
* …

# **Part 4 Testing**

Test fails + reasons:

* …

# **Part 5 Work distribution & retrospective**

Task distribution:

Part 1: …

Part 2: …

Part 3: ...

Part 4: …

What went well:

…

What could have been better:

…

What we learned:

…