# SOMAS Base Platform User Manual

Matthew Scott
matthew.scott18@ic.ac.uk

Ana Dimoska
ana.dimoska21@imperial.ac.uk

Jeremy Pitt
j.pitt@imperial.ac.uk

September 2023

# Contents


**1 Introduction** **1**

**2 Implementation** **2**
2.1 *baseagent* Package . . . 2
2.1.1 Composition Structure . . . 2
2.1.2 `IAgent` Interface . . . 3
2.1.3 `BaseAgent` Class . . . 4
2.2 *baseserver* Package . . . 5
2.2.1 Composition Structure . . . 5
2.2.2 Agent Injection . . . 6
2.2.3 `IServer` Interface . . . 7
2.2.4 `BaseServer` class . . . 8
2.3 *messaging* Package . . . 9
2.3.1 Composition Structure . . . 9
2.3.2 `IAgentMessenger` Interface . . . 10
2.3.3 `IMessage` Interface . . . 11
2.3.4 `BaseMessage` class . . . 11

**3 Instantiation** **12**
3.1 Agent Design . . . 13
3.1.1 `CommunicatorAgent` . . . 13
3.1.2 `HelloAgent` . . . 14
3.1.3 `WorldAgent` . . . 16
3.2 Message Design . . . 17
3.3 Server . . . 17
3.3.1 `HelloWorldServer` . . . 18
3.3.2 Agent Injection . . . 18
3.4 Simulator Running and Output . . . 19


# 1 Introduction

This user manual serves as the full documentation for the `basePlatformSOMAS` module / library. The purpose of this module is to 'democratise' the ability to produce a self-organising, multi-agent system (SOMAS), by supplying a 'vanilla' platform that can be parametrised and extended. This removes the need for a prerequisite knowledge of system design, leaving the user to implement a set of agent functions, without having to produce the relevant server to handle it.

This manual is split into two sections. The *implementation* gives the technical detail behind the module and its constituent components, and the *instantiation* describes how the module can be extended for use in a larger, more complex simulator. We finalise the *instantiation* section with a detailed example of how a 'hello world' simulator can be produced, where users can specify two agents that can say "hello" and respond with "world".

# 2 Implementation

The `basePlatformSOMAS` module comprises three (sub-)components:

1. The *baseagent* package supplies a parametrisable agent interface that abstracts the core functionalities of an agent, as well as a `BaseAgent` class that implements the interface, and can be extended / composed to gain access to a 'minimally functioning' agent.

2. Similarly to the *baseagent* package, the *baseserver* package supplies a parametrisable server interface that abstracts the core functionalities of a server, as well as a `BaseServer` class that implements the interface and can be extended / composed to gain access to a 'minimally functioning' server.

3. The *messaging* package supplies an abstract `BaseMessage` component that can be extended to allow for the passage of various data structures between agents for communication. This component implements a generic `IMessage` interface, to create a common grammar for agents.

In this section, we describe each of the aforementioned sub-components in detail. For reference, in this manual we name all interfaces with the pattern `I<Name>` and all classes with a capitalised first letter. In composition diagrams, any composed interfaces are put in bold to distinguish them from composed data classes.

## 2.1 *baseagent* Package

This section discusses the two components of the *baseagent* package: the `IAgent` interface, and `BaseAgent` class, as well as the intended use of these components through a composition diagram. This package is accessible from the main module by importing:
`github.com/MattSScott/basePlatformSOMAS/BaseAgent`

### 2.1.1 Composition Structure

This package is written with a focus on *extensibility*. We supply a base interface, `IAgent`, along with a base implementation of this interface, `BaseAgent`.

The intended use of this package is to assist in creating a parametrised agent that is suitable for a given problem specification. For example, if the problem requires that all agents must sleep, the user should add a method to the agents' interface such as `Sleep()`.

To achieve this, we recommend implementing an intermediary 'tier' of agent, in the form of an `IExtendedAgent` (the interface of an extended agent), which encapsulates all of the required functionality for a particular environment - the *action space*. This `IExtendedAgent` should compose (extend) the original `IAgent` interface to 1.) gain the core functionality from this interface and 2.) allow

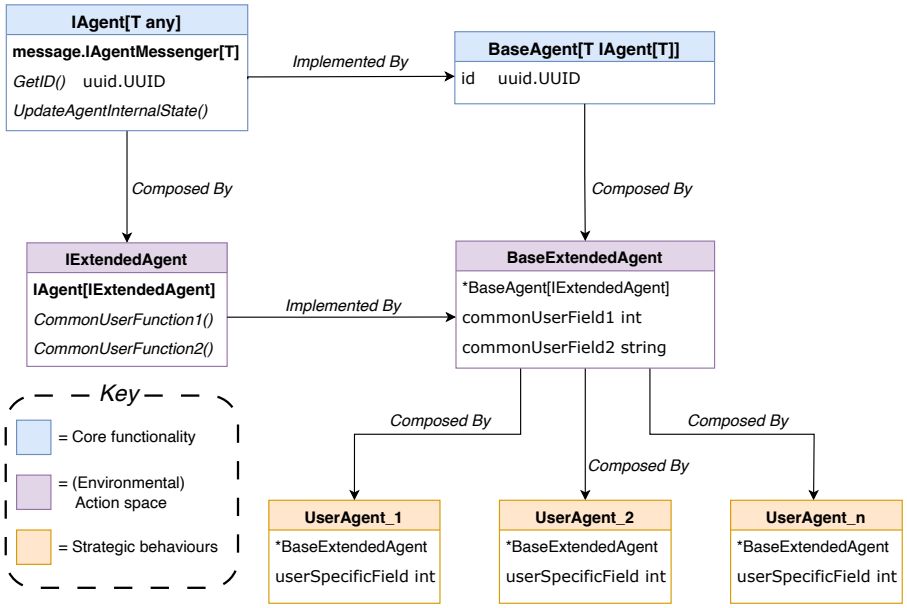


Figure 1: Composition structure diagram for producing a parametrised agent

the `IExtendedAgent` to be used in the *baseserver* package. This `IExtendedAgent` can subsequently be implemented by a `BaseExtendedAgent` class, say, to provide a 'baseline' implementation of the environment-specific functions.

Finally, we envision a final 'tier' of agent, that has the 'agent specific' implementation of the environment-specific functions. This agent should have its own implementation of the `IExtendedAgent` interface, as well as any additional functions that are relevant to informing the agent's *strategic* behaviour. We illustrate this composition structure in Figure 1.

### 2.1.2 `IAgent` Interface

The 'vanilla' agent interface, `IAgent`, comprises three functions, all of which must be implemented by a user's custom agent. We specify this interface below:

```
1  package baseagent
2
3  type IAgent[T any] interface {
4      // composes messaging passing capabilities
5      message.IAgentMessenger[T]
6      // returns the unique ID of an agent
7      GetID() uuid.UUID
8      // allows agent to update their internal state
9      UpdateAgentInternalState()
10 }
```

Listing 1: `IAgent` interface

The first element of the `IAgent` interface is `IAgentMessenger`, another interface, which this one composes. This composed interface holds the functionality for message passing, which is discussed in detail later in Section 2.3.

The second element of this interface is the `GetID()` method. Upon creating an agent, it is given a unique *ID* of type *UUID* (a secure alphanumeric string with low change of collision), for identification. The `BaseAgent` implements this method, and instantiates an agent with a randomised *UUID*.

The final element of this interface is the `UpdateAgentInternalState()` method. This method is called by the server once per turn, and is intended to be implemented / overridden by the user's agent to provide strategic behaviours. If the user wishes to keep track of, say, the number of turns it has been alive, then this can be implemented inside of this function, to provide iterated updates to any (private) attributes of the user's agent class.

Importantly, this interface is declared as a *generic* interface: i.e., `IAgent[T any]`. This allows for the interface to be parametrised with the `IExtendedAgent` interface, such that during message passing the appropriate type of agent can be retrieved.

### 2.1.3 `BaseAgent` Class

The `BaseAgent` class supplies a implementation of the `IAgent` interface, such that it can be composed for use in a `BaseExtendedAgent`, say. This class only stores the *ID* of the agent, which can be retrieved using its implementation of the `GetID()` function. We formalise this class below:

```go
package baseagent

type BaseAgent[T IAgent[T]] struct {
    // stores ID of agent
    id uuid.UUID
}

// generates a new BaseAgent instance with a randomised ID
func NewAgent[T IAgent[T]]() *BaseAgent[T] {
    return &BaseAgent[T]{
        id: uuid.New(),
    }
}
```

Listing 2: `BaseAgent` class

When composing / extending the `BaseAgent` class, it is recommended to pass a pointer to this class to the composing class (see Figure 1). To achieve this, the *baseagent* package supplies the `NewAgent()` function, which returns (a pointer to) an instantiated `BaseAgent` class, with a randomly generated *ID*.

Similarly to the `IAgent` interface, this class is created with *generics* to allow for parametrisation. This class is able to be parametrised with any interface that composes / extends the `IAgent` interface, such that any composing classes can be used in the server with a common type, and also such that messaging can be performed with `ExtendedAgents`. As such, during instantiation the most recent 'parent' interface should be passed as a parameter (the `IExtendedAgent` in Figure 1, say).

## 2.2 *baseserver* Package

This section discusses the two components of the *baseserver* package: the `IServer` interface, and `BaseServer` class, as well as the intended use of these components through a composition diagram, and how agents can be added into the system, through the use of agent injection methods. This package is accessible from the main module by importing:
`github.com/MattSScott/basePlatformSOMAS/BaseServer`

### 2.2.1 Composition Structure

Similarly to the implementation of the the *baseagent* package in Section 2.1, we design a set of 'core functions' that can be expanded upon for producing a parametrised server that is relevant to the environment-specific application of the module.

We supply an `IServer` interface, which encompasses the core functionality of a multi-agent simulator / system: updates to *environmental state* and updates to *agent (game) state*. We produce a `BaseServer` class that implements this interface, thereby providing a (basic) implementation of these functions which can be overriden through composition.

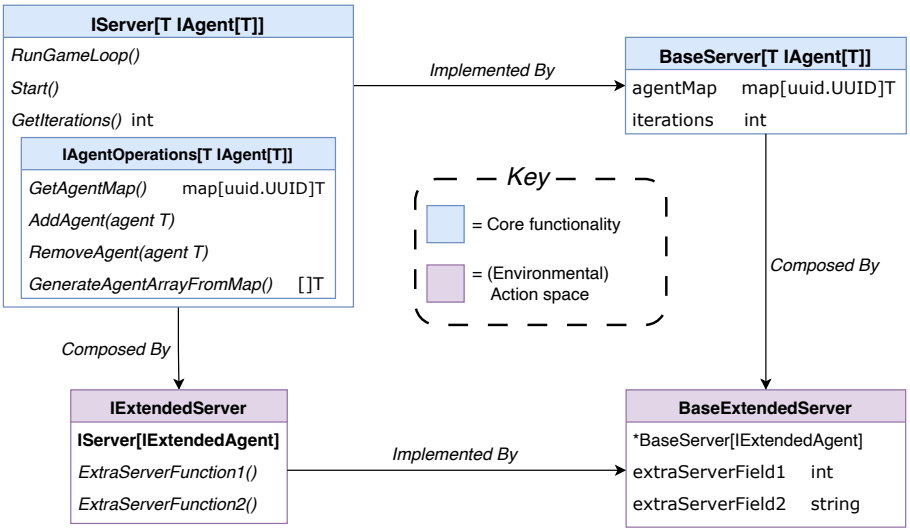


Figure 2: Composition structure diagram for producing a parametrised server

Unlike with the *baseagent* package, we don't envision a third 'tier' of composition. Whereas there was previously a need for agents to be given 'strategic' behaviours, requiring composition of an intermediate class, a single server is responsible for operation on all agents. As such, there is only a second 'tier' of composition, to produce an environmentally-specific server that adds additional functionality to the system, such as `RunSleepPhase()`, continuing with the example of an `agent.Sleep()` function, from before.

### 2.2.2 Agent Injection

In order to use a variety of agents in the simulator, the user must supply a 'generator function' that the server can call to instantiate the relevant type of agent. To assist with this, we provide two 'meta' types - the `AgentGenerator` and the `AgentGeneratorCountPair`.

```
1 package baseserver
2
3 type AgentGenerator[T baseagent.IAgent[T]] func() T
4
5 type AgentGeneratorCountPair[T baseagent.IAgent[T]] struct {
6     generator AgentGenerator[T]
7     count     int
8 }
```

Listing 3: Agent generator methods

The `AgentGenerator` is an alias for a function that returns any generic type that implements the `IAgent` interface. This generator function is then coupled with a *count* to produce the `AgentGeneratorCountPair`: a data class that comprises the type of agent the user wishes to generate, and the number of agents of this type that the user would like generated. An array of this data structure is then parsed by the server for the instantiation of $m$ counts of $k$ different agents. The total number of agents in the simulator, $n$, is then defined by $n = m * k$.

### 2.2.3 `IServer` Interface

The `IServer` interface defines all of the core functions that a server should carry out. We consider two kinds of functions: 1.) functions which affect the *environmental state* and 2.) functions which affect the *agent (game) state*.

We supply three functions that update/control the environmental state. Firstly, `RunGameLoop()` encompasses all of the functionality that runs in a single turn / iteration of a simulated system. This function can be overridden to create environmentally specific behaviours / phases in the system. Secondly, `GetIterations()` returns the number of iterations in the simulator. This value is set once upon the creation of the server and retrievable thereafter. Finally, `Start()` begins the simulator, running it for the predefined number of turns before exiting the program.

We also supply a second interface, `IAgentOperations`, that (for readability) is composed by the `IServer` interface to supply functionality for updating the *agent state*.

In this server implementation, agents are stored in a *hashmap*, mapping an agent's (UU)ID to a (pointer to) the agent. This allows for constant-time insertion, deletion and retrieval of the agent based on its ID. As such, the method `GetAgentMap()` returns this map. Supplementing this are the methods `AddAgent(agent)` and `RemoveAgent(agent)`, which add and remove an agent object from the map, respectively.

```
package baseserver

type IAgentOperations[T baseagent.IAgent[T]] interface {
    // gives access to the agents in the simulator
    GetAgentMap() map[uuid.UUID]T
    // adds an agent to the server
    AddAgent(agentToAdd T)
    // removes an agent from the server
    RemoveAgent(agentToRemove T)
    // translate the agent map into an array of agents
    GenerateAgentArrayFromMap() []T
}

type IServer[T baseagent.IAgent[T]] interface {
    // the set of functions defining how a 'game loop' should run
    RunGameLoop()
    // begins simulator
    Start()
    // operations for adding/removing agents from the simulator
    IAgentOperations[T]
    // access number of iterations in simulator
    GetIterations() int
}
```

Listing 4: `IServer` interface

Finally, we supply the function `GenerateAgentArrayFromMap()` which returns an array of (parametrised) agents. This allows for the map to be converted to an array of agents for ordered (non-random) iteration, say. It can be overridden to redefine how the agent map should be 'translated' to an array, and is used for functions in the *messaging* package, discussed in Section 2.3.

As with the agent interface, this server takes the generic type `T` as an argument. This allows for the server to be parametrised with the agents' most recent 'parent' interface (`IExtendedAgent`, in Section 2.1), to populate the agent map with the relevant type (and therefore provide access to any additional functions gained from composing the `IAgent` interface).

### 2.2.4 `BaseServer` class

The `BaseServer` class supplies a implementation of the `IServer` interface, such that it can be composed for use in a `BaseExtendedServer`, say. This class stores the `AgentMap`, which can be retrieved using its implementation of the `GetAgentMap()` function, and the number of turns the simulator should run for, `iterations`, which is accessible through the `GetIterations()` function.

```go
package baseserver

type BaseServer[T baseagent.IAgent[T]] struct {
    agentMap map[uuid.UUID]T
    numTurns int
}

// generate a server instance based on a mapping function and
// number of iterations
func CreateServer[T baseagent.IAgent[T]] \\
    (generatorArray []AgentGeneratorCountPair[T], iters int) \\
        *BaseServer[T] {
    serv := &BaseServer[T]{
        agentMap: make(map[uuid.UUID]T),
        iterations: iters,
    }
    serv.initialiseAgents(mapper)
    return serv
}
```

Listing 5: `BaseServer` class

Creating a server instance requires calling the `CreateServer()` function which accepts two arguments: a `generatorArray` and `iters`. The former is an array of `AgentGeneratorCountPair`, described in Section 2.2.2, which contains the function to instantiate an agent, and the number of times it should be called. This allows for the agents to be added to the system, with `initialiseAgents()`. The second argument is an integer which represents the number of turns that the server should run for.

## 2.3 *messaging* Package

The final package we supply is the *messaging* package, which allows for the specification of complex message types and facilitates communication between agents. This package is accessible from the main module by importing:
`github.com/MattSScott/basePlatformSOMAS/messaging`

### 2.3.1 Composition Structure

As with the *baseagent* and *baseserver* packages, the primary focus of this package is extensibility. We supply an architecture that allows for the specification of multiple, complex message types that are compatible with any agent, in any server.

We continue with the general composition structure discussed in the previous two packages, where we supply an `IMessage` interface to encompass the core functionality of a message, and a `BaseMessage` class that gives a base implementation of this interface which can be composed in more complex, user defined messages.

In the implementation of the `IAgent` interface, agents are required to return an `IMessage` from the `GetAllMessages()` function. Furthermore, the sender of an `IMessage` is defined as (something that composes) the `IAgent` interface. This

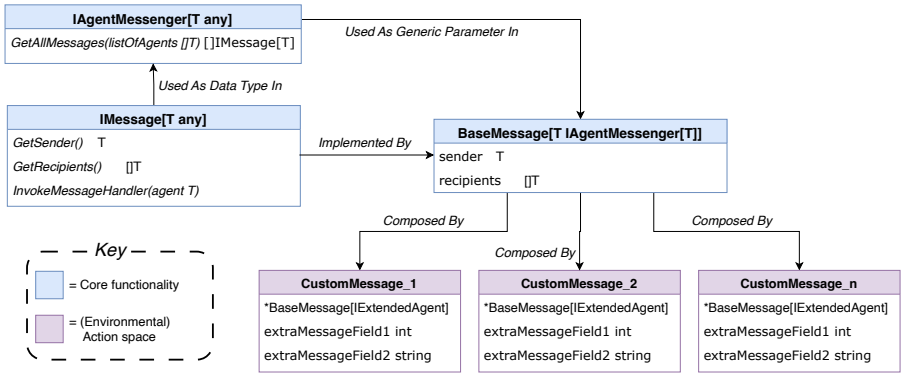


Figure 3: Composition structure diagram for producing various message types

cyclical dependency is solved by introducing a 'proxy' interface: the `IAgentMessenger`, which encompasses the message-specific functionality that agents must implement. This 'proxy' interface is then composed by the `IAgent` interface in Figure 1, to ensure that all user-specified agents which implement (a composition of) the `IAgent` interface are compatible with sending messages.

### 2.3.2 `IAgentMessenger` Interface

The `IAgentMessenger` interface contains the `GetAllMessages()` method for accessing the set of messages an agent wishes to send across the multi-agent system. This method accepts a list of agents as a parameter, which can be accessed in the implementation to inform the recipients for each message. We formalise this interface below:

```
1 package messaging
2
3 // base interface structure used for message passing that
4 // can be composed for more complex message structures
5 type IAgentMessenger[T any] interface {
6     // produces a list of messages (of any common interface)
7     // that an agent wishes to pass
8     GetAllMessages(listOfAgents []T) []IMessage[T]
9 }
```

Listing 6: `IAgentMessenger` interface

In this implementation, we represent the list of messages an agent wishes to send as an array of `IMessage` interfaces. In order to have each user-defined message type handled strategically, we employ the visitor design pattern, such that the `IMessage` interface invokes the appropriate message handler function on the agent. This means that, despite having an array of interface types, each object can be handled as a specific type of message. We discuss this pattern

further in the following section.

### 2.3.3 `IMessage` Interface

The `IMessage` interface comprises the core functionality for producing a customised message class. This interface has three methods, which all extended messages must implement: a method for accessing the *sender* of a message, a method for accessing the *recipients* of a message and a method for telling the agent how to handle the specific type of message:

```
1  package messaging
2
3  // base interface structure used for messages
4  // can be composed for more complex message structures
5  type IMessage[T any] interface {
6      // returns the sender of a message
7      GetSender() T
8      // returns the list of agents that the
9      // message should be passed to
10     GetRecipients() []T
11     // calls the appropriate messsage handler method
12     // on the receiving agent
13     InvokeMessageHandler(T)
14 }
```

Listing 7: `IMessage` interface

The `GetSender()` method returns the generic type used to parametrise the `IMessage` interface. As such, by parametrising it with `IExtendedAgent`, say, any agents that access sender of this message object will have access to the (exported) functions declared in the `IExtendedAgent` interface. This can help with informing how the message should be handled, since more information about the sender may be required for strategic behaviours.

The `GetRecipients()` method returns a list of the generic type used to parametrise the `IMessage` interface. This allows the server to access the intended receivers of a message for redistribution, where it will call the `InvokeMessageHandler()` method.

The `InvokeMessageHandler()` method is used to implement the visitor design pattern, such that specific agent methods can be called on a particular message object. We envision that the `IExtendedAgent` interface adds all of the appropriate message handler methods, as in Figure 4.

Following this, the custom message object can then override the `InvokeMessageHandler()` function to call the appropriate handler from the agent class, passing the message itself as a parameter.

### 2.3.4 `BaseMessage` class

The final component in the *messaging* package is the `BaseMessage` class. This class provides a base implementation of the `IMessage` interface, which can be used in composition for more complex message types.

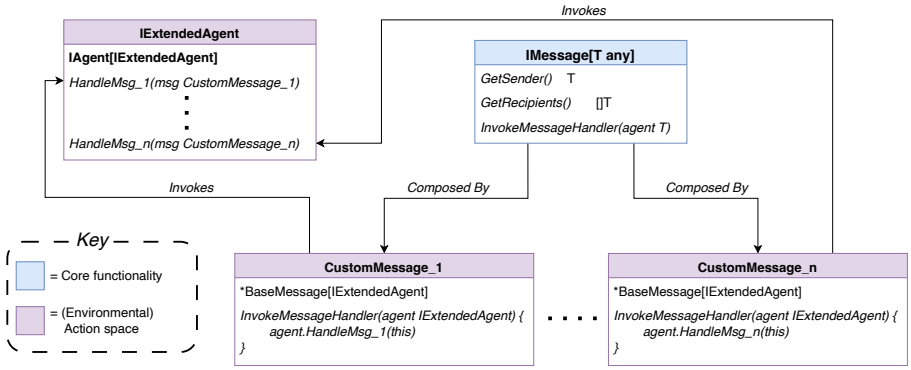


Figure 4: Visitor design pattern for handling various message types

This class contains two fields: `sender` and `recipients`, which store the sender of the message (a single agent that implements `IAgent`) and the recipients of the message (an array of agents that implement `IAgent`) respectively.

```
1  package messaging
2
3  // new message types can extend this
4  type BaseMessage[T IAgentMessenger[T]] struct {
5      sender     T
6      recipients []T
7  }
8
9  // create read-only message instance
10 func CreateMessage[T IAgentMessenger[T]] \\
11     (sender T, recipients []T) BaseMessage[T] {
12     return BaseMessage[T]{
13         sender:     sender,
14         recipients: recipients,
15     }
16 }
```

Listing 8: `BaseMessage` class

For composition, we supply a method for instantiating a `BaseMessage` instance. By calling `CreateMessage`, and passing the sender and recipients, a `BaseMessage` object is instantiated, giving composed messages access to the pre-implemented `GetSender()` and `GetRecipients()` methods.

# 3 Instantiation

In this section, we demonstrate how this package can be used to produce a 'Hello World' simulator. This simulator comprises two simple agents, a `HelloAgent`

and `WorldAgent`, that send the phrase "hello" and respond with "world", respectively.

To achieve this, we decompose the simulator into three sections. We start by formalising a common parent interface (similar to the `IExtendedAgent` from Section 2.1) and the additional methods that these agents will require for communication.

After this, we specify a messaging system, where we formalise a new message type, the `GreetingMessage`, that implements and extends the `IMessage` interface from Section 2.3. We also discuss the additional handler and generator methods that the agents will need in order to appropriately handle and generate a `GreetingMessage`, respectively.

Finally, we formalise a server that implements and extends the `IServer` interface from Section 2.2, illustrating how we can parametrise it with the common parent interface to successfully run the agents' messaging session.

## 3.1 Agent Design

In this section, we specify the agent composition structure for producing three total agents.

Firsly, we specify a 'parent' agent that holds the common methods needed for the "hello world" functionality and is used to parametrise the messaging and server interfaces. We call this the `CommunicatorAgent`.

Secondly, we specify two further agents that compose the 'parent'. These aer the `HelloAgent` and `WorldAgent` that send and receive "hello" and "world", respectively.

### 3.1.1 `CommunicatorAgent`

The `CommunicatorAgent` must encapsulate all of the functionality needed for composing agents to send and receive messages with a string content. For this reason, we compose the `IAgent` interface, parametrising it with itself, and adding two extra methods:

`HandleGreetingMessage` is used to handle a 'greeting' message. This message type is discussed in Section 3.2, and serves as the key data type sent between agents to permit communication.

We also supply each agent with a 'phrase' that they should sent, which can be retrieved with `GetPhrase()`. In order to have bi-directional communication, such that not only `HelloAgents` send a message, but `WorldAgents` too, we also give the latter the phrase of "wello" that the former should respond to with "horld" [1]. This gives both agents the capacity to send and handle a message.

For this functionality, we specify two data structures: an interface, comprising the two functions discusses above, and a class, which implements the interface and can be composed in the `HelloAgent` and `WorldAgent` classes. We first formalise this interface below:

[1] This is known as a *spoonerism*.

```
package baseCommunicatorAgent

type ExtraMessagingFunctions interface {
    HandleGreetingMessage(GreetingMessage)
}

type ICommunicatorAgent interface {
    IAgent[ICommunicatorAgent]
    ExtraMessagingFunctions
    GetPhrase() string
}
```

Listing 9: `CommunicatorAgent` interfaces

where we add a separate interface (that the agent composes) for clarity - the `ExtraMessagingFunctions` interface. This is used as a means of splitting up the functionality of the agent, to show that multiple different message types can be passed and handled by a single agent.

The `CommunicatorAgent` struct implements this interface. It composes the `BaseAgent` class and has a field *phrase* which stores the message that the agent passes. This object can be instantiated by the `GetBaseCommunicatorAgent()` method which accepts the phrase that the agent should be initialised with. This method is formalised below:

```
package baseCommunicatorAgent

type BaseCommunicatorAgent struct {
    *BaseAgent[ICommunicatorAgent]
    phrase string
}

func GetBaseCommunicatorAgent(phrase string) \\
    *BaseCommunicatorAgent {
    return &BaseCommunicatorAgent{
        BaseAgent: NewBaseAgent[ICommunicatorAgent](),
        phrase:    phrase,
    }
}
```

Listing 10: `BaseCommunicatorAgent` struct

where, like with the interface, the `BaseAgent` is parametrised using the 'parent' interface (itself) to enable the correct message type to be generated in the server. This class also implements a the interface methods in a 'standard' way, returning the phrase in the getter and simply 'ignoring' the message in the handler.

### 3.1.2 `HelloAgent`

The `HelloAgent` is responsible for sending the phrase "hello" and receiving the phrase "wello", to which it will reply "horld". To achieve this, this class must implement the methods `GetAllMessages()` (from `IAgent`), to generate the messages for sending and `HandleGreetingMessage()` (from `ICommunicatorAgent`)

to handle the message, once it's received. We formalise the entire class below:

```
package helloagent

type HelloAgent struct {
    *BaseCommunicatorAgent
}

func (ha *HelloAgent) CreateGreetingMessage \\
    (recips []ICommunicatorAgent) GreetingMessage {
    // for simplicity print the message being sent to console
    msgText := fmt.Sprintf \\
        ("%s said: '%s'", ha.GetID(), ha.GetPhrase())
    fmt.Println(msgText)
    // call method from baseCommunicatorAgent package
    return baseCommunicatorAgent.CreateGreetingMessage \\
            (ha, recips, ha.GetPhrase())
}

func (ha *HelloAgent) GetAllMessages \\
    (availableAgents []ICommunicatorAgent) \\
        []IMessage[ICommunicatorAgent] {

    // send a single message to all agents
    msg := ha.CreateGreetingMessage(availableAgents)

    // return this single message in an array
    return []IMessage[ICommunicatorAgent]{msg}

}

func (ha *HelloAgent) HandleGreetingMessage \\
    (msg GreetingMessage) {

    // check if incoming message has content "wello"
    if msg.GetGreeting() == "wello" {
        // for simplicity, print response to console
        respText := fmt.Sprintf \\
            ("%s responded: 'horld'", ha.GetID())
        fmt.Println(respText)
    }
}

// return nearest parent interface
func GetHelloAgent() ICommunicatorAgent {
    return &HelloAgent{
        GetBaseCommunicatorAgent("hello"),
    }
}
```

Listing 11: `HelloAgent` package

note that the console printing is used as a 'visual' output for what messages are being sent. This functionality could have instead been added to the server by overriding the `RunMessagingSession()` method, however we leave it here for readability.

### 3.1.3 `WorldAgent`

The `WorldAgent` is almost identical in implementation to the `HelloAgent`. We formalise the entire package below:

```
package worldagent

type WorldAgent struct {
    *BaseCommunicatorAgent
}

func (wa *WorldAgent) CreateGreetingMessage \\
    (recips []ICommunicatorAgent) GreetingMessage {
    // for simplicity print the message being sent to console
    msgText := fmt.Sprintf \\
        ("%s said: '%s'", wa.GetID(), wa.GetPhrase())
    fmt.Println(msgText)
    // call method from baseCommunicatorAgent package
    return baseCommunicatorAgent.CreateGreetingMessage \\
            (wa, recips, wa.GetPhrase())
}

func (wa *WorldAgent) GetAllMessages \\
    (availableAgents []ICommunicatorAgent) \\
        []IMessage[ICommunicatorAgent] {

    // send a single message to all agents
    msg := wa.CreateGreetingMessage(availableAgents)

    // return this single message in an array
    return []IMessage[ICommunicatorAgent]{msg}

}

func (wa *WorldAgent) HandleGreetingMessage \\
    (msg GreetingMessage) {

    // check if incoming message has content "hello"
    if msg.GetGreeting() == "hello" {
        // for simplicity, print response to console
        respText := fmt.Sprintf \\
            ("%s responded: 'world'", wa.GetID())
        fmt.Println(respText)
    }
}

// return nearest parent interface
func GetWorldAgent() ICommunicatorAgent {
    return &WorldAgent{
        GetBaseCommunicatorAgent("wello"),
    }
}
```

Listing 12: `WorldAgent` package

## 3.2 Message Design

As aforementioned, the data structure that agents pass between themselves must contain a string content, or 'greeting'. To achieve this, we compose the `BaseMessage` from Section 2.3 and add an additional string field and getter method. We formalise the entire data structure below:

```
package baseCommunicatorAgent

// compose parametrised Base Message class
type GreetingMessage struct {
    BaseMessage[ICommunicatorAgent]
    greeting string
}

// simple getter returns greeting from message
func (em *GreetingMessage) GetGreeting() string {
    return em.greeting
}

// calls the appropriate handler from the parent agent
func (em GreetingMessage) InvokeMessageHandler \\
    (agent ICommunicatorAgent) {
    agent.HandleGreetingMessage(em)
}

// instantiate Greeting Message by calling parametrised
// Base Message constructor
func CreateGreetingMessage(sender ICommunicatorAgent, \\
    recipients []ICommunicatorAgent, content string) \\
    GreetingMessage {
    return GreetingMessage{
        BaseMessage: CreateMessage[ICommunicatorAgent] \\
                        (sender, recipients),
        greeting:    content,
    }
}
```

Listing 13: `GreetingMessaging` class

Importantly, this data structure is stored in the *same* package as the `CommunicatorAgent`. This is because the agent must have access to the data structure in its handler, and the message must have access to the agent in the invoker function. This is a cyclical dependency commonly found as a by-product of the visitor design pattern.

## 3.3 Server

The final component for producing the `HelloWorldSimulator` is the server. The `BaseServer` provided in the package already supplies methods for running the messaging session and starting the game loop. As such, no additional methods are needed. All that is required is producing a parametrised server that is compatible with the newly specified `ICommunicatorAgent` interface.

### 3.3.1 `HelloWorldServer`

To achieve these requirements, we specify both an `IHelloWorldServer` interface and class, which can be instantiated for use in a main file. We formalise this package below:

```
package helloworldserver

// the IServer interface only needs parametrising with the
// parent class since no new methods are needed
type IHelloWorldServer interface {
    IServer[ICommunicatorAgent]
}

// the BaseServer class only needs parametrising with the
// parent class since no new fields are needed
type HelloWorldServer struct {
    *BaseServer[ICommunicatorAgent]
}

// instantiate new Communicator Server by calling constructor
// from baseserver package
func CreateHelloWorldServer \\
  (generatorArray []AgentGeneratorCountPair[ICommunicatorAgent],
    iterations int) *HelloWorldServer {
    return &HelloWorldServer{
         baseserver.CreateServer[ICommunicatorAgent] \\
            (mapper, iterations),
    }
}
```

Listing 14: *HelloWorldServer* package

### 3.3.2 Agent Injection

We wish to inject three `HelloAgents` and two `WorldAgents`, for this example. To achieve this, we supply the *generatorArray*, which consists of two `AgentGeneratorCountPairs` from Section 2.2.2. Formally, this object looks as follows:

```
// there are two generator pairs in the array
generatorArray := \\
        make([]AgentGeneratorCountPair[ICommunicatorAgent], 2)

// we pass the generator function from the HelloAgent 3 times
generatorArray[0] = \\
    MakeAgentGeneratorCountPair[ICommunicatorAgent] \\
        (GetHelloAgent, 3)

// we pass the generator function from the WorldAgent 2 times
generatorArray[1] =
    MakeAgentGeneratorCountPair[ICommunicatorAgent] \\
        (GetWorldAgent, 2)
```

Listing 15: *GeneratorArray* for agent instantiation

which can then be passed to a new instance of the `HelloWorldServer` for use in a main file.

## 3.4 Simulator Running and Output

Having specified all of these components, we can produce a working simulator through creating of a main file. Using the generator array from Listing 15, we can produce the following function:

```
1
2 func main() {
3
4     iters := 2
5     server := CreateHelloWorldServer(generatorArray, iters)
6
7     server.start()
8 }
```

Listing 16: Main file for simulator

By running this main file with `go run .` the output in Figure 5 is produced (with variation in the order of message passing and agent ID). This demonstrates that the simulator is running as expected, with two types of agents interacting and sharing messages between one another.

```
Main game loop running...

Agent 48ae255e-37c3-4204-b8f4-ddab7705936a updating state
Agent 424d602d-8f75-40bd-999a-dee95b87e341 updating state
Agent 4fa0dd93-2cd8-4294-aa46-d37480b305f5 updating state
Agent ab0e5308-86b1-4df0-b595-a9ecfa0b3b88 updating state
Agent 161249cd-8d81-4191-ac3a-f01dfe4444ad updating state

Main game loop finished.

Messaging session started...

424d602d-8f75-40bd-999a-dee95b87e341 said: 'wello'
ab0e5308-86b1-4df0-b595-a9ecfa0b3b88 responded: 'horld'
161249cd-8d81-4191-ac3a-f01dfe4444ad responded: 'horld'
4fa0dd93-2cd8-4294-aa46-d37480b305f5 responded: 'horld'
4fa0dd93-2cd8-4294-aa46-d37480b305f5 said: 'hello'
48ae255e-37c3-4204-b8f4-ddab7705936a responded: 'world'
424d602d-8f75-40bd-999a-dee95b87e341 responded: 'world'
ab0e5308-86b1-4df0-b595-a9ecfa0b3b88 said: 'hello'
48ae255e-37c3-4204-b8f4-ddab7705936a responded: 'world'
424d602d-8f75-40bd-999a-dee95b87e341 responded: 'world'
161249cd-8d81-4191-ac3a-f01dfe4444ad said: 'hello'
48ae255e-37c3-4204-b8f4-ddab7705936a responded: 'world'
424d602d-8f75-40bd-999a-dee95b87e341 responded: 'world'
48ae255e-37c3-4204-b8f4-ddab7705936a said: 'wello'
ab0e5308-86b1-4df0-b595-a9ecfa0b3b88 responded: 'horld'
161249cd-8d81-4191-ac3a-f01dfe4444ad responded: 'horld'
4fa0dd93-2cd8-4294-aa46-d37480b305f5 responded: 'horld'

Messaging session completed
```

Figure 5: Output from `HelloWorldSimulator`