

# Messaging Systems

---

**Name:** Shiva Gupta

**Student ID:** 102262514

**Unit Code:** COS30031

**Task Number:** 119

**Date:** 07/11/2021

**Repo link:** [https://bitbucket.org/ShivaGupta\\_/cos30031-102262514/src/master/19%20-%20Spike%20-%20Messaging\\_%20Announcements%20and%20Blackboards/](https://bitbucket.org/ShivaGupta_/cos30031-102262514/src/master/19%20-%20Spike%20-%20Messaging_%20Announcements%20and%20Blackboards/)

---

## Goals/Deliverables

[SPIKE REPORT] + [CODE]

## Knowledge Gap

Understanding the blackboard and dispatcher messaging systems and implementing such systems to send messages between entities. The types of messages can vary and the entities need to be able to use the messages sent meaningfully.

---

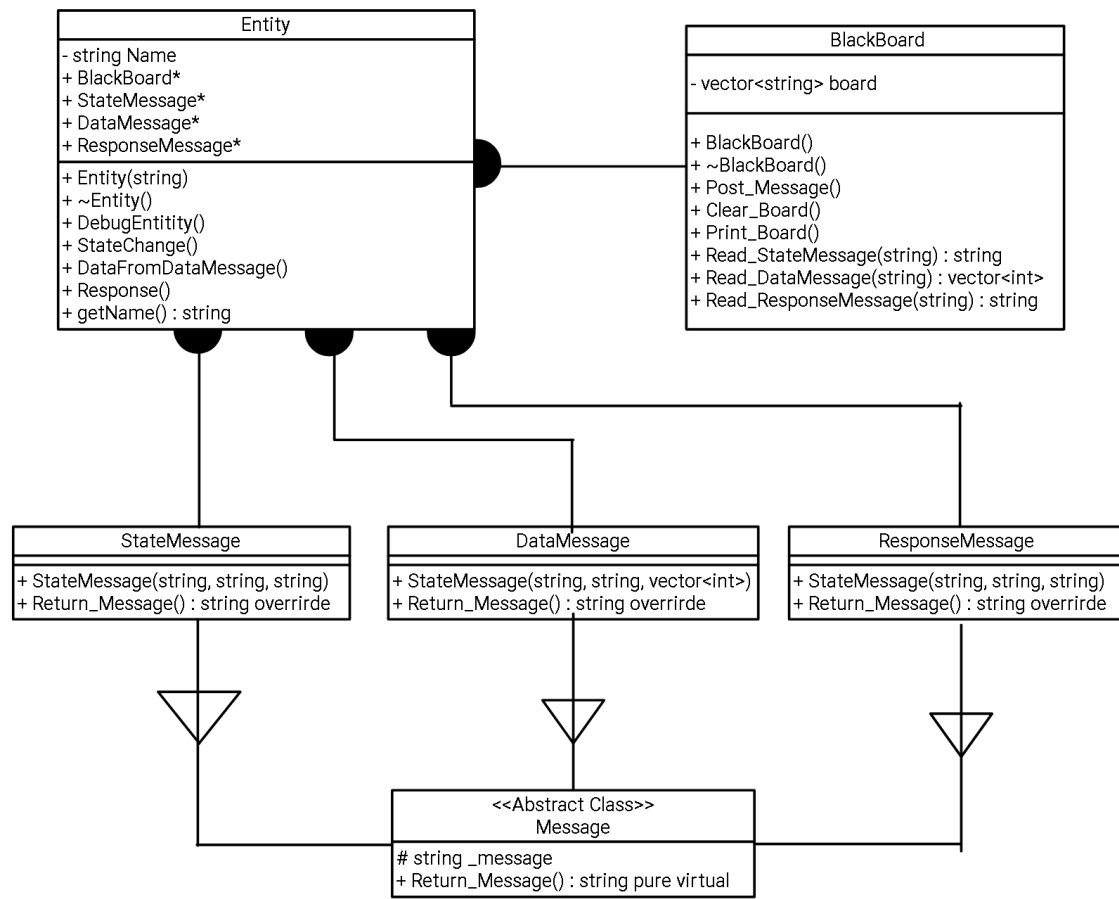
## Tasks Undertaken

### BlackBoard Messaging System

---

For the blackboard approach we first think about the architecture. The main features of this approach are as follows:

1. The entities can not communicate directly, only by subscribing to the blackboard.
2. Messages can be posted on and read from the blackboard.
3. Entities can use those messages meaningfully.



## Implementign the design

1. Now, we can begin implementing the design. We start by the `Entity` class.
2. Entity will know about it's name and contain references to the different messaging classes.
3. Since Entity is composed of the messaging classes and the blackboard it is responsible for destroying them.

```

class Entity
{
private:
    string name;
public:
    //Every Entitiy will know about the blackboard
    BlackBoard* blackboard = nullptr;
    StateMessage* stateMessage = nullptr;
    DataMessage* dataMessage = nullptr;
    ResponseMessage* responseMessage = nullptr;

    //default constructor
    Entity(const string& name)
        : name(name) {}
  
```

```

//when Entity is destroyed then destroy blackboard
~Entity()
{
    if (stateMessage != nullptr)
        delete stateMessage;
    if (dataMessage != nullptr)
        delete stateMessage;
    if (responseMessage != nullptr)
        delete stateMessage;
    delete blackboard;
}

...

```

4. We want the entities to make sense out of the messages that it reads so we will implement methods to decrypt those messages.

```

void stateChange(const string& state) const {
    if (state == "attacking")
        cout << "Entitiy -> " + name + " being attacked " << endl;
    if (state == "defending")
        cout << "Entitiy -> " + name + " trying to defend " << endl;
}

void DataFromDataMessage(const vector<int>& v) {
    for (int i : v)
        cout << "Name: " << name << " -> Data: " << i << endl;
}

void Response(const string& response) const {
    cout << "Name: " << name << " -> Response: " << response << endl;
}

```

5. Next we implement the `blackboard` class. The blackboard can receive messages, clear itself and print itself to the console.

```

BlackBoard() {}
~BlackBoard() {}

void Post_Message(const string& str)
{
    board.push_back(str);
}

void Clear_Board()
{
    //vector clear removes all the elemetns(which are destroyed)
    //sets the vector size to 0.
    board.clear();
}

void Print_Board()
{
    cout << "\t=====Black Board===== " << endl;
    for (auto msg : board)
    {

```

```

        cout << msg << endl;
    }
    cout << "\t===== " << endl;
}

```

6. Entities can subscribe and read the different messages posted on it and post it's messages.

```

string Read_StateMessage(const string& receiverName)
{
    //Go through the board and search for the message for the receiver
entity
    for (int n = 0; n < board.size(); n++)
    {
        vector<string> v = Split(board[n]);
        if (v[1] == receiverName)
            return v[2];
    }
}

vector<int> Read_DataMessage(const string& receiverName)
{
    //Go through the board and search for the message for the receiver
entity
    for (int n = 0; n < board.size(); n++)
    {
        vector<string> v = Split(board[n]);
        if (v[1] == receiverName)
        {
            vector<int> data;
            data.push_back(stoi(v[2]));
            data.push_back(stoi(v[3]));
            data.push_back(stoi(v[4]));
            return data;
        }
    }
}

string Read_ResponseMessage(const string& receiverName)
{
    //Go through the board and search for the message for the receiver
entity
    for (int n = 0; n < board.size(); n++)
    {
        vector<string> v = Split(board[n]);
        if (v[1] == receiverName)
            return v[3];
    }
}

```

---

## Types of messages

Our task requirements state that we need 3 types of messages that the entities can read and then perform actions based on the messages or the data passed between messages.

7. We will make a message abstract class which the specialized class will implement. In the abstract class we will guarantee that any class that inherits from it will be able to return the complete message.

```
class Message
{
protected:
    string _message;
public:
    virtual const string& Return_Message() const = 0;
};
```

8. Now, we will implement the specialized messages classes.
9. The state message class when called will take in strings which the entity can decrypt and perform action based on it.

```
class StateMessage : public Message
{
public:
    StateMessage
    (
        const string& sendingEntity,
        const string& receivingEntity,
        const string& stateMsg
    )
    {
        _message = sendingEntity + ":" + receivingEntity + ":" + stateMsg;
    }

    const string& Return_Message() const override
    {
        return _message;
    }
};
```

8. The data message class when called will append data to the message that the entity can decrypt and perform action on that data.

```
class DataMessage : public Message
{
public:
    DataMessage
    (
        const string& sendingEntity,
        const string& receivingEntity,
        const vector<int>& data
    )
    {
```

```

        _message = sendingEntity + ":" + receivingEntity;
        for (int i : data)
        {
            _message += ":" + to_string(i);
        }
    }

    const string& Return_Message() const override
    {
        return _message;
    }
};

```

9. The response message class will tag the message as response so that the entity can understand that it is a response.

```

class ResponseMessage : public Message
{
public:
    ResponseMessage
    (
        const string& sendingEntity,
        const string& receivingEntity,
        const string& response
    )
    {
        _message = sendingEntity + ":" + receivingEntity + ":respond" + ":"
+ response;
    }

    const string& Return_Message() const override
    {
        return _message;
    }
};

```

---

## Demonstration of the blackboard system

10. Now, we create the blackboard which by default, is empty. Then we can create the different messages that the entities can use to post to the board.

```

//A is attacking B
StateMessage* stateMessageAB = new StateMessage("A", "B", "attacking");
//C is defeding D
StateMessage* stateMessageCD = new StateMessage("C", "D", "defending");
//E is attacking F
StateMessage* stateMessageEF = new StateMessage("E", "F", "attacking");

DataMessage* dataMessageGH = new DataMessage("G", "H", { 1, 23, 45 });
DataMessage* dataMessageIJ = new DataMessage("I", "J", { 6, 78, 90 });

```

```

    DataMessage* dataMessageKL = new DataMessage("K", "L", { 1123123, 21233,
451 });

    StateMessage* stateMessage1 = new StateMessage("X", "Z", "Are you ready
to surrender?");
    ResponseMessage* responseMessage1 = new ResponseMessage("X", "Z", "Ready
to surrender");
    StateMessage* stateMessage2 = new StateMessage("V", "W", "Raise or
Call?");
    ResponseMessage* responseMessage2 = new ResponseMessage("V", "W",
"raising the bid");

    BlackBoard* blackboard = new BlackBoard();

```

11. We also need to create the entities and assign them the messages and the blackboard for them to post on and read from.
12. And then we can tell an entity to post messages on the blackboard and other entities to read from it provided the message is meant for them.

### State Message on blackboard

```

a.blackboard->Post_Message(a.stateMessage->Return_Message());
c.blackboard->Post_Message(c.stateMessage->Return_Message());
e.blackboard->Post_Message(e.stateMessage->Return_Message());
a.blackboard->Print_Board();

b.stateChange(h.blackboard->Read_StateMessage(h.getName()));
d.stateChange(j.blackboard->Read_StateMessage(j.getName()));
f.stateChange(l.blackboard->Read_StateMessage(l.getName()));

```

```

=====Black Board=====
A:B:attacking
C:D:defending
E:F:attacking
=====

```

```

Entity -> B being attacked
Entity -> D trying to defend
Entity -> F being attacked

```

### Data Message on Blackboard

```

h.DataFromDataMessage(h.blackboard->Read_DataMessage(h.getName()));
j.DataFromDataMessage(j.blackboard->Read_DataMessage(j.getName()));
l.DataFromDataMessage(l.blackboard->Read_DataMessage(l.getName()));

```

```

=====Black Board=====
A:B:attacking
C:D:defending
E:F:attacking
G:H:1:23:45
I:J:6:78:90
K:L:1123123:21233:451
=====

```

```
Name: H -> Data: 1
Name: H -> Data: 23
Name: H -> Data: 45
Name: J -> Data: 6
Name: J -> Data: 78
Name: J -> Data: 90
Name: L -> Data: 1123123
Name: L -> Data: 21233
Name: L -> Data: 451
```

### Response Message on Blackboard

```
x.blackboard->Post_Message(x.stateMessage->Return_Message());
x.blackboard->Post_Message(x.responseMessage->Return_Message());
v.blackboard->Post_Message(v.stateMessage->Return_Message());
v.blackboard->Post_Message(v.responseMessage->Return_Message());
v.blackboard->Print_Board();

z.Response(z.blackboard->Read_ResponseMessage(z.getName()));
w.Response(w.blackboard->Read_ResponseMessage(w.getName()));
```

```
=====Black Board=====
A:B:attacking
C:D:defending
E:F:attacking
G:H:1:23:45
I:J:6:78:90
K:L:1123123:21233:451
X:z:Are you ready to surrender?
X:Z:respond:Ready to surrender
V:w:Raise or Call?
v:W:respond:raising the bid
=====
```

```
Name: Z -> Response: Ready to surrender
Name: W -> Response: raising the bid
```

---

## Message Privacy

In our design we only allow the entities to read a message from the blackboard if the message is meant for them otherwise the message is not passed to the entity to decrypt.

---

## Dispatcher Messaging System

---

The main features of the dispatcher system are:

1. Dispatcher is responsible for delivering the message to the entity.



2. Entity does not directly communicate with other entity.
3. Dispatcher knows about all the entities.

So, we will start by implementing the entity.

13. The entity has a name and it contains a reference to dispatcher. We can send message to the dispatcher and understand different types of messages.

```
Entity::Entity(const string& name) : _name(name) {}
Entity::~Entity()
{
    delete dispatcher;
}

void Entity::Send_Message(const string& s, const string& receiverItem, const
string& type)
{
    dispatcher->Send_Message(s, receiverItem, type);
}

void Entity::Receive_StateMessage(const string& message)
{
    vector<string> args = Split(message);
    cout << "Name: " << _name << " State: " << args[2] << endl;
    SetState(args[2]);
}

vector<int> Entity::Receive_DataMessage(const string& Data)
{
    vector<string> args = Split(Data);
    vector<int> datas;
    cout << "Name: " << _name << " Data: " << args[2] << endl;
    datas.push_back(stoi(args[2]));
    cout << "Name: " << _name << " Data: " << args[3] << endl;
    datas.push_back(stoi(args[3]));
    cout << "Name: " << _name << " Data: " << args[4] << endl;
    datas.push_back(stoi(args[4]));
    return datas;
}

string Entity::Receive_ResponseMessage(const string& message)
{
    vector<string> args = Split(message);
    cout << "Name: " << _name << " Response: " << args[3] << endl;
    string str = args[3];
    return str;
}
```

14. Now we will implement the dispatcher class. The dispatcher has a list of all the entities. It can receive messages and then deliver the message to the receiver entity.
15. The appropriate method for the type of message will be called by the dispatcher.

```
Dispatcher::Dispatcher() {}

Dispatcher::~Dispatcher()
{
}
```

```

    for (auto e : entities)
        delete e;
}

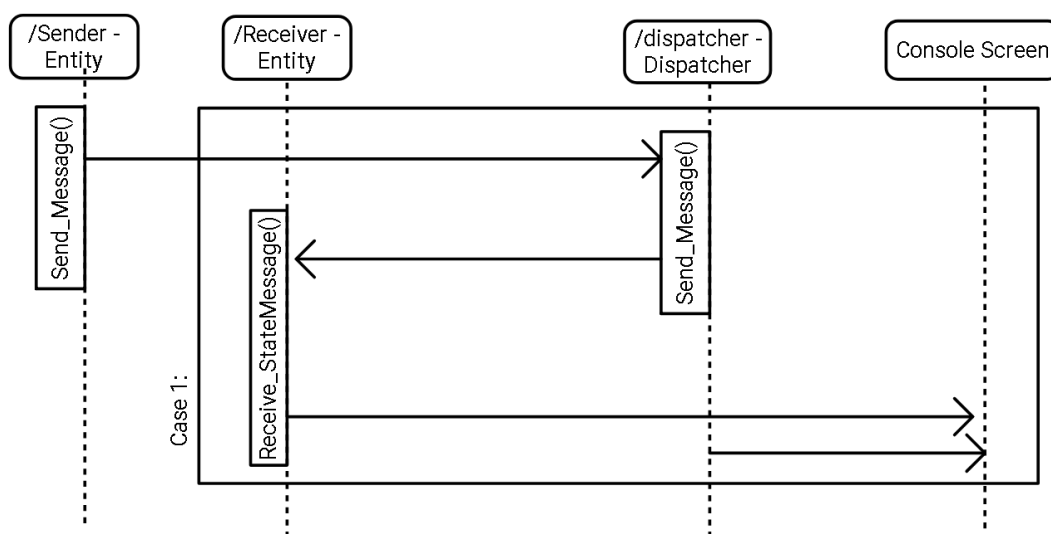
void Dispatcher::AddEntity(Entity* e)
{
    entities.push_back(e);
}

void Dispatcher::Send_Message(const string& message, const string&
receiverName, const string& type)
{
    messages.push_back(message);
    for (list<Entity*>::iterator it = entities.begin(); it !=
entities.end(); ++it)
        //for (auto e : entities)
        {
            if ((*it)->GetName() == receiverName)
            {
                if (type == "state")
                    (*it)->Receive_StateMessage(message);
                if (type == "data")
                    (*it)->Receive_DataMessage(message);
                if (type == "response")
                    (*it)->Receive_ResponseMessage(message);
            }
        }
}
}

```

## Demonstration of the blackboard system

### State Messages - Dispatcher



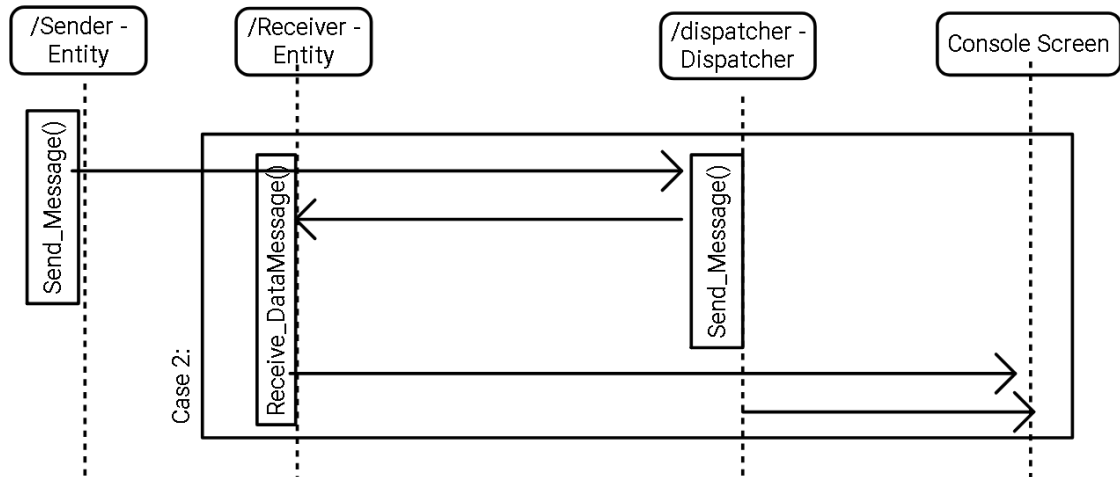
```

=====Dispatcher=====
A:B:running
=====

```

```
Name: B State: running
Entity: B -> has chosen to run away.
```

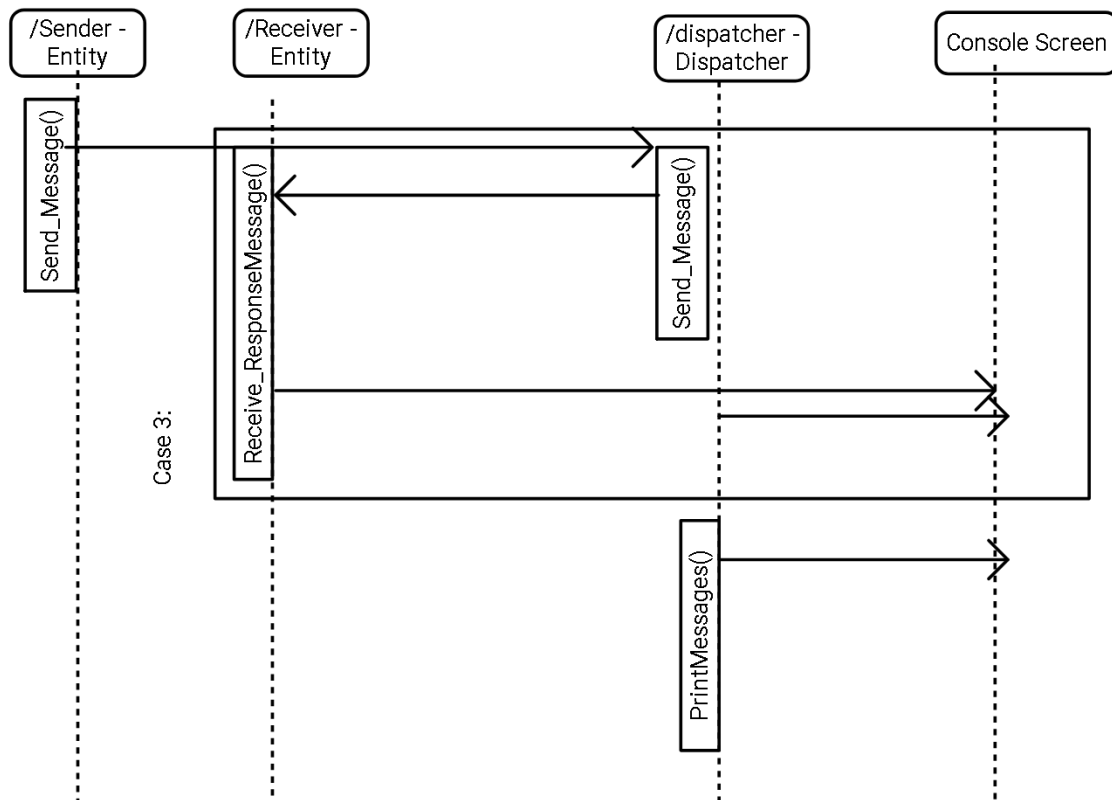
### Data Messages - Dispatcher



```
=====Dispatcher=====
A:B:running
C:D:123:456:678
=====
```

```
1 Name: D Data: 123
2 Name: D Data: 456
3 Name: D Data: 678
4
5 =====Dispatcher=====
6 A:B:running
7 C:D:123:456:678
8
9 =====
```

### Response Message - Dispatcher



```

=====Dispatcher=====
A:B:running
C:D:123:456:678
E:F:Choose to pay or steal?
F:E:Response: player choose to steal
=====

```

```


Name: D State: Choose to pay or steal?
Name: D Response: player choose to steal
=====Dispatcher=====
A:B:running
C:D:123:456:678
E:F:Choose to pay or steal?
F:E:Response: player choose to steal
=====

```

## What we found out

This completes our task for messaging systems. We learnt about two different techniques for handling messages between entities. Both the techniques have different features and advantages. however, both provide a strong foundation for scaling up applications.

### Git Log

 COS30031 - 102262514

<> Source

Commits

Branches

Pull requests

Pipelines

Deployments

Jira issues

Security


Downloads

Repository settings
















Commits

Search commits

Q

 All branches

▼

Author	Commit	Message	Date
 Shiva Gupta	<a href="#">5a7c83a</a>	[MODIFY]: task 19 exception fixed	2 minutes ago
 Shiva Gupta	<a href="#">d7393e9</a>	[ADD]: task 13 report complete task	yesterday
 Shiva Gupta	<a href="#">4ead830</a>	[ADD]: task 12 complete with reports	yesterday
 Shiva Gupta	<a href="#">a337ab6</a>	[ADD]: task 12 UML and code	yesterday
 Shiva Gupta	<a href="#">9475f3a</a>	[MODIFY]: task 13 code complete	yesterday
 Shiva Gupta	<a href="#">ef4c31e</a>	[MODIFY]: task 19 dispatcher code	2 days ago
 Shiva Gupta	<a href="#">da67906</a>	[ADD]: task 19 blackboard complete	2 days ago
 Shiva Gupta	<a href="#">555e9ea</a>	[ADD]: task 19 setup	2 days ago
 Shiva Gupta	<a href="#">543fd8d</a>	[MODIFY]: task 13 command take, put, look, etc. ...	2 days ago
 Shiva Gupta	<a href="#">631a566</a>	[MODIFY]: Take commands for task 13	2 days ago
 Shiva Gupta	<a href="#">6a8c99e</a>	[ADD]: code has items that contain other items	3 days ago
 Shiva Gupta	<a href="#">f805d5b</a>	[ADD]: code command processor complete	4 days ago
 Shiva Gupta	<a href="#">3985ef8</a>	[MODIFY]: move command in task 12	5 days ago
 Shiva Gupta	<a href="#">93abe74</a>	[ADD]: task 22 video and spike report	2021-10-28
 Shiva Gupta	<a href="#">acea58a</a>	[MODIFY]: task 22 code with circle collision detec...	2021-10-28