EECS3311-W2017 — Project Report

Submitted electronically by:

Team members	**Name**	**Prism Login**	**Signature**
Member 1:	Matthew MacEachern	mcmaceac	
Member 2:			
*Submitted under Prism account:		mcmaceac	

Contents

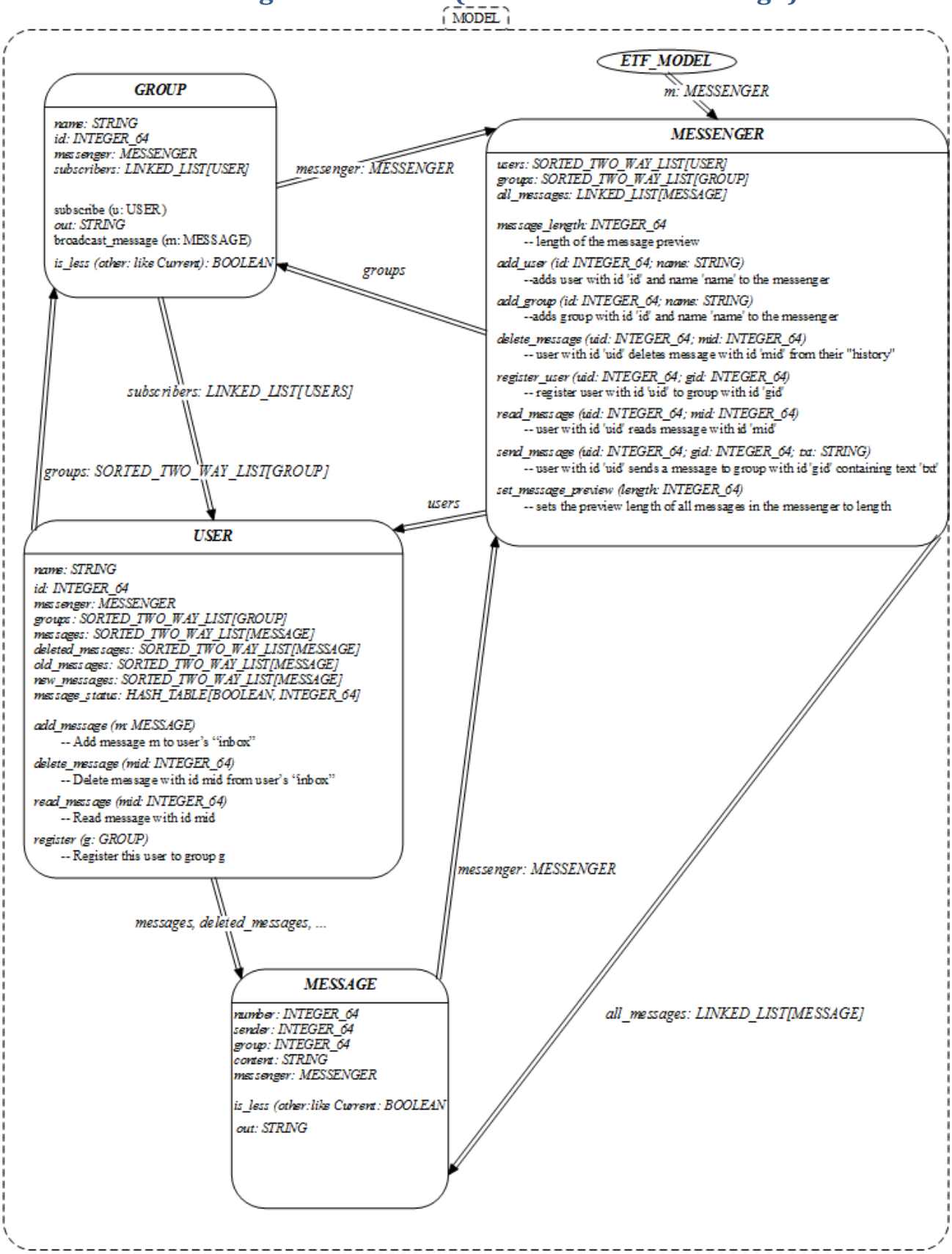
1. Requirements for Project Messenger

As specified in the project specification, this messenger project is a mission critical application. It is very important that privacy between groups is maintained and only users who are registered to certain groups can read the messages that are sent to those groups. As shown by the grammar of the project, the messenger application must allow functionality for adding users, adding groups, registering users to groups, sending a message from a user to a group that they are registered in, list groups in the system, list users in the system, read messages, delete messages, and set message preview length for output.

The grammar along with the oracle program for this application clearly set the guidelines for how this application was supposed to run, and made replicating the desired functionality streamlined and straightforward. User and group IDs should be in the range of 1 to 9,223,372,036,854,775,807. Users can only send messages to groups they are registered in and those messages will be broadcast in a sense to all the members of the group. Groups can have zero to many users and users can be registered in zero to many groups. Users can read messages that have been sent to them and have not already been read, after which they can choose to delete those messages. Deleting a message only deletes the message for the user specified, since other users that may want to see the message locally. Listing groups, users, new messages and old messages are self explanatory commands (see appendix for more detail). Finally set message preview allows the user of the application to change the length of the message shown on the screen after any command is entered. This preview length must be at least 1.

See messenger.definitions.txt for a more detailed explanation of what each command should take as input and what those input types are.

2. BON class diagram overview (architecture of the design)



When choosing what design I was going to use in this project, I wanted to keep it as simple as I could. This would not only make bugs easy to find and easy to fix, but make imitating the oracle straight forward and simple as well.

In my design the MESSENGER class is the class that has most of the core functionality of the application, while trying as much as possible not to be a superman class. The MESSENGER class holds all the users, groups, and messages that are currently in a system. I chose to have this class as a central "storage" of these items since a user could be a part of no groups and a group in the system could have no users that are a part of it. This made storing the users in the group class and vice versa unreasonable and the application would not function the same as the oracle program. All messages are stored in this class because if a message is not available to a user, the application's output needs to reflect this under the message state section of the output. The MESSENGER class is also the most important as far as contracting is concerned, since it is the first class that is encountered after the ETF commands and the defensive programming (for error messages) that is used in those command classes.

The USER class holds all the information about a single user. This includes their name, id, the groups that they are a part of, and the messages that they currently have access to. USERs can have messages added to their inbox, delete messages, read messages, and register for different groups. This way the MESSENGER class does not need to worry about how the USER stores the messages and groups, therefore that information is hidden from the MESSENGER class.

The GROUP class is a simple class that holds the name of the group, the id of the group, and the list of users that are a part of that group named subscribers. This list is needed in order to send messages to all of the members of the group. A group has routines subscribe, which subscribes a USER to the current group, and broadcast, which broadcasts a message that was sent to the current group, out to all the subscribers to the group. In this process, the MESSENGER application sends a message to a group, and this group broadcasts the message to every USER that is registered to this group. This was the most simple and straightforward way that I could imagine implementing this functionality, which allowed me to fix bugs and replicate the oracle program easily.

Finally, the MESSAGE class is another simple class that holds the message number, the sender id of the message, the group id the message is being sent to, and the content of the message. In this way when MESSENGER application is processing messages to be sent to certain groups and subsequently the users that are registered to those groups, the information for the message is encapsulated into this straightforward class.

3. Table of modules — responsibilities and information hiding

1	MESSENGER	**Responsibility**: holds the users, groups, and all messages in the system. Also, provides functionality for these items to interact with each other.	**Alternative**: none
	Concrete	**Secret**: none	

1.1	SORTED_TWO_WAY_LIST[USER]	**Responsibility**: stores all the users that have been added to the system.	**Alternative**: none
	Concrete	**Secret**: sorted list so the output is easily sorted without any more implementation.	

1.2	SORTED_TWO_WAY_LIST[GROUP]	**Responsibility**: stores all the groups that have been added to the system.	**Alternative**: none
	Concrete	**Secret**: sorted list so the output is easily sorted without any more implementation.	

1.3	LINKED_LIST[MESSAGE]	**Responsibility**: stores all the messages that have been sent through the messenger system.	**Alternative**: none
	Concrete	**Secret**: none.	

2	USER	**Responsibility**: stores information pertaining to a user in the system.	**Alternative**: none
	Concrete	**Secret**: none.	

2.1	SORTED_TWO_WAY_LIST[GROUP]	**Responsibility**: stores all the groups that the user has registered for	**Alternative**: none
	Concrete	**Secret**: sorted list so the output is easily sorted without any more implementation.	

2.2	SORTED_TWO_WAY_LIST[MESSAGE]	**Responsibility**: stores all the messages that the user has available to them.	**Alternative**: none
	Concrete	**Secret**: none.	

3	GROUP	**Responsibility**: stores information pertaining to a user in the system.	**Alternative**: none
	Concrete	**Secret**: none	

3.1	LINKED_LIST[USER]	**Responsibility**: stores all the users that are registered to this particular group.	**Alternative**: none
	Concrete	**Secret**: none.	

4	MESSAGE	**Responsibility**: stores information pertaining to a message sent by a user to a group in the system.	**Alternative**: none
	Concrete	**Secret**: none.	

4. Expanded description of design decisions

When I was first considering designs for my version of the messenger application, I came across many interesting design patterns including the pub sub design pattern, which includes publishers and subscribers to content, and in a way, my design is a modification of this already established design pattern.

The MESSENGER is the most important module in my design, and it holds all the core features of the application. It holds all the users, groups, and messages stored in the system and stores them in lists appropriate for their end display. When I was formulating designs in my head, there was no way around some sort of central "engine" class that would coordinate other smaller classes to have this application run properly. This is because there needs to be a central location where all users, groups and messages reside since users may be registered for no groups and groups may have no members, but we still need to have a record of them. Also, all the messages need to be stored here since in the message status section of the output, the message availability needs to be displayed for all users, not just ones who received the messages from the groups that they are registered in. Because users and groups need to be sorted in a certain way for output, I chose to store them in a sorted two-way list. The messages do not need this sorted functionality so I stored them in a linked list.

As mentioned above, the MESSENGER class is a sort of centralized repository for the data that is input into the system. This centralized repository is needed to have all necessary data in one place so comparisons can be made in the various queries within the class, and so the appropriate error messages can be displayed back to the client when they attempt to enter certain commands. For example, in the query message_id_exists, I would not be able to check if the message exists for the specific user without having all users and all messages available to myself in a central accessible location since a user with that id might not even exist and likewise for the message id. If I were to only have groups I would need to go through every group, and every user inside that group, but what if that user is not registered in a group? This is my justification for this essential class in my system.

One of the key aspects of this class is that when a message is being sent to a group, the sender does not need to know all the users that the message will be sent to. Instead, the messenger sends the message to the group specified, and the group sends the message to all its users. In this way, some of the load is taken off the user and simplicity is maintained when attempting to send messages to various groups.

5. Significant Contracts (Correctness)

The most significant contracts in my messenger system occur within the MESSENGER class of my application. Many of these contracts are ensured by the defensive programming that occurs in each of the ETF command classes while some do not.

The add_user and add_group command in the MESSENGER class both have three preconditions: the id given must be greater than 0, the first letter of the name entered must be an alphabetical letter, and the id must not already be in use in the system. With this precondition, it is guaranteed that the user or group is added to the system and that their information is stored in the central repository (hidden from the client through abstraction). Once the group or user has been added to the system, it is guaranteed through these preconditions that groups or users cannot have the same id as the one that has just entered the system, ensuring that no bugs in that sense occur in the application.

The delete_message command requires that both the user id and the message id are greater than 0, the user id exists in the system (that is, it exists within the central repository that is the MESSENGER class), the message id exists for that specific user (the user must have access to that message), and that message that exists for the user must be an old message for it to be deleted. To check if the message id exists for a specific user, a query is called to check if the user with that user id has a message with that specific message id. If they do, the message must be read before it is deleted (which is specified by the message being in the old messages of the user). From these preconditions, the user is ensured that the message will be deleted from that specific user's "inbox", however the central repository of messages will stay the same, since the other users that may be registered in the group that the message was sent to might not have read the message yet. This again emphasizes the need for a central class to handle this functionality.

The register_user command requires that both the user id and the group id be greater than 0, both the user id and the group id must exist in the central repository, and finally there must not already be a registration between the user with that user id and the group with that group id. When satisfying these preconditions, it is ensured that the user is not only registered to that group, but that group adds that user to its subscriber list to notify them when it receives incoming messages. This is a core functionality of the messaging system.

The read_message command requires that both the user id and the message id are greater than 0, the user id exists in the central repository, and the message is available to that specific user. It also requires that the user is authorized to access that message. This ensures that users that do not have access to certain messages do not see those messages (creating privacy breaches in the system). Finally, it is required that the message is not already read for that specific user. After these preconditions are satisfied, it is guaranteed that the user will read that specific

message in their system, marking that message as read for that one user only. Keep in mind that when one user reads a message it is only read for them, not the entire group.

The send_message command requires that the user id and the group id be greater than 0, both the group id and the user id must exist in the central repository, the message body must be nonempty, and the user with the id specified is authorized to send messages to the group with the specified group id (that is, they are registered in that group and the registration exists in the system). When these preconditions are satisfied, the message is created as a MESSAGE object, and the message number is increased to ensure that the message id is unique. The group is then found in the system and the message is passed on to the group, where it will be broadcast to all its subscribers. The message is also added to the all messages central repository of messages in the MESSENGER class to ensure that a record of all the messages sent in the system is kept.

Finally, the set_message_preview command simply requires that the length that is specified be greater than 0, since a message preview length of 0 would mean that the message would not be shown in the output (which would be meaningless). When this precondition is met, the message length preview is set and the display will be updated accordingly from then on.

These contracts form the backbone of the main functionality of the system, and all the commands that are in the system. Without these contracts, design by contract would not be able to be maintained, and I personally would not have the reassurance that my system is working bug free.

.

6. Summary of Testing Procedures

Test file	Description	Passed
at1.txt	This test file goes through normal scenarios with the messenger and goes through some of the errors and bad input that it might receive during its execution. It also goes through some of the errors in sending messages are functioning properly. This test also checks that the set message preview functionality is working properly.	✓
at2.txt	This test goes through some normal inputs and checking to see if messages that are longer than the default 15-character limit are truncated properly. It also uses the list new and old message commands as well as the read message and delete message commands to ensure that they are functioning properly.	✓
at3.txt	This short test checks to see the output when a message is deleted and then the user tries to read that deleted message. It also lists the old messages after the message has been deleted to ensure that the application is functioning the same as the oracle program in that respect. When the message is deleted and the user tries to read the message, there should be an error message saying that the message is unavailable and when the user lists old messages there should be no old messages for the user.	✓
at4.txt	This test attempts to see what happens when a user attempts to read a message from a group they are not registered in and then reading the message after they have registered to the group, to see the various outputs that the oracle produces in these various situations.	✓
at5.txt	This test attempts to add a group with a very long id to test the limit of the id system that is put in place in my system. The system must handle ids that are greater than zero and are in the limit of a 64-bit integer.	✓
at1.txt (instructor)	This test file goes through the normal scenarios where the messenger application is used in a variety of ways, including errors. This test file covers the most basic functionality that the messenger application should have.	✓

7. Appendix (Contract view of all classes)

```
-- Automatic generation produced by ISE Eiffel --
note
description: "Summary description for {MESSENGER}."
author: "Matthew MacEachern"
date: "03/30/2017
revision: "$Revision$"

class interface
MESSENGER

create
make

feature -- attributes
message_to_read: STRING_8
message_number: INTEGER_64
message_length: INTEGER_64
sort_by_id: BOOLEAN
num_users: INTEGER_64
num_groups: INTEGER_64

feature -- creation
make

feature -- commands
add_user (id: INTEGER_64; name: STRING_8)
-- adds user with id 'id' and name 'name' to the messenger
require
id_positive: id > 0
first_letter_alpha: not (name.count = 0) and name.at (1).is_alpha
id_not_in_use: not user_id_exists (id)

add_group (id: INTEGER_64; name: STRING_8)
-- adds group with id 'id' and name 'name' to the messenger
require
id_positive: id > 0
first_letter_alpha: name.count > 0 and name.at (1).is_alpha
id_not_in_use: not group_id_exists (id)

delete_message (uid: INTEGER_64; mid: INTEGER_64)
-- user with id 'uid' deletes message with id 'mid' from their "history"
require
uid > 0 and mid > 0
user_id_exists (uid)
message_id_exists (uid, mid)
old_message_exists (uid, mid)

register_user (uid: INTEGER_64; gid: INTEGER_64)
-- register user with id 'uid' to group with id 'gid'
require
uid > 0 and gid > 0
user_id_exists (uid)
group_id_exists (gid)
not registration_exists (uid, gid)

read_message (uid: INTEGER_64; mid: INTEGER_64)
-- user with id 'uid' reads message with id 'mid'
require
uid > 0 and mid > 0
user_id_exists (uid)
message_id_exists (uid, mid)
user_authorized_to_access_message (uid, mid)
not message_read (uid, mid)
```

```
send_message (uid: INTEGER_64; gid: INTEGER_64; txt: STRING_8)
-- user with id 'uid' sends a message to group with id 'gid' containing text 'txt'
require
uid > 0 and gid > 0
user_id_exists (uid)
group_id_exists (gid)
no_empty_message: not txt.is_empty
authorization: registration_exists (uid, gid)

set_message_preview (length: INTEGER_64)
--sets the preview length of all messages in the messenger to length
require
length > 0
ensure
message_length = length

feature -- queries
user_authorized_to_access_message (uid: INTEGER_64; mid: INTEGER_64): BOOLEAN
-- checks to see if user with id 'uid' is authorized to access the message with id 'mid'
user_id_exists (id: INTEGER_64): BOOLEAN
-- checks to see if the user with id 'id' exists in the messenger
user_no_new_message (id: INTEGER_64): BOOLEAN
-- checks to see if the user with id 'id' has no old messages
user_no_old_message (id: INTEGER_64): BOOLEAN
-- checks to see if the user with id 'id' has no old messages
group_id_exists (id: INTEGER_64): BOOLEAN
-- group with id 'id' exists in the messenger
message_id_exists (uid: INTEGER_64; mid: INTEGER_64): BOOLEAN
--check to see if this message id exists for this user
list_all_messages: STRING_8
--lists all of the messages sent
list_new_messages (uid: INTEGER_64): STRING_8
-- lists the new messages for user with id 'uid'
require
uid > 0
user_id_exists (uid)
not user_no_new_message (uid)

list_old_messages (uid: INTEGER_64): STRING_8
-- lists the old messages for user with id 'uid'
require
uid > 0
user_id_exists (uid)
not user_no_old_message (uid)

message_status: STRING_8
--lists the message status for each message and each user
message_deleted (uid: INTEGER_64; mid: INTEGER_64): BOOLEAN
-- checks to see if user with id 'uid' has deleted a message with id 'mid'
user_member_of_group (uid: INTEGER_64; mid: INTEGER_64): BOOLEAN
-- checks to see if user with id 'uid' is a member of the group
-- that message with id 'mid' is sent to
message_read (uid: INTEGER_64; mid: INTEGER_64): BOOLEAN
list_users_by_id: STRING_8
--lists the users in order of their id
list_users: STRING_8
--lists the users in order of their name
list_groups_by_id: STRING_8
--lists the groups in order of their id
list_groups: STRING_8
--lists the users in order of their name
require
num_groups > 0

list_registrations: STRING_8
-- returns a formatted list of the registrations for the default output
```

```
old_message_exists (uid: INTEGER_64; mid: INTEGER_64): BOOLEAN
-- user with id 'uid' has an old message with id 'mid'

registration_exists (uid: INTEGER_64; gid: INTEGER_64): BOOLEAN
-- registration exists between user with id 'uid' and group with id 'gid'

end -- class MESSENGER
-- Generated by ISE Eiffel --
-- For more details: http://www.eiffel.com --
```

```
note
description: "Summary description for {GROUP}."
author: "Matthew MacEachern"
date: "03/30/2017"
revision: "$Revision$"

class interface
GROUP

create
make

feature --creation
make (l_name: STRING_8; l_id: INTEGER_64; l_m: MESSENGER)

feature --attributes
name: STRING_8
id: INTEGER_64
messenger: MESSENGER
subscribers: LINKED_LIST [USER]

feature --queries
out: STRING_8
-- New string containing terse printable representation
-- of current object

feature --commands
subscribe (u: USER)
--subscribe user u to this group
broadcast_message (m: MESSAGE)
--broadcast a message sent to this group to all of the subscribers

feature --comparable
is_less alias "<" (other: like Current): BOOLEAN
-- Is current object less than `other'?
end -- class GROUP
-- Generated by ISE Eiffel --
-- For more details: http://www.eiffel.com --
```

```
note
description: "Summary description for {USER}."
author: "Matthew MacEachern"
date: "03/30/2017
revision: "$Revision$"
class interface
USER
create
make
feature --creation
make (l_name: STRING_8; l_id: INTEGER_64; l_m: MESSENGER)

feature --attributes
name: STRING_8
id: INTEGER_64
messenger: MESSENGER
groups: SORTED_TWO_WAY_LIST [GROUP]
messages: SORTED_TWO_WAY_LIST [MESSAGE]
deleted_messages: SORTED_TWO_WAY_LIST [MESSAGE]
message_status: HASH_TABLE [BOOLEAN, INTEGER_64]
--message id to read status for this user
old_messages: SORTED_TWO_WAY_LIST [MESSAGE]
--lists the old messages for this user
new_messages: SORTED_TWO_WAY_LIST [MESSAGE]
--lists the new messages for this user

feature --commands
add_message (m: MESSAGE)
--add message m to users "inbox"
delete_message (mid: INTEGER_64)
--delete message with id mid from user
read_message (mid: INTEGER_64)
--read message with id mid
register (g: GROUP)
--register this user to group g

feature --queries
authorized_to_access_message (mid: INTEGER_64): BOOLEAN
--Check to see if user is able to access message mid
has_message (mid: INTEGER_64): BOOLEAN
--checks if this user has a message with mid
no_new_message: BOOLEAN
--Checks to see if this user has any new messages
no_old_message: BOOLEAN
--Checks to see if this user has any old messages
out: STRING_8
-- New string containing terse printable representation
-- of current object
list_new_messages: STRING_8
--lists new messages for this user in a readable format
list_old_messages: STRING_8
--lists old messages for this user in a readable format
list_registrations: STRING_8
--list this user's registrations in a readable format
number_of_groups: INTEGER_64
--number of groups this user is a part of
member_of (gid: INTEGER_64): BOOLEAN
--is this user a member of group with id gid?
message_id_exists (mid: INTEGER_64): BOOLEAN
--does the user have access to a message with id mid?
message_deleted (mid: INTEGER_64): BOOLEAN
--checks to see if a message with id mid was deleted by this user

feature --comparable
is_less alias "<" (other: like Current): BOOLEAN
-- Is current object less than `other'?
end -- class USER
```

```
note
description: "Summary description for {MESSAGE}."
author: "Matthew MacEachern"
date: "03/30/2017"
revision: "$Revision$"

class interface
MESSAGE

create
make

feature --creation
make (num: INTEGER_64; s: INTEGER_64; g: INTEGER_64; c: STRING_8; m: MESSENGER)

feature --attributes
number: INTEGER_64
sender: INTEGER_64
group: INTEGER_64
content: STRING_8
messenger: MESSENGER

feature --queries
is_less alias "<" (other: like Current): BOOLEAN
-- Is current object less than `other'?
out: STRING_8
-- New string containing terse printable representation
-- of current object
end -- class MESSAGE
-- Generated by ISE Eiffel --
-- For more details: http://www.eiffel.com --
```